O'REILLY®

Python

Разработка на основе тестирования

Гарри Персиваль

ДМК Пресс
издательство

Гарри Персиваль

Python. Разработка на основе тестирования

Повинуйся Билли-тестировщику, используя Django, Selenium и JavaScript

Test-Driven Development with Python

Obey the Testing Goat: Using Django, Selenium, and JavaScript

Harry J.W. Percival

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Python. Разработка на основе тестирования

Повинуйся Билли-тестировщику, используя Django, Selenium и JavaScript

Гарри Персиваль

Москва, 2018 ДМК ПРЕСС издательство

УДК 373.167.1:004.42+004.42(075.3)
ББК 32.973.721
П27

П27 Персиваль Г.

Python. Разработка на основе тестирования. / пер. с англ. Логунов А. В. – М.: ДМК Пресс, 2018. – 622 с.: ил.

ISBN 978-5-97060-594-3

Книга демонстрирует преимущества методологии разработки на основе тестирования (TDD) на языке Python. Вы научитесь писать и выполнять тесты для создания любого фрагмента вашего приложения и затем разрабатывать минимальный объем программного кода, необходимого для прохождения этих тестов. Вы также научитесь работать с различными инструментами и фреймворками, такими как Django, Selenium, Git, jQuery и Mock.

Издание предназначено всем разработчикам, кто уже освоил начальный уровень программирования на Python и хочет перейти на следующий.

УДК 373.167.1:004.42+004.42(075.3)
ББК 32.973.721

ISBN 978-1-49195-870-4 (англ.)
ISBN 978-5-97060-594-3 (рус.)

Оглавление

Предисловие 16

Предпосылки и предположения 21

Сопутствующее видео 30

Признательности 31

Часть I. Основы TDD и Django 33

Глава 1. Настройка Django с использованием функционального теста 34

Повинуйтесь Билли-тестировщику! Ничего не делайте, пока у вас не будет теста 34
Приведение Django в рабочее состояние 37
Запуск репозитория Git 40

Глава 2. Расширение функционального теста при помощи модуля unittest 44

Использование функционального теста для определения минимального дееспособного приложения 44
Модуль unittest стандартной библиотеки Python 47
Фиксация 50

Глава 3. Тестирование простой домашней страницы при помощи модульных тестов 52

Первое приложение Django и первый модульный тест 53
Модульные тесты и их отличия от функциональных тестов 53
Модульное тестирование в Django 55
MVC в Django, URL-адреса и функции представления 56
Наконец-то! Мы на самом деле напишем прикладной код! 58
Файл urls.py 59
Модульное тестирование представления 62
Цикл «модульный-тест/программный-код» 64

Глава 4. И что же делать со всеми этими тестами (и рефакторизацией)? 68

Программировать – все равно что поднимать ведро с водой из колодца 69
Использование Selenium для тестирования взаимодействий пользователя 71
Правило «Не тестировать константы» и шаблоны во спасение 74
Перестройка программного кода для использования шаблона 75
Тестовый клиент Django 79
О рефакторизации 81
Еще немного о главной странице 83
Резюме: процесс TDD 85

Глава 5. Сохранение вводимых пользователем данных: тестирование базы данных 88

Подключение формы для отправки POST-запроса ... 89
Обработка POST-запроса на сервере ... 92
Передача переменных Python для вывода в шаблоне ... 93
Если клюнуло трижды, рефакторизуй ... 98
Django ORM и первая модель ... 100
Первая миграция базы данных ... 102
Тест продвинулся удивительно далеко ... 103
Новое поле означает новую миграцию ... 104
Сохранение POST-запроса в базу данных ... 105
Переадресация после POST-запроса ... 108
Лучшие приемы модульного тестирования: каждый тест должен проверять одну единицу кода ... 109
Генерирование элементов в шаблоне ... 110
Создание производственной базы данных при помощи команды migrate ... 112
Резюме ... 115

Глава 6. Усовершенствование функциональных тестов: обеспечение изоляции и удаление методов sleep ... 118

Обеспечение изоляции в функциональных тестах ... 118
Выполнение только модульных тестов ... 122
Ремарка: обновление Selenium и Geckodriver ... 123
О неявных и явных ожиданиях и методе time.sleep ... 124

Глава 7. Работа в инкрементном режиме ... 130

Маломасштабные конструктивные изменения по мере необходимости ... 130
В первую очередь маломасштабные конструктивные изменения ... 131
Вам это никогда не понадобится! ... 131
REST (-овский) подход ... 132
Инкрементная реализация новой структуры кода на основе TDD ... 133
Обеспечение теста на наличие регрессии ... 134
Итеративное движение в сторону новой структуры кода ... 137
Первый самодостаточный шаг: один новый URL-адрес ... 139
Новый URL-адрес ... 140
Новая функция представления ... 140
Зеленый? Рефакторизуй! ... 142
Еще один шажок: отдельный шаблон для просмотра списков ... 143
Третий шажок: URL-адрес для добавления элементов списка ... 146
Тестовый класс для создания нового списка ... 146
URL-адрес и представление для создания нового списка ... 148
Удаление теперь уже избыточного кода и тестов ... 149
Регрессия! Наведение форм на новый URL-адрес ... 149
Стиснув зубы: корректировка моделей ... 151
Связь по внешнему ключу ... 153
Адаптация остальной части мира к новым моделям ... 155
Каждому списку – свой URL-адрес ... 157
Извлечение параметров из URL-адресов ... 158
Адаптирование new_list к новому миру ... 160

Функциональные тесты обнаруживают еще одну регрессию ... 161
Еще одно представление для добавления элементов в существующий список ... 162
Остерегайтесь «жадных» регулярных выражений! ... 163
Последний новый URL-адрес ... 164
Последнее новое представление ... 165
Тестирование контекстных объектов отклика напрямую ... 166
Финальная рефакторизация с использованием URL-включений ... 168

Часть II. Непременные условия веб-разработки ... 171

Глава 8. Придание привлекательного вида: макет, стилевое оформление сайта и что тут тестировать ... 172

Что функционально тестируется в макете и стилевом оформлении ... 172
Придание привлекательного вида: использование платформы CSS ... 176
Наследование шаблонов в Django ... 178
Интеграция платформы Bootstrap ... 180
Строки и столбцы ... 180
Статические файлы в Django ... 182
Переход на StaticLiveServerTestCase ... 183
Использование компонентов Bootstrap для улучшения внешнего вида сайта ... 184
Класс jumbotron ... 184
Большие поля ввода ... 185
Стилистическое оформление таблицы ... 185
Использование собственного CSS ... 185
О чем мы умолчали: collectstatic и другие статические каталоги ... 187
Несколько вещей, которые не удались ... 190

Глава 9. Тестирование развертывания с использованием промежуточного сайта ... 191

TDD и опасные зоны развертывания ... 192
Как всегда, начинайте с теста ... 194
Получение доменного имени ... 196
Ручное обеспечение работы сервера для размещения сайта ... 196
Выбор места размещения сайта ... 197
Запуск сервера ... 197
Учетные записи пользователей, SSH и полномочия ... 198
Инсталляция Nginx ... 198
Инсталляция Python 3.6 ... 200
Конфигурирование доменов для промежуточного и реального серверов ... 200
Использование ФТ для подтверждения, что домен работает и Nginx выполняется ... 201
Развертывание исходного кода вручную ... 201
Корректировка расположения базы данных ... 202
Создание Virtualenv вручную и использование requirements.txt ... 204
Простое конфигурирование Nginx ... 206
Создание базы данных при помощи команды migrate ... 209
Победа! Наше хакерское развертывание работает ... 210

Глава 10. Переход к развертыванию, готовому к эксплуатации 212
Переход на Gunicorn 212
Настройка Nginx для раздачи статических файлов 214
Переход на использование сокетов Unix 215
Присвоение DEBUG значения False и настройка ALLOWED_HOSTS 216
Применение Systemd для проверки, что Gunicorn запускается на начальной загрузке 217
Сохранение изменений: добавление Gunicorn в файл requirements.txt 218
Размышления об автоматизации 219
Сохранение шаблонов конфигурационных файлов этапа обеспечения работы 219
Сохранение хода выполнения 222

Глава 11. Автоматизация развертывания с помощью Fabric 224
Описание частей сценария Fabric для развертывания 225
Создание структуры каталогов 226
Получение исходного кода из репозитория командой git 226
Обновление файла settings.py 227
Обновление virtualenv 229
Миграция базы данных при необходимости 229
Испытание автоматизации 230
Развертывание на работающем сайте 231
Конфигурирование Nginx и Gunicorn при помощи sed 234
Маркировка релиза командой git tag 235
Дополнительные материалы для чтения 236

Глава 12. Разделение тестов на многочисленные файлы и обобщенный помощник ожидания 238
Начало с ФТ валидации данных: предотвращение пустых элементов 238
Пропуск теста 239
Разбиение функциональных тестов на несколько файлов 241
Выполнение только одного файла с тестами 244
Новый инструмент функционального тестирования: обобщенная вспомогательная функция явного ожидания 245
Завершение ФТ 249
Рефакторизация модульных тестов на несколько файлов 251

Глава 13. Валидация на уровне базы данных 254
Валидация на уровне модели 255
Контекстный менеджер self.assertRaises 255
Причуда Django: сохранение модели не выполняет валидацию 256
Выведение на поверхность ошибок валидации модели в представлении 257
Проверка, чтобы недопустимые входные данные не сохранялись в базе данных 261
Схема Django: обработка POST-запросов в том же представлении, которое генерирует форму 263
Рефакторизация: передача функциональности new_item в view_list 264
Обеспечение валидации модели в view_list 267
Рефакторизация: удаление жестко кодированных URL-адресов 269

Тег {% url %} шаблона ... 269
Использование get_absolute_url для переадресаций ... 270

Глава 14. Простая форма ... 274

Перемещение программной логики валидации из формы ... 274
Исследование API форм при помощи модульного теста ... 275
Переход на Django ModelForm ... 277
Тестирование и индивидуальная настройка валидации формы ... 278
Использование формы в представлениях ... 281
Использование формы в представлении с GET-запросом ... 281
Глобальный поиск и замена ... 282
Использование формы в представлении, принимающем POST-запросы ... 285
Адаптация модульных тестов к представлению new_list ... 285
Использование формы в представлении ... 287
Использование формы для отображения ошибок в шаблоне ... 287
Использование формы в другом представлении ... 288
Вспомогательный метод для нескольких коротких тестов ... 289
Неожиданное преимущество: бесплатная валидация на стороне клиента из HTML5 ... 291
Одобряющее похлопывание по спине ... 293
Не потратили ли мы уйму времени впустую? ... 294
Использование собственного для формы метода save ... 295

Глава 15. Более развитые формы ... 298

Еще один ФТ на наличие повторяющихся элементов ... 298
Предотвращение дубликатов на уровне модели ... 299
Небольшое отступление по поводу упорядочивания Queryset и представлений строковых значений ... 301
Новое написание старого теста модели ... 304
Некоторые ошибки целостности проявляются при сохранении ... 306
Экспериментирование с проверкой на наличие повторяющихся элементов на уровне представлений ... 307
Более сложная форма для проверки на наличие повторяющихся значений ... 308
Использование существующей формы для элемента списка в представлении для списка ... 310
Итоги: что мы узнали о тестировании Django ... 313

Глава 16 Пробуем окунуться, очень робко, в JavaScript ... 316

Начинаем с ФТ ... 316
Настройка элементарного исполнителя тестов на JavaScript ... 318
Использование элемента div для jQuery и фикстуры ... 320
Создание модульного теста на JavaScript для требуемой функциональности ... 324
Фикстуры, порядок выполнения и глобальное состояние: ключевые проблемы тестирования на JS ... 326
console.log для отладочной распечатки ... 326
Использование функции инициализации для большего контроля над временем выполнения ... 328
Коломбо говорит: стереотипный код для onload и организация пространства

имен ... 330
Тестирование на Javascript в цикле TDD ... 332
Несколько вещей, которые не удалось сделать ... 332

Глава 17. Развертывание нового программного кода ... 334

Развертывание на промежуточном сервере ... 334
Развертывание на реальном сервере ... 335
Что делать, если вы видите ошибку базы данных ... 335
Итоги: маркировка нового релиза командой git tag ... 335

Часть III. Основы TDD и Django ... 337

Глава 18. Аутентификация пользователя, импульсное исследование и внедрение его результатов ... 338

Беспарольная аутентификация ... 338
Разведочное программирование, или Импульсное исследование ... 339
Открытие ветки для результатов импульсного исследования ... 340
Авторизация на стороне клиента в пользовательском интерфейсе ... 341
Отправка электронных писем из Django ... 341
Использование переменных окружения для предотвращения секретов в исходном коде ... 344
Хранение маркеров в базе данных ... 344
Индивидуализированные модели аутентификации ... 345
Завершение индивидуализированной авторизации в Django ... 346
Внедрение результатов импульсного исследования ... 351
Возвращение импульсного исходного кода в прежний вид ... 353
Минимальная индивидуализированная модель пользователя ... 354
Тесты в качестве документирования ... 357
Модель маркера для соединения электронных адресов с уникальным идентификатором ... 358

Глава 19. Использование имитаций для тестирования внешних зависимостей или сокращения дублирования ... 362

Перед началом: получение базовой инфраструктуры ... 362
Создание имитаций вручную, или Обезьянья заплатка ... 363
Библиотека Mock ... 367
Использование unittest.patch ... 368
Продвижение ФТ чуть дальше вперед ... 371
Тестирование инфраструктуры сообщений Django ... 371
Добавление сообщений в HTML ... 374
Начало с URL-адреса для входа в систему ... 375
Подтверждение отправки пользователю ссылки с маркером ... 376
Создание индивидуализированного серверного процессора аутентификации на основе результатов импульсного исследования ... 378
Еще один тест для каждого случая ... 379
Метод get_user ... 382

Использование серверного процессора аутентификации в представлении входа в систему ... 384
Еще одна причина использовать имитации: устранение повторов ... 385
Использование mock.return_value ... 388
Установка заплатки на уровне класса ... 390
Момент истины: пройдет ли ФТ? ... 392
Теоретически – работает! Работает ли на практике? ... 394
Завершение ФТ, тестирование выхода из системы ... 396

Глава 20. Тестовые фикстуры и декоратор для явных ожиданий ... 399

Пропуск регистрации в системе путем предварительного создания сеанса ... 399
Проверка работоспособности решения ... 402
Финальный вспомогательный метод явного ожидания: декоратор ожидания ... 405

Глава 21. Отладка на стороне сервера ... 410

Чтобы убедиться в пудинге, надо его попробовать: использование предварительного сервера для отлавливания финальных дефектов ... 410
Настройка журналирования ... 411
Установка секретных переменных окружения на сервере ... 413
Адаптация ФТ для тестирования реальных электронных писем POP3 ... 414
Управление тестовой базой данных на промежуточном сервере ... 418
Управляющая команда Django для создания сеансов ... 418
Настройка ФТ для выполнения управляющей команды на сервере ... 420
Использование fabric напрямую из Python ... 421
Резюме: создание сеансов локально по сравнению с промежуточным сервером ... 422
Внедрение программного кода журналирования ... 424
Итоги ... 425

Глава 22. Завершение приложения «Мои списки»: TDD с подходом «снаружи внутрь» ... 427

Альтернатива: «изнутри наружу» ... 427
Почему «снаружи внутрь» предпочтительнее? ... 428
ФТ для «Моих списков» ... 428
Внешний уровень: презентация и шаблоны ... 431
Спуск на один уровень вниз к функциям представления (к контроллеру) ... 431
Еще один проход снаружи внутрь ... 433
Быстрая реструктуризация иерархии наследования шаблонов ... 433
Конструирование API при помощи шаблона ... 434
Спуск к следующему уровню: что именно представление передает в шаблон ... 436
Следующее техническое требование из уровня представлений: новые списки должны записывать владельца ... 437
Момент принятия решения: перейти к следующему уровню с неработающим тестом или нет ... 438
Спуск к уровню модели ... 439
Финальный шаг: подача .name API из шаблона ... 441

Глава 23. Изоляция тестов и «слушание своих тестов» ... 444

Пересмотр точки принятия решения: уровень представлений зависит от ненаписанного кода моделей ... 444
Первая попытка использования имитаций для изоляции ... 446
Использование имитации side_effects для проверки последовательности событий ... 447
Слушайте свои тесты: уродливые тесты сигнализируют о необходимости рефакторизации ... 449
Написание тестов по-новому для представления, которое будет полностью изолировано ... 450
Держите рядом старый комплект интегрированных тестов в качестве проверки на токсичность ... 450
Новый комплект тестов с полной изоляцией ... 451
Мышление с точки зрения взаимодействующих объектов ... 452
Спуск вниз на уровень форм ... 457
Продолжайте слушать свои тесты: удаление программного кода ORM из нашего приложения ... 458
Наконец, спуск вниз на уровень моделей ... 462
Назад к представлениям ... 464
Момент истины (и риски имитации) ... 466
Точка зрения о взаимодействиях между уровнями как о контрактах ... 467
Идентификация неявных контрактов ... 468
Исправление недосмотра ... 469
Еще один тест ... 471
Наведение порядка: что следует убрать из комплекта интегрированных тестов ... 472
Удаление избыточного кода на уровне форм ... 472
Удаление старой реализации представления ... 473
Удаление избыточного кода на уровне форм ... 474
Выводы: когда писать изолированные тесты, а когда – интегрированные ... 475
Пусть вычислительная сложность будет вашим гидом ... 476
Следует ли делать оба типа тестов? ... 477
Вперед! ... 477

Глава 24. Непрерывная интеграция ... 479

Инсталляция сервера Jenkins ... 479
Конфигурирование сервера Jenkins ... 481
Первоначальная разблокировка ... 481
Набор плагинов на первое время ... 481
Конфигурирование пользователя-администратора ... 482
Добавление плагинов ... 483
Указание серверу Jenkins, где искать Python 3 и Xvfb ... 483
Завершение с HTTPS ... 484
Настройка проекта ... 484
Первая сборка! ... 485
Установка виртуального дисплея, чтобы ФТ выполнялись бездисплейно ... 487
Взятие снимков экрана ... 489
Если сомневаетесь – встряхните тайм-аут! ... 492
Выполнение тестов JavaScript QUnit на Jenkins вместе с PhantomJS ... 494
Установка node ... 494

Добавление шагов сборки в Jenkins ... 495
Больше возможностей с CI-сервером ... 497

Глава 25. Социально зачимый кусок, шаблон проектирования «Страница» и упражнение для читателя ... 499

ФТ с многочисленными пользователями и addCleanup ... 499
Страничный шаблон проектирования ... 501
Расширение ФТ до второго пользователя и страница «Мои списки» ... 504
Упражнение для читателя ... 506

Глава 26. Быстрые тесты, медленные тесты и горячий поля ... 509

Тезис: модульные тесты сверхбыстры и к тому же хороши ... 511
Более быстрые тесты означают более быструю разработку ... 511
Священное состояние потока ... 511
Медленные тесты не выполняются часто, что приводит к плохому коду ... 512
Теперь у нас все в порядке, но интегрированные тесты со временем становятся медленнее ... 512
Не верьте мне ... 512
И модульные тесты управляют хорошей структурой кода ... 512
Проблемы с «чистыми» модульными тестами ... 512
Изолированные тесты труднее читать и писать ... 512
Изолированные тесты не тестируют интеграцию автоматически ... 513
Модульные тесты редко отлавливают неожиданные дефекты ... 513
Тесты с имитациями могут стать близко привязанными к реализации ... 513
Но все эти проблемы могут быть преодолены ... 514
Синтез: что мы хотим от всех наших тестов? ... 514
Правильность ... 514
Чистый код, удобный в сопровождении ... 514
Продуктивный поток операций ... 515
Оценивайте свои тесты относительно преимуществ, которые вы хотите от них получить ... 515
Архитектурные решения ... 516
Порты и адаптеры/шестиугольная/чистая архитектура ... 516
Функциональное ядро, императивная оболочка ... 517
Заключение ... 518
Дополнительные материалы для чтения ... 518

Повинуйтесь Билли-тестировщику! ... 521

Тестировать очень тяжело ... 521
Держите свои сборки CI-сервера на зеленом уровне ... 521
Гордитесь своими тестами так же, как своим программным кодом ... 522
Не забудьте дать на чай персоналу заведения ... 522
Не пропадайте! ... 522

Приложение A. PythonAnywhere ... 523

Выполнение сеансов Firefox Selenium при помощи Xvfb ... 523
Настройка Django как веб-приложение PythonAnywhere ... 525
Очистка папки /tmp ... 526

Снимки экрана ... 526
Глава о развертывании ... 526

Приложение B. Представления на основе классов в Django ... 528

Обобщенные представления на основе классов ... 528
Домашняя страница как FormView ... 529
Использование form_valid для индивидуализации CreateView ... 530
Более сложное представление для обработки просмотра и добавления к списку ... 533
Сравните старую верию с новой ... 536
Лучшие приемы модульного тестирования обобщенных представлений на основе классов ... 537

Приложение C. Обеспечение работы серверной среды при помощи Ansible ... 539

Установка системных пакетов и Nginx ... 539
Конфигурирование Gunicorn и использование обработчиков для перезапуска служб ... 541
Что делать дальше ... 542

Приложение D. Тестирование миграций базы данных ... 544

Попытка развертывания на промежуточном сервере ... 544
Выполнение тестовой миграции локально ... 545
Вставка миграции данных ... 546
Совместное тестирование новых миграций ... 547
Выводы ... 548

Приложение E. Разработка на основе поведения (BDD) ... 550

Что такое BDD? ... 550
Базовые служебные операции ... 551
Написание ФТ как компонента при помощи синтаксиса языка Gherkin ... 552
Программирование шаговых функций ... 553
Определение первого шага ... 555
Эквиваленты setUp и tearDown в environment.py ... 555
Еще один прогон ... 556
Извлечение параметров в шагах ... 556
BDD по сравнению с ФТ с локальными комментариями ... 559
BDD способствует написанию структурированного тестового кода ... 561
Страничный шаблон проектирования как альтернатива ... 561
BDD может быть менее выразительным, чем локальные комментарии ... 563
Будут ли непрограммисты писать тесты? ... 563
Некоторые предварительные выводы ... 564

Приложение F. Создание REST API: JSON, Ajax и имитирование на JavaScript ... 565

Наш подход для этого раздела ... 565
Выбор подхода к тестированию ... 566
Организация базовой конвейерной передачи ... 566
Получение фактического отклика ... 568

Добавление POST-метода ... 569
Тестирование клиентского Ajax при помощи Sinon.js ... 570
Соединение всего в шаблоне, чтобы убедиться, что это реально работает ... 574
Реализация Ajax POST-запроса, включая маркер CSRF ... 576
Имитация в JavaScript ... 578
Валидация данных. Упражнение для читателя ... 582

Приложение G. Django-Rest-Framework ... 587

Инсталляция ... 587
Сериализаторы (на самом деле – объекты ModelSerializer) ... 588
Объекты Viewset (на самом деле – объекты ModelViewset) и объекты Router ... 589
Другой URL для элемента с POST-запросом ... 592
Адаптирование на стороне клиента ... 593
Что дает инфраструктура Django-Rest-Framework ... 595

Приложение H. Шпаргалка ... 599

Начальная настройка проекта ... 599
Основной поток операций TDD ... 599
Выход за пределы тестирования только на сервере разработки ... 600
Общие приемы тестирования ... 601
Лучшие приемы на основе Selenium / функциональных тестов ... 601
«Снаружи внутрь», изоляция тестов против интегрированных тестов и имитация ... 602

Приложение I. Что делать дальше ... 603

Уведомления – на сайте и по электронной почте ... 603
Переходите на Postgres ... 604
Выполняйте тесты относительно разных браузеров ... 604
Тесты на коды состояния 404 и 500 ... 604
Сайт администратора Django ... 604
Напишите несколько тестов защищенности ... 605
Тест на мягкую деградацию ... 605
Кэширование и тестирование и производительности ... 605
MVC-инфраструктуры для JavaScript ... 605
Async и протокол Websocket ... 606
Перейдите на использование py.test ... 606
Попробуйте coverage.py ... 606
Шифрование на стороне клиента ... 606
Здесь место для ваших предложений ... 607

Приложение J. Примеры исходного кода ... 608

Полный список ссылок для каждой главы ... 608
Использование Git для проверки вашего прогресса ... 610
Скачивание ZIP-файла для главы ... 611
Не позволяйте этому превращаться в костыль! ... 611

Предметный указатель ... 612

Предисловие

Эта книга стала результатом моей попытки рассказать миру о путешествии, которое я начал с «чистого хакерства» и в итоге пришел к «программной инженерии». В основном речь пойдет о тестировании, но повествование коснется многих других аспектов, в чем вы скоро сами убедитесь.

Я хочу поблагодарить вас за то, что взялись прочитать мою книгу.

Если вы ее приобрели, то я очень благодарен. Если вы читаете бесплатную онлайн-версию, я *по-прежнему* благодарен за то, что вы сочли ее достойной потраченного времени. Кто знает, возможно, добравшись до конца, вы решите, что она достаточно хороша, чтобы приобрести печатное издание для себя или друзей.

Если у вас есть какие-либо комментарии, вопросы или предложения, буду рад получить от вас известие. Меня можно найти непосредственно по ссылке obeythetestinggoat@gmail.com либо в Твиттере @hjwp. Можно также зайти на мой веб-сайт и блог (http://www.obeythetestinggoat.com/), есть также список рассылки[1].

Надеюсь, от прочтения настоящей книги вы получите удовольствие в той же степени, что и я, когда писал ее.

Почему я посвятил книгу разработке на основе тестирования

«Кто ты такой, почему ты пишешь эту книгу и почему я должен ее прочитать?» – спросите вы.

Я все еще нахожусь в самом начале карьеры программиста. Говорят, в любой дисциплине человек проходит путь от ученика до подмастерья и в конечном счете (иногда) становится мастером. Должен сказать, что я, в лучшем случае, программист-путешественник. Но в самом начале своей карьеры мне посчастливилось попасть в группу фанатиков методологии TDD, и это оказало такое огромное влияние на мой стиль программирования, что я испытываю желанием поделиться им со всеми. Можно сказать, горю энтузиазмом новообращенного – опыт обучения свеж в моей памяти, поэтому я надеюсь, что все еще способен хорошо понимать новичков.

Когда я приступил к изучению Python (опираясь на превосходную книгу Марка Пилгрима *«Погружение в Python»*), я столкнулся с понятием TDD и подумал: «Определенно, в этом есть смысл». Возможно, у вас была подобная реакция, когда вы впервые услышали про TDD? Этот подход и правда выглядит разумным, действительно хорошая привычка – все равно что регулярно пользоваться зубной нитью или что-то в этом роде.

---

[1] См. https://groups.google.com/forum/%23!forum/obey-the-testing-goat-book

Затем подоспел мой первый большой проект. Вы можете догадаться, что произошло: клиент, сроки, уйма работы – и все мои благие намерения по поводу TDD отправились прямиком в урну.

По сути, все шло прекрасно. Я был в порядке.

Сначала.

Я понимал, что в сущности TDD мне не нужен, потому что это был небольшой веб-сайт и я мог в ручном режиме легко проверить, что все работает. Нажми на ссылку *здесь*, выбери из выпадающего списка *там* и получи *это* – все просто! Вся эта затея с написанием тестов, казалось, будет занимать *вечность*. Кроме того, с высоты своих трех недель зрелого опыта программирования я мнил себя довольно хорошим программистом. Я мог справляться с работой. Легко.

Затем пробил час грозной богини Сложности. Она-то и показала мне пределы моего опыта.

Проект рос. Одни части системы начинали зависеть от других. Я прилагал все усилия, чтобы следовать хорошим принципам, таким как DRY (Don't Repeat Yourself – Не повторяйся), но в итоге он завел меня на довольно опасную территорию. Вскоре я уже был в окружении множественного наследования. Иерархии классов глубиной восемь уровней. Операторов `eval`.

Я начал испытывать страх перед внесением изменений в свой код. Я больше не был уверен, что понимаю, что от чего зависит и что может произойти, если изменить этот код в этом месте. Проклятие! Кажется, вот тот кусок наследует отсюда. Да нет же, он переопределен! Стоп, так он же зависит вон от той переменной класса. Верно! Ну, в общем, пока я переопределяю переопределение, все должно прекрасно работать. Просто проверю и все.

Но проверка становилась все труднее. Теперь у моего сайта было много разделов, и щелканье по ним вручную становилось непрактичным. Лучше все оставить в покое, забыть о рефакторизации и обойтись тем, что есть.

Вскоре у меня уже была отвратительная, уродливая мешанина кода. Внесение в него новых наработок стало болезненным.

Некоторое время спустя мне посчастливилось получить задание от компании Resolver Systems (теперь это PythonAnywhere[2]), где экстремальное программирование (XP) было нормой. Они представили меня строгой методологии TDD.

Несмотря на то что предыдущий опыт, конечно же, открыл мой ум для возможных преимуществ автоматизированного тестирования, я все еще волочил ноги на каждом этапе. «Признаю, тестирование в целом может быть неплохой идеей, но постойте! Все эти тесты? Некоторые из них выглядят как тотальная трата времени… Что? Функциональные тесты *вместе с* модульными? Да ладно вам, ну вы загнули! И весь этот цикл TDD «тест/

[2] См. https://www.pythonanywhere.com/

минимальное-изменение-кода/тест»... Это же просто глупо! Кому нужны все эти крохотные шажочки! Давайте-ка посмотрим, чей ответ верный и просто перейдем к финалу!»

Поверьте, я подвергал сомнению каждое правило, предлагал всякого рода короткие пути, требовал обоснований по любому, на первый взгляд бессмысленному, аспекту TDD... В итоге я увидел в этом здравый смысл. Я потерял счет числу раз, когда я ловил себя на мысли: «Спасибо вам, тесты», поскольку функциональный тест раскрывает регрессию, которую мы бы никогда не предсказали, и модульный тест спасает меня от действительно глупой логической ошибки. В психологическом отношении методология TDD сделала разработку гораздо менее напряженным процессом. Она порождает программный код, работать с которым одно удовольствие.

Так что позвольте рассказать вам об этом *все*, что я знаю!

Цели этой книги

Моя главная цель состоит в том, чтобы донести до вас методологию, как выполнять веб-разработку, которая, думаю, позволяет создавать самые лучшие веб-приложения и делает разработчиков счастливее. Не много толка от книги, предлагающей материал, который вы легко можете найти в Гугле. Так что эта книга *как таковая* не является руководством по синтаксису Python или учебным руководством по веб-разработке. Вместо этого я надеюсь научить вас использовать методологию TDD, чтобы более надежным образом прийти к нашей общей священной цели: *чистому коду, который работает*.

С учетом всего сказанного, я постоянно буду обращаться к реальным практическим примерам, создавая веб-приложение с нуля с использованием таких инструментов, как Django, Selenium, jQuery и Mock. Я не исхожу из ваших предварительных знаний ни об одном из них, поэтому вам следует обратиться к концу этой книги, где вы найдете достаточное введение в эти инструменты, а также в дисциплину TDD.

В экстремальном программировании мы всегда имеем дело с парным программированием, и, занимаясь написанием данной книги, я представлял, будто общаюсь в паре с самим собой прошлым, объясняя, каким образом работают инструменты, и отвечая на вопросы, почему мы программируем именно так, а не иначе. Так что если я когда-нибудь приму нечто вроде покровительственного тона, это только потому, что я вовсе не такой умный и должен себя хорошенько сдерживать. А если начну оправдываться, то потому, что я похож на зануду, который систематически не соглашается, кто что бы ни говорил, поэтому иногда приходится приводить много обоснований, чтобы убедиться в чем-либо.

Схема

Я разделил эту книгу на три части:

Первая часть (главы 1–7) – основы

Погружает прямо в создание простого веб-приложения с использованием методологии TDD. Мы начинаем с написания функционального теста (при помощи Selenium), затем проходим основы Django – модели, представления, шаблоны – со строгим модульным тестированием на каждом этапе. Также я представляю Билли-тестировщика.

Вторая часть (главы 8–17) – существенные аспекты веб-разработки

Охватывает некоторые более сложные, но неизбежные аспекты веб-разработки; показывает, как тестирование может в этом помочь: статические файлы, развертывание в производственной среде, валидация данных в формах, миграция баз данных и ужасающий JavaScript.

Часть III (главы 18–26) – более продвинутые темы тестирования

Применение объектов-имитаций, интеграция сторонней системы, тестовые фикстуры, TDD с подходом «снаружи-внутрь» и непрерывная интеграция (CI).

Переходим к некоторым служебным операциям…

Условные обозначения, принятые в книге

В книге используются следующие условные обозначения:

Курсив

Указывает новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

`Моноширный шрифт`

Используется для распечаток программ, а также внутри абзацев для ссылки на элементы программ, такие как переменные или имена функций, базы данных, типы данных, переменные окружающей среды, операторы и ключевые слова.

`Жирный моноширный шрифт`

Показывает команды либо другой текст, который должен быть напечатан пользователем, а также ключевые слова в программном коде.

Иногда я буду использовать символ:

`[...]`
: чтобы обозначить пропущенную часть содержимого с целью сократить куски результатов работы или перейти к соответствующему разделу.

Данный элемент обозначает подсказку или совет.

Общее замечание или ремарка.

Предупреждение или предостережение.

Предпосылки и предположения

Вот краткое описание того, что должен знать читатель этой книги и какое программное обеспечение ему потребуется.

Python 3 и программирование

Я попытался написать эту книгу, ориентируясь на начинающих программистов, но если вы в программировании новичок, то я исхожу из того, что вы уже изучили основы Python. Поэтому, если вы еще не приступили, убедительно советую прочесть учебник по Python для начинающих или ознакомительную книгу, например «*Погружение в Python*» (Dive Into Python) или «*Python на горьком опыте*» (Learn Python the Hard Way) или только для развлечения «*Придумывайте свои компьютерные игры вместе с Python*» (Invent Your Own Computer Games with Python) – все эти книги являются превосходными введениями[1].

Если вы – опытный программист, но новичок в Python, вы отлично поладите. Python удивительно доступен для понимания.

В настоящей книге я использую *Python 3*. На тот момент, когда я писал ее в 2013–2014 годах, Python 3 уже существовал несколько лет, и тогда мир стоял перед выбором: какой версии Python отдать предпочтение. Вы сможете руководствоваться текстом книги, работая в Mac, Windows или Linux. Подробные инструкции по инсталляции в каждой ОС следуют ниже.

Эта книга была протестирована для Python 3.6. Если у вас более ранняя версия, вы найдете незначительные различия (синтаксис f-string, например), поэтому будет лучше, если вы обновитесь, по возможности.

Я бы не рекомендовал пытаться использовать Python 2, поскольку различия между двумя версиями весьма существенные. Вы, конечно, сможете применить все знания, полученные из этой книги, если ваш следующий проект окажется на основе Python 2. Но временные затраты на выяснение, в чем причина того, что результаты работы вашей программы отличаются от моих – Python 2 или же все-таки фактическая ошибка, которую вы допустили сами, – не будут продуктивными.

Если вы раздумываете, использовать *PythonAnywhere* (PaaS-стартап, на который я работаю)[2] или локально установленный Python, то перед началом работы вам следует пройти в конец книги и ознакомиться с Приложением А.

---

[1] См. библиографию – *Прим. перев.*

[2] См. https://www.pythonanywhere.com/

В любом случае вам потребуется доступ к Python и вы должны знать, как запускать его из командной строки, как редактировать файлы Python и их выполнять. И еще раз: взгляните на те три книги, которые я порекомендовал ранее, если вы находитесь в сомнении.

Если у вас уже установлен Python 2 и вы переживаете, что установка Python 3 как-то ему навредит, не беспокойтесь! Python 3 и 2 могут сосуществовать мирно в одной системе, особенно при использовании virtualenv, которым мы воспользуемся.

Как работает HTML

Также я исхожу, что у вас есть базовые знания о том, как работает веб: что такое HTML, POST-запрос и т. д. Если вы в этом не уверены, изучите элементарное руководство по HTML; некоторые можно найти на http://www.webplatform.org/. Если вы способны создать страницу HTML на ПК, просмотреть ее в браузере, а также понимаете, что такое форма и как она может работать, то у вас, похоже, все в порядке.

Django

Настоящая книга использует программную инфраструктуру (фреймворк) Django, которая является, пожалуй, самой известной в мире системой разработки веб-приложений на Python. Я написал книгу, исходя из того, что у читателя нет предварительных знаний о Django, но если вы новичок в Python *и* веб-разработке *и* тестировании, можете порой замечать, что приходится пробовать и принимать на борт слишком много тем и наборов понятий, чтобы их все охватить. Если это так, рекомендую делать перерывы в чтении книги и переключаться на учебники по Django. DjangoGirls – лучший из них, это самое дружественное пособие для новичков, которое я знаю. Официальное учебное руководство превосходно подойдет для более опытных программистов[3].

Смотрите ниже инструкции по установке Django.

JavaScript

Во второй половине книги мы кратко остановимся на JavaScript. Если вы новичок в JavaScript, не переживайте: если вы окажетесь в небольшом замешательстве, то в той части книги я порекомендую несколько соответствующих руководств.

[3] См. https://tutorial.djangogirls.org/ и https://docs.djangoproject.com/en/1.11/intro/tutorial01/

Примечание по IDE

Если вы пришли из мира Java или .NET, возможно, вам будет интересно использовать интегрированную среду разработки (IDE) для программирования на Python. Они располагают разнообразными полезными инструментами, включая интеграцию с системой управления версиями (VCS). Среди них есть несколько превосходных, предназначенных как раз для Python. Я сам использовал одну из них, когда только начинал, она очень пригодилась для моих первых двух проектов.

Могу ли я полагагать (и это только предположение), что вы не используете IDE, по крайней мере на время чтения этого учебного пособия? IDE гораздо менее востребованы в мире Python, и я написал об этом целую книгу, допустив, что вы просто используете элементарный текстовый редактор и командную оболочку. Иногда это все, что есть (когда вы работаете с сервером, например), поэтому всегда сначала стоит учиться использовать базовые инструменты, понимать, как они работают. Эти инструменты будут всегда под рукой, даже если после изучения этой книги вы решите вернуться к вашему IDE и всем его полезным инструментам.

Инсталлируемое программное обеспечение

Кроме Python, вам понадобятся:

Веб-браузер Firefox
: Selenium может управлять любым из основных браузеров, но для примера лучше всего воспользоваться Firefox, потому что он доказал свою надежную кросс-платформенность и, в качестве бонуса, меньше всего продается корпоративным клиентам.

Система управления версиями Git
: Она доступна для любой платформы и расположена на `http://git-scm.com/`. В Windows она поставляется вместе с командной оболочкой **Bash**, которая необходима для настоящей книги.

virtualenv с Python 3, Django 1.11 вместе с Selenium 3
: Python'овская среда окружения virtualenv и инструменты менеджера пакетов pip поставляются в комплекте начиная с Python 3.4+ (так было не всегда, так что гип-гип-ура). Подробные инструкции по подготовке virtualenv последуют далее.

Geckodriver
: Это драйвер, который позволит нам удаленно управлять Firefox через Selenium. Я дам ссылку на скачивание в разделе «Инсталляция Firefox» ниже.

Примечания по поводу Windows

В мире с открытым исходным кодом пользователи Windows иногда могут чувствовать себя немного обделенными вниманием, поскольку OS X и Linux там настолько доминируют, что легко забывается о существовании мира вне парадигмы Unix. Обратные каталожные разделители? Буквенные метки дисков? Что?! Тем не менее вполне можно следовать по тексту книги, работая в Windows. Вот несколько подсказок:

1. Устанавливая Git для Windows, убедитесь, что вы выбрали **Run Git and included Unix tools from the Windows command prompt** (Выполнять Git вместе с включенными средствами Unix для командной строки Windows). Далее вы получите доступ к программе Git Bash. Используйте ее в качестве основной командной оболочки, и вы получите все полезные инструменты командной строки GNU, такие как `ls`, `touch` и `grep`, плюс прямые каталожные разделители.
2. Также в установщике Git выберите **Use Windows' default console** (Использовать консоль Windows по умолчанию), иначе Python не будет работать должным образом в окне Git-Bash.
3. Когда вы устанавливаете Python 3 при уже установленном Python 2 и хотите его сохранить в качестве среды по умолчанию, выберите опцию **Add Python 3.6 to PATH** (Добавьте Python 3.6 к системному пути PATH), как на рис. p-1, чтобы можно было легко выполнять Python из командной строки.

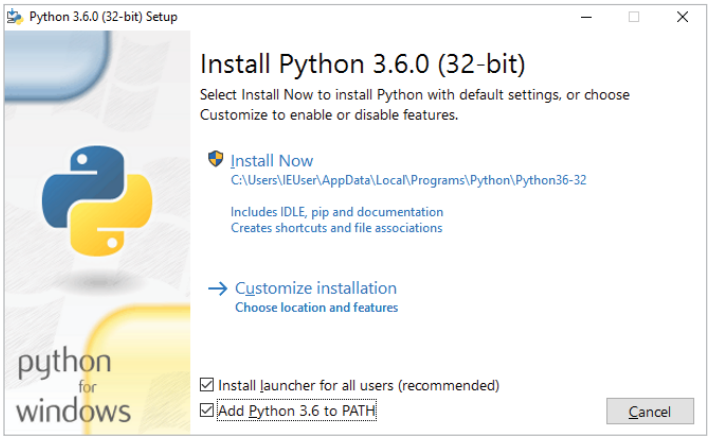


Рис. p-1. Добавьте Python к системному пути из установщика

Проверьте, что все это сделано: вы можете открывать командную оболочку Git-Bash и запускать Python или pip из любой папки.

Примечания по поводу MacOS

MacOS немного более вменяема, чем Windows, несмотря на то что до сего времени установка `pip` по-прежнему представляла определенную сложность. С момента появления версии 3.4 все стало довольно прямолинейным:

- Python 3.6 установится без суеты при помощи загружаемого установщика (http://www.python.org/). Он автоматически установит `pip`.
- Установщик Git тоже должен сразу заработать.

Так же как в Windows, проверкой работоспособности будет то, что вы можете открыть терминал и просто выполнить `git`, `python3` или `pip` из любого места. Если вы столкнетесь с какой-либо проблемой, то поиск «системный путь» и «команда не найдена» должны обеспечить хорошие ресурсы для устранения неисправностей.

Возможно, вы захотите обратиться к Homebrew (http://brew.sh/). Раньше он был единственным надежным средством установки в Mac[4] большого количества инструментов Unixy (включая Python 3). Несмотря на то что с нормальным установщиком Python теперь все в порядке, Homebrew может пригодиться в будущем. Правда, он потребует скачать весь 1,1 Гб XCode, но он также предоставит вам компилятор C, который будет полезным побочным продуктом.

Редактор Git по умолчанию и другие базовые настройки Git

Я предоставлю пошаговые инструкции для Git, но сейчас предлагаю немного заняться конфигурированием. Например, когда вы в первый раз будете фиксировать (комитить) внесенные изменения в репозитории, по умолчанию появится текстовый редактор *vi*, и вы, возможно, не будете иметь понятия, что с ним сделать. Так вот, поскольку vi имеет два режима, у вас, соответственно, будет два варианта. Первый – изучить минимальное количество команд vi (*нажать клавишу i, чтобы войти в режим вставки, набрать текст, нажать* `<Esc>` *для возвращения в нормальный режим, затем записать файл и выйти при помощи* `:wq<Enter>`). В результате вы присоединитесь к большому братству людей, которые знают этот древний, уважаемый текстовый редактор.

Или же вы можете решительно отказаться участвовать в таком смехотворном откате к 1970-м и сконфигурировать Git для использования редактора по вашему выбору. Выйдите из vi при помощи клавиши `<Esc>`, со-

[4] Правда, я бы не рекомендовал устанавливать Firefox посредством Homebrew :`brew` помещает бинарный файл Firefox в непонятное место, и это приводит Selenium в замешательство. Вы сможете работать в обход этого, но проще просто установить Firefox обычным способом.

провождаемой `:q!`, затем поменяйте на свой редактор Git по умолчанию. Смотрите документацию по Git относительно процедуры базовой конфигурации Git[5].

Инсталляция Firefox и Geckodriver

Firefox можно скачать для Windows и OSX с https://www.mozilla.org/firefox/. В Linux вы, вероятно, уже его установили, в противном случае ваш диспетчер пакетов будет его иметь .

Geckodriver доступен на https://github.com/mozilla/geckodriver/releases. Вам нужно его скачать, извлечь и поместить где-нибудь в системном пути:

- в OSX или Linux я рекомендую положить его в `~/.local/bin`;
- в Windows положите его в вашу папку Python «Scripts».

Чтобы проверить, что все работает, откройте консоль Bash, и у вас должно получиться выполнить следующее:

```
geckodriver --version
geckodriver 0.17.0

The source code of this program is available at
https://github.com/mozilla/geckodriver.

This program is subject to the terms of the Mozilla Public License 2.0.
You can obtain a copy of the license at https://mozilla.org/MPL/2.0/.
```

Если это не работает, возможно, `~/.local/bin` не находится в вашем системном пути `PATH` (это относится к некоторым системам Mac и Linux). Неплохо иметь эту папку в вашем системном пути, потому что именно туда Python будет устанавливать компоненты, когда вы будете использовать команду `pip install --user`. Вот как добавить его в ваш `.bashrc`:

```
echo 'PATH=~/.local/bin:$PATH' >> ~/.bashrc
```

Закройте свой терминал, откройте снова и убедитесь, что `geckodriver --version` теперь работает.

Настройка виртуальной среды virtualenv

В Python виртуальная среда virtualenv (сокращенно от virtual environment) – это то, как вы настраиваете свою программную среду для различных проектов Python. Она позволяет использовать в каждом проекте разные пакеты: например, разные версии Django и даже разные версии Python.

[5] См. http://git-scm.com/book/en/Customizing-Git-Git-Configuration.

И поскольку вы не устанавливаете компоненты в масштабе всей системы, это означает, что вам не требуются полномочия root.

Virtualenv входит в Python с версии 3.4, но я всегда рекомендую вспомогательный инструмент под названием virtualenvwrapper. Давайте сначала установим его (не важно, для какой версии Python вы его устанавливаете):

```
# в Windows
pip install virtualenvwrapper
# в MacOS / Linux
pip install --user virtualenvwrapper
# затем дайте Bash загрузить virtualenvwrapper автоматически
echo "source virtualenvwrapper.sh" >> ~/.bashrc
source ~/.bashrc
```

В Windows `virtualenvwrapper` будет работать только в оболочке Git-Bash, не из обычной командной строки.

Virtualenvwrapper хранит все ваши виртуальные среды virtualenv в одном месте и обеспечивает удобные инструменты для их активации и деактивации.

Давайте создадим виртуальную среду superlists[6], в которой установлен Python 3:

```
# в MacOS/Linux:
mkvirtualenv --python=python3.6 superlists
# в Windows
mkvirtualenv --python=`py -3.6 -c"import sys; print(sys.executable)"`
superlists
# (небольшой трюк, чтобы убедиться, что у нас virtualenv в версии python 3.6)
```

Активация и деактивация virtualenv

Каждый раз во время работы с книгой у вас будет возникать желание убедиться, что ваша виртуальная среда virtualenv активна. Об этом можно узнать по тому, что в командной строке вы увидите `(superlists)` в скобках. Что-то вроде этого:

```
$
(superlists) $
```

Сразу после создания виртуальной среды virtualenv она будет активной. Это можно перепроверить, выполнив `which python`:

[6] Слышу, как вы говорите: «Почему superlists?» Никаких спойлеров! Вы узнаете в следующей главе.

```
(superlists) $ which python
/home/harry/.virtualenvs/superlists/bin/python
# (в Windows это будет что-то вроде
# /C/Users/IEUser/.virtualenvs/superlists/Scripts/python)

(superlists) $ deactivate
$ which python
/usr/bin/python
$ python --version
Python 2.7.12 # у меня вне virtualenv команда python по умолчанию покажет Python 2.

$ workon superlists
(superlists) $ which python
/home/harry/.virtualenvs/superlists/bin/python
(superlists) $ python --version
Python 3.6.0
```

Чтобы активировать виртуальную среду virtualenv, нужно выполнить команду `workon superlists`. Чтобы проверить, активна ли она, ищите в командной строке `(superlists) $` либо выполните `which python`.

Инсталляция Django и Selenium

Теперь мы установим Django 1.11 и последнюю версию Selenium – Selenium3.

```
(superlists) $ pip install "django==1.12" "selenium<4"
Collecting django==1.11.3
  Using cached Django-1.11.3-py2.py3-none-any.whl
Collecting selenium>3
  Using cached selenium-3.4.3-py2.py3-none-any.whl
Installing collected packages: django, selenium
Successfully installed django-1.11.3 selenium-3.4.3
```

Некоторые сообщения об ошибках, которые вы, вероятно, увидите, когда *неминуемо* провалите активацию вашего virtualenv

Если вы новичок в virtualenv или, по правде говоря, даже если нет, в какой-то момент вы *гарантированно* забудете ее активировать и уставитесь в сообщение об ошибке. Со мной это случается постоянно. Вот примеры того, что вы увидите:

```
ImportError: No module named selenium
```

Или:

```
ImportError: No module named django.core.management
```

Как всегда, смотрите, чтобы в вашей командной оболочке было `(superlists)` и быстрая команда `workon superlists`, вероятно, позволит вернуть среду в рабочее состояние.

Вот еще парочка для пущей достоверности:

```
bash: workon: command not found
```

Это сообщение означает, что вы перешли на шаг раньше и не добавили virtualenvwrapper к своему *.bashrc*. Выше найдите команды `echo source virtualenvwrapper.sh` и выполните их повторно.

```
'workon' is not recognized as an internal or external command,
operable program or batch file.
```

Вы запустили стандартную командную оболочку Windows – `cmd` – вместо Git-Bash. Закройте ее и откройте последнюю.

Успешного программирования!

Помогли ли вам эти инструкции? Или у вас есть кое-что получше? Свяжитесь со мной по адресу: obeythetestinggoat@gmail.com!

Сопутствующее видео

В качестве сопровождения для настоящей книги я записал видео из 10 частей[1]. Видео охватывает содержание глав с 1 по 6. Если вы лучше усваиваете видеоматериал – приглашаю вас обратиться к нему. В дополнение к тому, что имеется в книге, этот материал позволит вам почувствовать, что собой представляет "поток" TDD, дав возможность переключаться между тестами и кодом, получать объяснения мыслительного процесса по мере продвижения.

Плюс я ношу восхитительную желтую футболку.

[1] См. http://oreil.ly/1svTFqB.

Признательности

Следует поблагодарить многих людей, без которых эта книга никогда бы не появилась и/или была бы еще хуже, чем есть.

Огромное спасибо в первую очередь Грегу. Он был первым, кто вселил в меня уверенность, что она может быть осуществлена. Несмотря на то что у его работодателей оказались чрезмерно регрессивные представления об авторском праве, я всегда буду благодарен, что он в меня поверил.

Спасибо Майклу Фурду – еще одному бывшему сотруднику Resolver Systems – за то, что он стал примером человека, который первым решился написать книгу, и спасибо за его непрекращающуюся поддержку данного проекта. Также хочу поблагодарить своего босса Гайлса Томаса за то, что сглупил, позволив еще одному из своих сотрудников написать книгу (хотя, полагаю, он теперь внес в стандартный трудовой договор поправку «никаких книг»). Ему также спасибо за непрекращающееся здравомыслие и за то, что поставил меня на путь тестирования.

Благодарю своих других коллег, Гленна Джоунса и Хансела Данлопа, за то, что были моими неоценимыми пробными слушателями, и за их терпение к моему заезженному разговору весь прошлый год.

Спасибо моей жене Клементине и обеим моим семьям, без поддержки и терпения которых я бы никогда не справился. Прошу прощения за все то время, когда я с головой погружался в компьютере, вместо участия в незабываемых семейных событиях. Когда я пустился во все тяжкие, я понятия не имел, что эта книга сделает с моей жизнью («Пиши ее в свое свободное время, говоришь? Надо подумать…»). Вероятно, я не написал бы ее без вас.

Благодарю своих технических рецензентов, Джонатана Хартли, Николаса Толлерви и Эмили Бах, за их поддержку и бесценную обратную связь. В особенности Эмили, которая в действительности добросовестно прочитала каждую главу. Частичная похвала Нику и Джону, но она все же должна рассматриваться как вечная благодарность. Ваше присутствие сделало всю эту затею не такой одинокой. Без вас эта книга была бы не более чем бессмысленной болтовней идиота.

Спасибо всем, кто уделил мне часть своего времени, чтобы поделиться мнением о книге просто так, по доброте душевной: Гэри Бернхард, Марк Лавин, Матт Одоннел, Майкл Фурд, Хинек Шлавак, Рассел Кейт-Магей, Эндрю Годуин, Кеннет Рейтц и Натан Стокс. Спасибо, что вы намного умнее меня и что не дали мне сказать несколько глупых вещей. Разумеется, в книге осталось еще много глупостей, за которые вы все не можете нести абсолютно никакой ответственности.

Благодарю своего редактора Меган Бланшетт за то, что была очень дружелюбным и симпатичным надзирателем, за то, что следила за процессом написания книги с точки зрения сроков и ограничения моих более глупых идей. Благодарю всех остальных в издательстве O'Reilly за их советы, в том числе Сару Шнайдер, Кару Эбраим и Дэна Фокссмита за том, что дали возможность сохранить британский вариант английского языка. Благодарю Чарльза Румелиотиса за его советы относительно стиля и грамматики. Мы никогда не сойдемся во взглядах насчет достоинств правил цитирования/пунктуации Чикагской школы, но, должен заве-

рить, я рад, что вы были рядом. И спасибо отделу дизайна за предоставленного нам животного для обложки!

Самую большую благодарность хочу высказать всем моим читателям предварительного выпуска книги за выявленные опечатки, советы и предложения, за все то, чем они помогли сгладить кривую обучения в книге, и больше всего – за их добрые слова ободрения и поддержки, которые помогали мне в работе. Спасибо Jason Wirth, Dave Pawson, Jeff Orr, Kevin De Baere, crainbf, dsisson, Galeran, Michael Allan, James O'Donnell, Marek Turnovec, SoonerBourne, julz, Cody Farmer, William Vincent, Trey Hunner, David Souther, Tom Perkin, Котелок Sorcha, Jon Poler, Charles Quast, Siddhartha Naithani, Steve Young, Roger Camargo, Wesley Hansen, Johansen Christian Vermeer, Ian Laurain, Sean Robertson, Хари Jayaram, Bayard Randel, Konrad Korżel, Matthew Waller, Julian Harley, Barry McClendon, Simon Jakobi, Angelo Cordon, Jyrki Kajala, джайн Manish, Mahadevan Sreenivasan, Konrad Korżel, Deric Crago, Cosmo Smith, Markus Kemmerling, Andrea Costantini, Daniel Patrick, Ryan Allen, Jason Selby, Greg Vaughan, Jonathan Sundqvist, Richard Bailey, Diane Soini, Dale Stewart, Mark Keaton, Johan Warlander, Simon Scarfe, Eric Grannan, MarcAnthony Taylor, Maria McKinley, John McKenna, Rafai Szymanski, Roel van der Goot, Ignacio Reguero, TJ Tolton, Jonathan Means, Theodor Nolte, Луна Jungsoo, Craig Cook, Gabriel Ewilazarus, Vincenzo Pandolfo, David "farbish2", Nico Coetzee, Daniel Gonzalez, Jared Contrascere, Zhao Mro и многих-многих других. Если я пропустил чье-то имя, вы имеете абсолютное право расстроиться; я невероятно благодарен и вам, так что напишите мне, и я попытаюсь загладить свою вину любым возможным способом.

И наконец благодарю вас, последний читатель, за то, что обратили внимание на эту книгу! Надеюсь, она вам понравится.

Дополнительные благодарности относительно второго издания

Большое спасибо моему замечательному редактору второго издания Нэну Барберу, а также Сьюзен Конант, Кристен Браун и всей команде O'Reilly. И еще раз выражаю свою благодарность Эмили и Джонатану за техническую рецензию, а также Эдварду Уонгу за его подробные замечания. Все остальные ошибки и несоответствия остаются на моей совести.

Также благодарю читателей свободного издания книги, которые внесли комментарии и предложения, а некоторые даже сделали запрос на включение кода. В этом списке я определенно мог кого-то из вас пропустить, так что прошу извинить, если вашего имени здесь нет, и тем не менее, благодарю Emre Gonulates, Jésus Gómez, Jordon Birk, James Evans, Iain Houston, Jason DeWitt, Ronnie Raney, Spencer Ogden, Suresh Nimbalkar, Darius, Caco, LeBodro, Jeff, wasabigeek, joegnis, Lars, Mustafa, Jared, Craig, Sorcha, TJ, Ignacio, Roel, Justyna, Nathan, Andrea, Alexandr, bilyanhadzhi, mosegontar, sfarzy, henziger, hunterji, das-g, juanriaza, GeoWill, Windsooon, gonulate и многих-многих других.

Часть I

Основы TDD и Django

В этой первой части я собираюсь представить азы разработки на основе тестирования (TDD от англ. Test-Driven Development). Мы разработаем реальное веб-приложение с нуля, на каждом этапе создавая сначала тесты.

Мы рассмотрим функциональное тестирование с использованием Selenium, а также модульное тестирование и увидим между ними разницу. Я представлю поток операций TDD – то, что я называю циклом «модульный-тест/программный-код». Мы выполним небольшую рефакторизацию и увидим, как она укладывается в TDD. Поскольку система управления версиями – абсолютно необходимый элемент серьезной программной инженерии, я буду также использовать Git. Мы обсудим, как и когда фиксировать изменения и интегрировать их с потоком операций веб-разработки и TDD.

Мы будем использовать Django – пожалуй, самую популярную в мире Python'овскую программную инфраструктуру для веб-разработки. Я старался представлять понятия Django медленно и по одному и приводить большое количество ссылок на дополнительные материалы для чтения. Если вы начинаете работать с Django с абсолютного нуля, убедительно рекомендую не торопиться с их изучением. Если вы почувствуете, что немного заблудились, уделите пару часов, чтобы просмотреть официальное учебное руководство по Django, а затем возвращайтесь к этой книге.

Вы также познакомитесь с Билли-тестировщиком…

Будьте осторожны с копированием и вставкой

Если вы работаете с цифровой версией книги, то вполне естественным будет желание копипастить распечатки программного кода из книги по мере того, как вы читаете. Будет намного лучше, если вы этого не будете делать: набор кода вручную остается в вашей мышечной памяти, да и сам по себе ощущается намного реальнее. Вы также неизбежно сделаете случайные опечатки, исправление которых является важной составной частью обучения.

Совершенно отдельно от этого вы обнаружите, что причуды формата PDF выражаются в том, что, когда вы пытаетесь из него скопировать/вставить, иногда происходят странные вещи…

Глава 1

Настройка Django с использованием функционального теста

TDD – это не то, что происходит естественным образом. Это дисциплина, подобная боевому искусству, и точно так же, как в фильме про кунг фу, вам нужен раздражительный и неблагоразумный наставник, который будет заставлять вас постигать дисциплину. Нашим наставником будет Билли-тестировщик.

Повинуйтесь Билли-тестировщику! Ничего не делайте, пока у вас не будет теста

Горный козел по прозвищу Билли-тестировщик – это неофициальный талисман TDD в сообществе тестирования Python. Для разных людей он, вероятно, имеет разное значение[1], но для меня Билли-тестировщик – это внутренний голос, который не дает мне сбиться с Истинного пути тестирования – подобно одному из тех ангелочков или дьяволят, которые выскакивают из-за плеча героев мультфильмов, но с очень своеобразным набором озабоченностей. Я надеюсь, что с помощью этой книги мне тоже удастся внедрить Билли-тестировщика и в вашу голову тоже.

Мы решили создать веб-сайт, хотя не совсем уверены, зачем он нужен. В веб-разработке обычно первый шаг состоит в инсталляции и конфигурировании вашей веб-платформы. *Скачайте это, установите то, сконфигурируйте другое, выполните сценарий…* Но TDD требует совсем другого образа мыслей. Когда вы выполняете TDD, у вас все время на уме Билли-тестировщик – целенаправленный, как все горные козлы, блеющий: «Сначала тест, сначала тест!»

В TDD первым шагом всегда является одно и то же: *написать тест*.

[1] Кашмирский козел является официальным талисманом пехотных подразделений Королевских валлийцев Британской армии – *Прим. перев.*

Сначала мы пишем тест, *затем* мы его выполняем и убеждаемся, что он не проходит, как ожидалось. И *только потом* мы идем вперед и создаем часть нашего приложения. Повторите это себе голосом Билли.

Еще одна вещь, которая касается сородичей нашего наставника, заключается в том, что они двигаются постепенно, шаг за шагом. Вот почему они редко падают с гор, гляньте: не важно, на какой крутизне они стоят. Как вы можете убедиться из рис. 1.1.

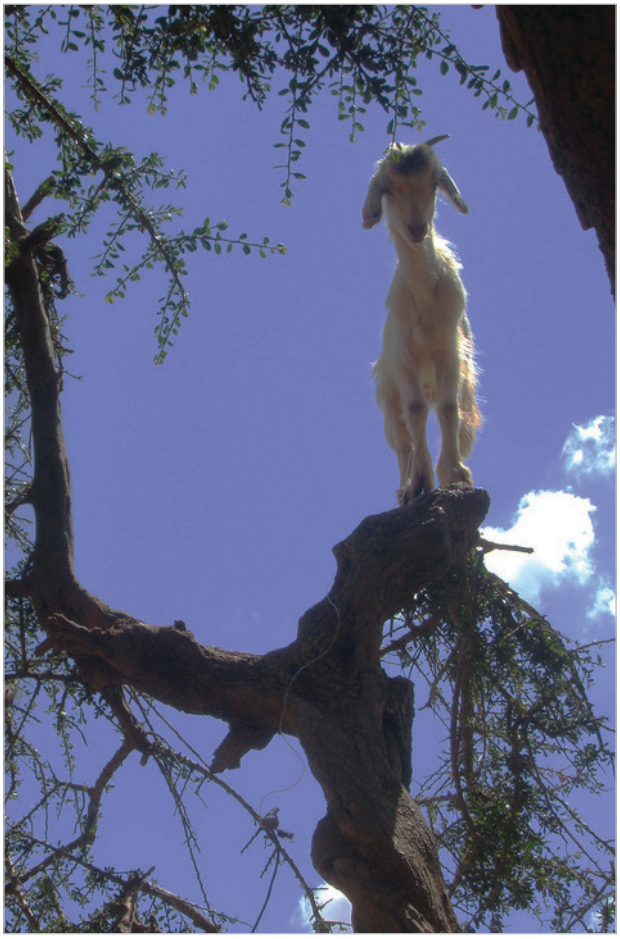


Рис. 1.1. Горные козлы гибче, чем вы думаете (Источник: Caitlin Stewart, на Flickr, http://www.flickr.com/photos/caitlinstewart/2846642630/)

Мы будем продвигаться небольшими шажками и для создания нашего приложения будем использовать Django – популярную программную инфраструктуру для разработки веб-приложений на Python.

Первое, что мы хотим сделать, – это проверить, что Django у нас установлена и готова к работе. *Способ* проверки сводится к подтверждению, что мы можем завести сервер разработки Django и воочию убедиться, что

он раздает веб-страницу на наш веб-браузер на нашем локальном ПК[2]. Для этого воспользуемся средством автоматизации браузеров *Selenium*.

Создайте новый файл Python под названием *functional_tests.py* в любом месте, где вы хотите хранить программный код своего проекта, и введите следующий исходный код. Если во время работы вы испытываете желание издать несколько восклицаний голосом Билли, то это наверняка поможет:

functional_tests.py

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get('http://localhost:8000')

assert 'Django' in browser.title
```

Это наш первый *функциональный тест* (ФТ); о том, что я подразумеваю под функциональными тестами и как они контрастируют с модульными тестами, я буду рассказывать много. Пока же достаточно просто убедиться, что мы понимаем, что он делает:

- Запускает «webdriver» Selenium, чтобы появилось окно реального браузера Firefox.
- Использует его для открытия веб-страницы, которая, как мы ожидаем, будет роздана из локального ПК.
- Проверяет (выполняя тестовое утверждение), что в заголовке страница есть слово «Django».

Попробуем его выполнить:

```
$ python functional_tests.py
  File ".../selenium/webdriver/remote/webdriver.py", line 268, in get
    self.execute(Command.GET, {'url': url})
  File ".../selenium/webdriver/remote/webdriver.py", line 256, in execute
    self.error_handler.check_response(response)
  File ".../selenium/webdriver/remote/errorhandler.py", line 194, in
check_response
    raise exception_class(message, screen, stacktrace)
selenium.common.exceptions.WebDriverException: Message: Reached error page: abo
ut:neterror?e=connectionFailure&u=http%3A//localhost%3A8000/[...]
```

[2] Здесь и далее в тексте процесс разработки конечного программного продукта подразделяется на три этапа: этап непосредственной разработки (development), этап подготовки к внедрению, или промежуточный этап (staging), и этап внедрения в производственную среду, или производственный этап (production). Соответственно, на каждом этапе используется свой сервер: сервер разработки, промежуточный сервер и производственный сервер. То же самое касается и версий проектируемого программного обеспечения, в данном случае веб-сайта: разрабатываемая, промежуточная и производственная версии – *Прим. перев.*

Вы увидите, как появится окно браузера и попытается открыть *localhost:8000* и показать страницу с ошибкой «Unable to connect» (Не в состоянии установить соединение). Если вы перейдете назад на свою консоль, то увидите большое уродливое сообщение об ошибке, говорящее о том, что Selenium попал на страницу ошибки. И затем вы, вероятно, будете раздражены тем, что этот тест оставил на вашем рабочем столе окно Firefox где попало, вынуждая вас наводить порядок. Этот вопрос мы уладим чуть позже!

Если вместо этого вы видите ошибку с попыткой импортировать Selenium или ошибку с попыткой найти "geckodriver", то вам, возможно, надо вернуться и еще раз взглянуть на главу «*Предпосылки и предположения*».

А пока у нас есть *неработающий тест*, а это значит, что мы можем приступить к созданию нашего приложения.

Прощайте, римские цифры!

Столько много предисловий к TDD используют римские цифры в качестве примера, что это стало ходячим анекдотом – одно такое я даже сам начинал писать. Если вам любопытно, вы можете найти его на моей странице в GitHub[3].

Римские цифры и хороши, и плохи одновременно. Это хорошая «игрушечная» задача, разумно ограниченная в объеме, и вы можете вполне хорошо объяснить ими TDD.

Проблема состоит в том, что ее бывает очень трудно увязать с реальным миром. Вот почему в качестве моего примера я решил обратиться к созданию реального веб-приложения, начиная с пустого места. Несмотря на то что это веб-приложение простое, надеюсь, вам будет проще его перенести в свой следующий реальный проект.

Приведение Django в рабочее состояние

Поскольку к настоящему моменту вы определенно прочли главу «*Предпосылки и предположения*», программная инфраструктура Django у вас уже инсталлирована. Первый шаг по приведению Django в рабочее состояние заключается в создании *проекта*, который будет основным контейнером для нашего сайта. Для этого Django обеспечивает небольшой инструмент командной строки:

```
$ django-admin.py startproject superlists
```

[3] См. https://github.com/hjwp/tdd-roman-numeral-calculator/

Эта команда создаст папку под названием *superlists* и ряд файлов и подпапок внутри нее:

```
.
├── functional_tests.py
├── geckodriver.log
└── superlists
        ├── manage.py
        └── superlists
                ├── __init__.py
                ├── settings.py
                ├── urls.py
                └── wsgi.py
```

Да, внутри папки *superlists* есть папка *superlists*. Немного сбивает с толку, но это только одна из таких вещей; тому существуют серьезные причины, если вы оглянетесь назад на историю Django. А пока важно знать, что папка *superlists/superlists* предназначена для вещей, которые относятся ко всему проекту – как, например, файл *settings.py*, используемый для хранения глобальной конфигурационной информации о сайте.

Вы также обратите внимание на файл *manage.py*. Для Django данный файл – это швейцарский нож, и часть из его функций заключается в выполнении сервера разработки. Теперь испытаем его. Командой **`cd superlists`** перейдите в корневую папку *superlists* (мы много будем работать из этой папки) и затем выполните:

```
$ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until
they are applied. Run 'python manage.py migrate' to apply them.

Django version 1.11, using settings 'superlists.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Пока ничего страшного, если проигнорировать это сообщение об «unapplied migrations» (о незадействованных миграциях). Мы обратимся к миграциям в главе 5.

Это тот самый сервер разработки Django, теперь он приведен в рабочее состояние на нашей машине. Оставьте его там и откройте другую команд-

ную оболочку. В ней мы сможем попытаться выполнить наш тест еще раз (из папки, в которой мы запустили):

```
$ python functional_tests.py
$
```

Поскольку вы только что открыли новое окно терминала, чтобы все заработало, вам придется активировать вашу виртуальную среду командой `virtualenv workon superlists`.

Совсем немного действий с командной строкой, но вы должны заметить две вещи: во-первых, нет уродливой ошибки `AssertionError` и, во-вторых, окно Firefox, которое Selenium показал, содержит совсем другую страницу.

Конечно, не так уж и много, но это был наш самый первый успешно выполненный тест! Ура!

Если все это выглядит почти как волшебство, будто этого не должно быть, почему бы не пойти и не взглянуть на сервер разработки вручную, открыв веб-браузер самостоятельно и посетив http://localhost:8000? Вы увидите нечто, как на рис. 1.2.

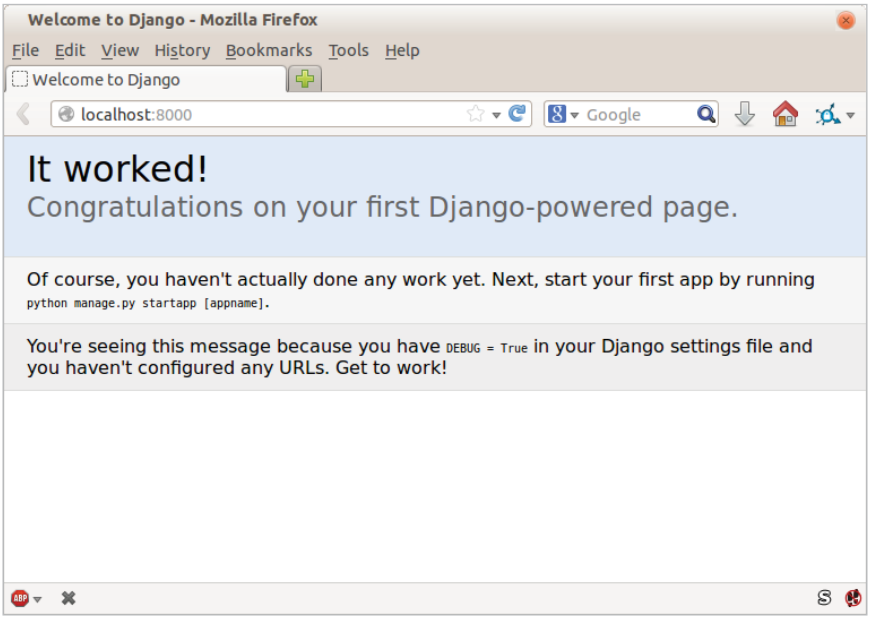


Рис. 1.2. Сработало!

Если хотите, можете вернуться из сервера разработки в исходную оболочку при помощи **Ctrl+C**.

Запуск репозитория Git

Прежде чем мы закончим главу, надо сделать еще одну вещь: начать передавать нашу работу *системе управления версиями* (VCS). Если вы опытный программист, вам не нужно слушать мои проповеди об управлении версиями. Но если вы плохо знакомы с этой системой, поверьте мне, что VCS – это необходимая вещь. Когда вашему проекту исполнится несколько недель и он вырастит до нескольких строк программного кода, иметь инструмент, который позволяет оглянуться назад, на старые версии кода, обратить изменения, безопасно исследовать новые идеи и даже просто выполнить резервное копирование… О боже! TDD идет рука об руку с управлением версиями, и я хочу быть уверенным, что правильно передаю мысль о том, как оно вписывается в поток операций.

Итак, наша первая фиксация изменений! Если немного поздновато, позор нам. В качестве системы управления версиями мы используем *Git*, потому что он – лучший.

Давайте начнем, переместив файл *functional_tests.py* в папку *superlists* и выполнив команду `git init` для запуска репозитория:

```
$ ls
superlists functional_tests.py geckodriver.log
$ mv functional_tests.py superlists/
$ cd superlists
$ git init .
Initialised empty Git repository in /.../superlists/.git/
```

Нашим рабочим каталогом с этого момента будет корневая папка superlists

С этой точки и впредь корневая папка *superlists* будет нашим рабочим каталогом.

(Для простоты в моих распечатках команд я буду всегда показывать ее как */.../ superlists/*, хотя, вероятно, на самом деле она будет чем-то вроде */home/kind-reader-username/my-python-projects/superlists/*.)

Всегда, когда я показываю команду для ввода, подразумевается, что мы находимся в этом каталоге. Аналогично, если я упомяну путь к файлу, то он будет относительно этого корневого каталога. Так, *superlists/settings.py* означает файл *settings.py* внутри папки *superlists* второго уровня.

Ясно как день? Если есть сомнения, отыщите файл *manage.py*; вы должны быть в том же каталоге, что и *manage.py*.

Теперь посмотрим, какие файлы мы хотим фиксировать:

```
$ ls
db.sqlite3 manage.py superlists functional_tests.py
```

db.sqlite3 – это файл базы данных. Он не нужен в управлении версиями. Ранее мы также встречали *geckodriver.log*, журнальный файл от Selenium, изменения которого мы тоже не хотим отслеживать. Оба эти файла мы добавим в специальный файл под названием *.gitignore*, который «говорит» Git, что именно следует проигнорировать:

```
$ echo "db.sqlite3" >> .gitignore
$ echo "geckodriver.log" >> .gitignore
```

Затем мы можем добавить остальную часть содержимого текущей папки, «.»:

```
$ git add .
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: .gitignore
        new file: functional_tests.py
        new file: manage.py
        new file: superlists/__init__.py
        new file: superlists/__pycache__/__init__.cpython-36.pyc
        new file: superlists/__pycache__/settings.cpython-36.pyc
        new file: superlists/__pycache__/urls.cpython-36.pyc
        new file: superlists/__pycache__/wsgi.cpython-36.pyc
        new file: superlists/settings.py
        new file: superlists/urls.py
        new file: superlists/wsgi.py
```

Ух ты! У нас там целая куча файлов *.pyc*; бессмысленно их фиксировать. Удалим их из Git и тоже добавим в *.gitignore*:

```
$ git rm -r --cached superlists/__pycache__
rm 'superlists/__pycache__/__init__.cpython-36.pyc'
rm 'superlists/__pycache__/settings.cpython-36.pyc'
rm 'superlists/__pycache__/urls.cpython-36.pyc'
rm 'superlists/__pycache__/wsgi.cpython-36.pyc'
$ echo "__pycache__" >> .gitignore
$ echo "*.pyc" >> .gitignore
```

Теперь посмотрим, где мы находимся… (в дальнейшем вы увидите, что я много использую `git status` – да так много, что часто применяю псевдо-

ним `git st` ..., хотя я не скажу, как это делается; оставлю вам возможность обнаружить секреты псевдонимов Git самостоятельно!):

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file: .gitignore
        new file: functional_tests.py
        new file: manage.py
        new file: superlists/__init__.py
        new file: superlists/settings.py
        new file: superlists/urls.py
        new file: superlists/wsgi.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified: .gitignore
```

Выглядит неплохо, мы готовы сделать нашу первую фиксацию!

```
$ git add .gitignore
$ git commit
```

Когда вы наберете команду `git commit`, она раскроет окно редактора, чтобы вы написали в нем свое сообщение о фиксации. Мое выглядело как на рис. 1.3[4].

Если вы хотите оседлать Git по-настоящему, то пора узнать, как отправить вашу работу в облачную службу управления версиями, такую как GitHub или BitBucket. Они будут полезны, если вы намерены работать, следуя по тексту этой книги, на других ПК. Оставляю на вас обязанность узнать, как они работают; они располагают превосходной документацией. Как вариант, можно подождать до главы 9, когда мы будем использовать одну из них для развертывания.

[4] Редактор *vi* появился, и вы не представляете, что надо делать? Или же вы видите сообщение об идентичности регистрационной записи и `git config --global user.username`? Вернитесь и еще раз прочтите главу «*Предпосылки и предположения*»; там есть немного коротких инструкций.

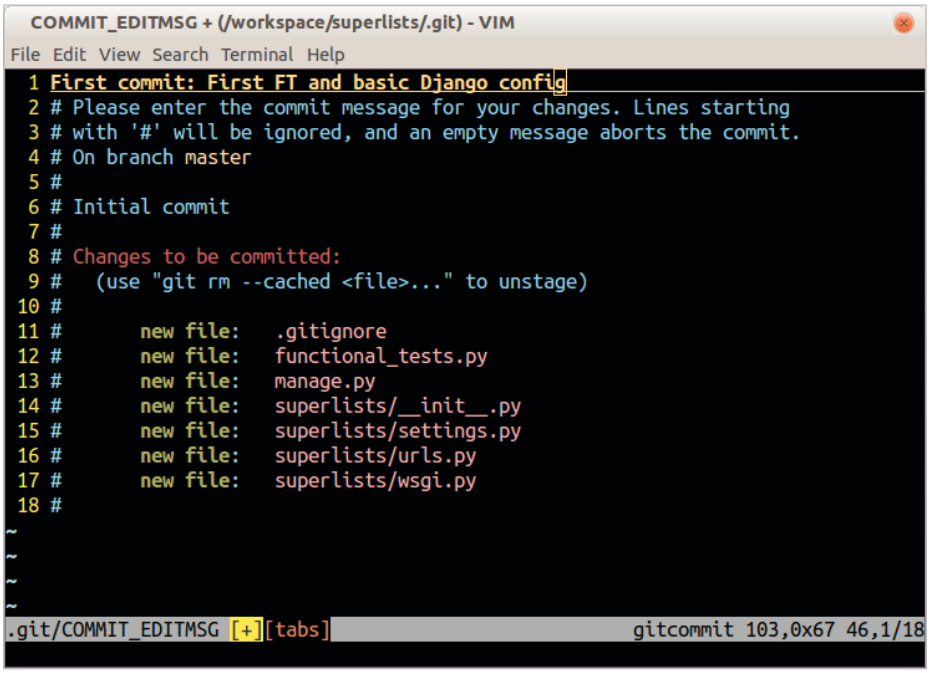


Рис. 1.3. Первая фиксация в Git

Вот и все, что касается лекции по системам управления версиями (VCS). Поздравляю! Вы написали функциональный тест при помощи Selenium, и у вас установлена платформа Django, которая работает самоочевидным, заверенным Билли способом, в стиле TDD и с первоочередным тестированием. Вы можете заслуженно похлопать себя по спине, перед тем как перейти дальше к главе 2.

Глава 2

Расширение функционального теста при помощи модуля unittest

Давайте адаптируем наш тест, который в настоящее время проверяет стандартную страницу Django «it worked» (сработало), и вместо нее проверим кое-какие вещи, которые мы хотим увидеть на реальной главной странице нашего сайта.

Пора раскрыть секрет по поводу того, какое веб-приложение мы создаем: сайт списков неотложных дел! При этом мы во многом следуем моде: несколько лет назад все учебные материалы по веб касались создания блога. Тогда это были форумы и опросы; сегодня – списки неотложных дел (to-do list).

Вся причина в том, что список неотложных дел – действительно удачный пример. По своей сути он очень прост – просто список текстовых строк – и поэтому «минимально дееспособное» приложение для обработки списка очень легко доводится до рабочего состояния. Помимо этого, его можно расширить – различными моделями долговременного хранения данных, добавлением сроков исполнения, напоминаний, обменом с другими пользователями и улучшением клиентского пользовательского интерфейса (UI). Кроме того, нет никаких причин ограничиваться только списками неотложных дел; это может быть любой вид списков. Но главное – это мне позволит продемонстрировать все основные аспекты веб-программирования и то, как применять к ним методологию TDD.

Использование функционального теста для определения минимального дееспособного приложения

Тесты, которые используют Selenium, позволяют управлять реальным веб-браузером, тем самым они дают нам возможность увидеть, как при-

ложение *функционирует* с точки зрения пользователя. Вот почему они называются *функциональными тестами*.

Это означает, что ФТ может выступать для вашего приложения своего рода спецификацией. Он помогает очертить контуры того, что вы, по-видимому, называете *историей пользователя*, и следит за тем, как пользователь мог бы работать с конкретным элементом и как приложение должно на них отвечать.

Терминология: функциональный тест == приемочный тест == сквозной тест

То, что я называю функциональными тестами, некоторые специалисты предпочитают именовать приемочными либо сквозными тестами или испытаниями. Их суть заключается в том, что эти виды тестов смотрят на характер функционирования всего приложения с внешней стороны. Есть и другой термин – тест черного ящика, потому что он ничего не знает о внутреннем устройстве тестируемой системы.

Функциональные тесты должны иметь удобочитаемую историю, которую можно прослеживать. Мы определяем ее в явной форме, используя комментарии, которые сопровождают программный код теста. При создании нового ФТ мы сначала пишем комментарии, чтобы зафиксировать ключевые моменты истории пользователя. Поскольку они легко читаемы, вы даже можете обмениваться ими с непрограммистами, чтобы обсудить технические требования и функциональные особенности вашего приложения.

TDD и методологии гибкой разработки программного обеспечения часто сочетаются, и одной из наиболее обсуждаемых является тема минимально дееспособного приложения. Каким будет самое простое возможное приложение, которое по-прежнему останется полезным? Предлагаю начать с создания такого приложения, чтобы как можно быстрее прощупать почву.

Минимальный дееспособный список неотложных дел, в общем-то, нуждается только в том, чтобы обеспечить пользователю возможность вводить некоторые элементы списка и запоминать их для следующего посещения.

Откройте *functional_tests.py* и напишите историю, что-то вроде этой:

functional_tests.py

```
from selenium import webdriver

browser = webdriver.Firefox()

# Эдит слышала про крутое новое онлайн-приложение со списком
```

```
# неотложных дел. Она решает оценить его домашнюю страницу
browser.get('http://localhost:8000')

# Она видит, что заголовок и шапка страницы говорят о списках
# неотложных дел
assert 'To-Do' in browser.title

# Ей сразу же предлагается ввести элемент списка

# Она набирает в текстовом поле "Купить павлиньи перья" (ее хобби –
# вязание рыболовных мушек)

# Когда она нажимает enter, страница обновляется, и теперь страница
# содержит "1: Купить павлиньи перья" в качестве элемента списка

# Текстовое поле по-прежнему приглашает ее добавить еще один элемент.
# Она вводит "Сделать мушку из павлиньих перьев"
# (Эдит очень методична)

# Страница снова обновляется, и теперь показывает оба элемента ее списка

# Эдит интересно, запомнит ли сайт ее список. Далее она видит, что
# сайт сгенерировал для нее уникальный URL-адрес – об этом
# выводится небольшой текст с объяснениями.

# Она посещает этот URL-адрес – ее список по-прежнему там.

# Удовлетворенная, она снова ложится спать
browser.quit()
```

Вы заметите, что, кроме детального описания теста в комментариях, я обновил утверждение `assert` для поиска слова «To-Do» вместо слова «Django». Это означает, что теперь тест ожидаемо не сработает. Попробуем его выполнить.

Сначала запустите сервер:

```
$ python manage.py runserver
```

Затем в другой оболочке запустите тесты:

```
$ python functional_tests.py
Traceback (most recent call last):
  File "functional_tests.py", line 10, in <module>
    assert 'To-Do' in browser.title
AssertionError
```

У нас свой термин для комментариев...

Когда я только начал работу в Resolver, я имел обыкновение добродетельно приправлять свой код хорошими описательными комментариями. Коллеги сказали: «Гарри, у нас есть свой термин для комментариев. Мы называем их брехней». Я был потрясен! Ведь я же со школьной скамьи знал, что комментарии являются хорошей практикой!

Они преувеличивали для пущего словца. Безусловно, есть место для комментариев, которые добавляют контекст и намерение. Но коллеги имели в виду, что бессмысленно писать комментарий, который просто повторяет то, что вы делаете в коде:

```
# увеличить wibble на 1
wibble += 1
```

Мало того, что это бессмысленно, существует опасность, что вы забудете их обновить при обновлении исходного кода и в итоге они станут вводить в заблуждение. В идеале нужно стремиться делать код настолько читаемым, использовать настолько хорошие имена переменных и функций и структурировать его так хорошо, чтобы вам больше не нужны были никакие комментарии для объяснения того, что делает программный код. Оставить лишь несколько тут и там для объяснения *причины*.

Есть и другие места, где комментарии очень полезны. Мы увидим, что Django часто применяет их в файлах, которые генерирует для нас, чтобы мы использовали их в качестве подсказок по поводу полезных элементов ее API. И, конечно, мы используем комментарии, чтобы объяснить историю пользователя в своих функциональных тестах, составляя из теста связную историю. Это даст гарантию, что мы всегда будем тестировать с точки зрения пользователя.

В этой области есть много других забавных вещей – например, *Разработка на основе поведения* (см. приложение E) и предметно-ориентированные языки для тестирования (DSL), но это темы других книг.

Это то, что мы называем *ожидаемой неполадкой* (failure), что в сущности представляет собой хорошую новость – не столь хорошую, как тест, который проходит, но по крайней мере он не сработал по правильной причине; у нас может быть некоторая уверенность, что мы написали тест правильно.

Модуль unittest стандартной библиотеки Python

Есть пара досадных мелочей, с которыми мы должны разобраться. Прежде всего, сообщение «AssertionError» не очень полезное – было бы неплохо, если бы тест сообщал нам о том, что именно он нашел в качестве заголовка браузера. Кроме того, он оставил окно Firefox где попало на рабочем столе – было бы неплохо, если бы он автоматически его убирал.

Один из вариантов – использовать второй параметр ключевого слова `assert`, что-то вроде:

```
assert 'To-Do' in browser.title, "Browser title was " + browser.title
```

И мы могли бы также применить блок `try/finally`, чтобы убрать старое окно Firefox. Однако эти виды проблем встречаются в тестировании довольно часто, и в модуле `unittest` стандартной библиотеки у нас имеется несколько готовых решений. Давайте ими воспользуемся! В файле *functional_tests.py*:

functional_tests.py

```
from selenium import webdriver
import unittest

class NewVisitorTest(unittest.TestCase):  ❶
    '''тест нового посетителя'''

    def setUp(self):  ❸
        '''установка'''
        self.browser = webdriver.Firefox()

    def tearDown(self):  ❸
        '''демонтаж'''
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):  ❷
        '''тест: можно начать список и получить его позже'''
        # Эдит слышала про крутое новое онлайн-приложение со
        # списком неотложных дел. Она решает оценить его
        # домашнюю страницу
        self.browser.get('http://localhost:8000')

        # Она видит, что заголовок и шапка страницы говорят о
        # списках неотложных дел
        self.assertIn('To-Do', self.browser.title)  ❹
        self.fail('Закончить тест!')  ❺

        # Ей сразу же предлагается ввести элемент списка
        [...остальные комментарии, как и прежде]

if __name__ == '__main__':  ❻
    unittest.main(warnings='ignore')  ❼
```

Здесь вы, вероятно, заметите несколько вещей:

❶ Тесты организованы в классы, которые наследуют от `unittest.TestCase`.

❷ Главная часть теста находится в методе под названием `test_can_start_a_list_and_retrieve_it_later`. Любой метод, имя которого начинается с `test`, является методом тестирования и будет выполнен исполнителем тестов. В одном классе может быть более одного метода `test_`. Неплохо также для методов тестирования иметь хорошо говорящие имена.

❸ `setUp` и `tearDown` – особые методы, которые выполняются перед и после каждого теста. Я их использую для запуска и остановки браузера – обратите внимание, что они немного походят на `try/except` в том, что `tearDown` будет выполняться, даже если во время самого теста произойдет ошибка[1]. Больше никаких окон Firefox, разбросанных где попало!

❹ Для создания тестовых утверждений мы используем `self.assertIn` вместо просто `assert`. Модуль `unittest` предоставляет много подобных вспомогательных функций для создания тестовых утверждений – например, `assertEqual`, `assertTrue`, `assertFalse` и т. д. Подробности вы можете узнать в документации по `unittest`[2].

❺ `self.fail` никогда не срабатывает и всегда генерирует переданное сообщение об ошибке. Я использую его в качестве напоминания об окончании теста.

❻ В конце у нас оператор `if __name__ == '__main__'` (если вы с ним раньше не встречались, то знайте: так сценарий Python проверяет, был ли он выполнен из командной строки, а не просто импортирован другим сценарием). Мы вызываем `unittest.main()`, который запускает исполнителя тестов `unittest`, а он в свою очередь автоматически найдет в файле тестовые классы и методы и их выполнит.

❼ `warnings='ignore'` подавляет излишние предупреждающие сообщения `Resourcewarning`, которые выдавались во время написания. Возможно, к тому моменту, когда вы будете это читать, они уже не будут появляться; не стесняйтесь, попробуйте удалить эту инструкцию!

Попробуем выполнить тест!

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
```

[1] Единственное исключение: когда у вас исключение внутри `setUp`, `tearDown` не работает.

[2] См. http://docs.python.org/3/library/unittest.html.

```
  File "functional_tests.py", line 18, in
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
AssertionError: 'To-Do' not found in 'Welcome to Django'

 ---------------------------------------------------------------------
Ran 1 test in 1.747s
FAILED (failures=1)
```

Теперь немного лучше, не правда ли? Окно Firefox убрано, и мы получили отформатированный отчет о том, сколько тестов было запущено и сколько из них не сработало, а `assertIn` выдало полезное сообщение об ошибке с информацией об отладке. Шикарно!

Во время чтения документации Django по тестированию вы, возможно, видели что-то похожее под названием `LiveServerTestCase` и интересуетесь, следует ли нам теперь его использовать. Вам 5 баллов за то, что прочли дружелюбное руководство! Пока `LiveServerTestCase` будет немного сложноват, но обещаю, что воспользуюсь им в одной из следующих глав.

Фиксация

Сейчас самое время выполнить фиксацию изменений; точнее, приятное автономное изменение. Мы развернули функциональный тест, включив комментарии, которые описывают поставленную задачу – минимальный дееспособный список неотложных дел. Мы также переписали этот тест для использования Python'овского модуля `unittest` и его разнообразных вспомогательных функций тестирования.

Выполните команду `git status` – она подтвердит, что единственный изменившийся файл – *functional_tests.py*. Затем выполните команду `git diff`, которая покажет разницу (дельту) между последней фиксацией и тем, что в настоящее время находится на диске. Она должна сообщить, что *functional_tests.py* достаточно существенно изменился:

```
$ git diff
diff --git a/functional_tests.py b/functional_tests.py
index d333591..b0f22dc 100644
--- a/functional_tests.py
+++ b/functional_tests.py
@@ -1,6 +1,45 @@
from selenium import webdriver
```

```
+import unittest

-browser = webdriver.Firefox()
-browser.get('http://localhost:8000')
+class NewVisitorTest(unittest.TestCase):

-assert 'Django' in browser.title
+    def setUp(self):
+        self.browser = webdriver.Firefox()
+
+    def tearDown(self):
+        self.browser.quit()
[...]
```

Теперь выполним:

```
$ git commit -a
```

`-a` означает «автоматически добавить любые изменения в отслеживаемых файлах» (то есть в любых файлах, которые мы фиксировали прежде). Эта опция не добавит совершенно новые файлы (вам придется самостоятельно добавлять их в явном виде с помощью команды `git`), но часто, как в данном случае, нет никаких новых файлов, так что это будет полезное сокращение.

Когда раскроется редактор, добавьте описательное сообщение о фиксации, например: «Расширен первый ФТ в комментариях, теперь использует unittest»[3].

Теперь мы находимся в превосходном положении, чтобы начать писать реальный код для приложения со списками. Продолжайте читать!

Полезные понятия TDD

История пользователя

Описание того, как приложение будет работать с точки зрения пользователя. Используется для структурирования функционального теста.

Ожидаемая неполадка

Когда тест не срабатывает ожидаемым для нас образом.

[3] См. http://pr0git.blogspot.ru/2015/02/git_4.html по поводу кодировки и комментариев на русском языке в Git – *Прим. перев.*

Глава 3

Тестирование простой домашней страницы при помощи модульных тестов

В предыдущей главе мы остановились на неработающем функциональном тесте, который сообщает, что ему требуется домашняя страница сайта с заголовком «To-Do». Пора приступить к работе над нашим приложением.

Внимание: скоро все станет по-настоящему

Первые две главы я намеренно сделал приятными и легкими. С этого момента мы входим в более содержательное программирование. Вот прогноз: в какой-то момент дела могут пойти не так, как надо. Вы увидите результаты, которые будут отличаться от того, что, по моим словам, вы должны увидеть. Ничего страшного – это будет подлинный опыт обучения, который формирует характер™.

Одна из причин может заключаться в том, что я дал какие-то неоднозначные объяснения и вы сделали не то, что я имел в виду. Вернитесь назад и поразмыслите о том, чего в этом месте в книге мы пытаемся достигнуть. Какой файл мы редактируем, что мы хотим, чтобы пользователь был в состоянии сделать, что мы тестируем и почему? Может случиться так, что вы отредактировали неправильный файл или функцию, или запускаете неправильные тесты. Я считаю, что в такие моменты с остановками и обдумыванием вы узнаете о TDD больше, чем когда следуете инструкциям и беспроблемно копипастите программный код.

Или же это может оказаться реальным дефектом. Будьте стойки, внимательно прочтите сообщение об ошибке (см. *мою ремарку по поводу чтения отчетов об обратной трассировке чуть позже в данной главе*), и вы докопаетесь до сути. Возможно, это окажется недостающая запятая или замыкающая косая черта, или отсутствующая «s» в одном из методов поиска Selenium. Но, как хорошо сказал Зед Шоу, такого рода отладка, без всяких сомнений, также является жизненно важной частью обучения, поэтому непременно с ней справьтесь!

Вы всегда можете написать мне на электронную почту (или попробовать связаться в Группе Google), если вы действительно попали в тупик. Успехов в отладке!

Первое приложение Django и первый модульный тест

Django способствует тому, чтобы вы структурировали свой программный код в *приложения*. Главная идея заключается в том, что один проект может иметь много приложений, вы можете использовать сторонние приложения, разработанные другими людьми, и даже снова использовать одно из собственных приложений из другого проекта… Хотя, надо признать, мне никогда не доводилось делать это самому! И все же приложения являются хорошим способом держать программный код в организованном порядке.

Запустим приложение для обработки списков неотложных дел:

```
$ python manage.py startapp lists
```

Эта команда создаст папку в *superlists/lists* рядом с *superlists/superlists* и в ней – несколько файлов-заготовок для таких вещей, как модели, представления и тесты, которые имеют для нас непосредственный интерес:

```
superlists/
├── db.sqlite3
├── functional_tests.py
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── __pycache__
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Модульные тесты и их отличия от функциональных тестов

Как и со многими метками, которые мы назначаем вещам, граница между модульными тестами и функциональными тестами может время от времени становиться немного расплывчатой. Основное различие, однако,

заключается в том, что функциональные тесты проверяют приложение с внешней стороны – с точки зрения пользователя. Модульные тесты проверяют приложение изнутри – с точки зрения программиста.

Подход TDD, которому я следую, хочет, чтобы приложение было охвачено обоими типами тестов. Поток операций будет примерно таким:

1. Мы начинаем с написания *функционального теста*, описывающего новую функциональность с точки зрения пользователя.
2. Когда у нас есть функциональный тест, который не срабатывает, мы начинаем думать о том, как написать код, который может заставить его пройти успешно (или по крайней мере заставить перешагнуть через текущую неполадку). Мы теперь используем один или несколько *модульных тестов*, чтобы выработать поведение кода, которого мы хотим добиться. Смысл в том, что каждая строка производственного программного кода, которую мы пишем, должна быть протестирована по крайней мере одним из наших модульных тестов.
3. Когда у нас есть неработающий модульный тест, мы пишем минимально возможный объем *прикладного кода* – ровно столько, чтобы модульный тест прошел успешно. Мы можем несколько раз итеративно переключаться между шагами 2 и 3, пока не убедимся, что функциональный тест продвинулся чуть дальше.
4. Теперь мы можем выполнить функциональные тесты повторно и убедиться, что они проходят или продвинулись немного дальше. Этот этап может нас побудить написать немного новых модульных тестов и нового исходного кода и т. д.

Вы видите, что на всем пути функциональные тесты влияют на характер осуществляемой нами разработки с верхнего уровня, тогда как модульный тест руководит тем, что мы делаем на низком уровне.

Не кажется ли это немного избыточным? Иногда это может ощущаться именно так, но функциональные и модульные тесты действительно имеют совсем разные задачи, и они, как правило, будут в конечном счете выглядеть очень по-разному.

Функциональные тесты должны вам помочь построить приложение с правильной функциональностью и гарантировать, что вы никогда не повредите его случайным образом. Модульные тесты помогут писать код, который будет чистым и свободным от дефектов.

Пока достаточно теории, предлагаю взглянуть, как это смотрится на практике.

Модульное тестирование в Django

Давайте посмотрим, как написать модульный тест для нашего представления (view) домашней страницы. Откройте новый файл в *lists/tests.py*, и вы увидите примерно следующее:

lists/tests.py

```
from django.test import TestCase

# Создайте здесь свои тесты.
```

Платформа Django с готовностью предположила, чтобы мы использовали предоставляемую ею специальную версию класса для работы с тестовыми случаями `TestCase`. Этот класс является расширенной версией стандартного `unittest.TestCase` с некоторыми дополнительными особенностями, характерными для Django, которые мы обнаружим в нескольких следующих главах.

Вы уже знаете, что цикл TDD предусматривает начало работы с теста, который не срабатывает, затем написание программного кода, который заставляет его выполняться успешно. Так вот, прежде чем мы сможем добраться до этой точки, мы хотим знать, что модульный тест, который мы пишем, будет определенно выполнен нашим автоматизированным исполнителем тестов, каким бы он ни был. В случае с *functional_tests.py* мы выполняем его непосредственно, но этот файл, созданный Django, немного смахивает на волшебство. Поэтому, чтобы только убедиться, предлагаю создать намеренно глупый, не срабатывающий тест:

lists/tests.py

```
from django.test import TestCase

class SmokeTest(TestCase):
    '''тест на токсичность'''

    def test_bad_maths(self):
        '''тест: неправильные математические расчеты'''
        self.assertEqual(1 + 1, 3)
```

Теперь обратимся к этому таинственному исполнителю тестов Django. Как обычно, это команда *manage.py*:

```
$ python manage.py test
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_bad_maths (lists.tests.SmokeTest)
```

```
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests.py", line 6, in test_bad_maths
    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3

 ----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

Превосходно! Похоже, механизм работает. Это отличная причина для фиксации:

```
$ git status # должна вам показать, что lists/ не отслеживается
$ git add lists
$ git diff --staged # покажет вам разницу/дельту, которую вы будете комитить
$ git commit -m "Добавлено приложение для списков с намеренно неработающим
модульным тестом"
```

Как вы наверняка угадали, флаг `-m` позволяет передать сообщение о фиксации в командной строке, поэтому вам не нужно использовать редактор. Выбор способа, которым вы хотели бы использовать оболочку Git, зависит от вас, я просто покажу вам основные, которые мне встречались. Основное правило: *прежде чем что-то фиксировать, всегда старайтесь проанализировать то, что вы собираетесь фиксировать*.

MVC в Django, URL-адреса и функции представления

Программная инфраструктура Django во многом структурирована согласно классическому шаблону проектирования «Модель-Представление-Контроллер» (MVC). Именно *во многом*. Она определенно имеет модели, но ее представления больше походят на контроллер, и именно шаблоны визуализации на самом деле реализуют представления, но общая идея именно та же самая. Если интересно, вы можете взглянуть на тонкости обсуждения в Django FAQ[1].

Независимо от любого из этих подходов основная работа Django, как и у любого другого веб-сервера, состоит в решении, что делать, когда пользователь запрашивает определенный URL-адрес на нашем сайте. Поток операций Django проходит примерно так:

[1] См. https://docs.djangoproject.com/en/1.11/faq/general/.

1. На определенный URL-адрес приходит HTTP-*запрос*.
2. Django использует некие правила, чтобы решить, какая функция *представления* должна обработать запрос (это называется *преобразованием* URL).
3. Функция представления обрабатывает запрос и возвращает HTTP-*отклик*.

Таким образом, мы хотим протестировать две вещи:

- Можем ли мы преобразовать URL для корня сайта («/») в конкретную функцию представления, которую мы создали?
- Можем ли мы заставить эту функцию представления вернуть некий HTML, который заставит функциональный тест пройти успешно?

Начнем с первого. Откройте *lists/tests.py* и измените наш глупый тест на что-то вроде этого:

lists/tests.py

```
from django.urls import resolve
from django.test import TestCase
from lists.views import home_page  ❷

class HomePageTest(TestCase):
    '''тест домашней страницы'''

    def test_root_url_resolves_to_home_page_view(self):
        '''тест: корневой url преобразуется в представление
           домашней страницы'''
        found = resolve('/')  ❶
        self.assertEqual(found.func, home_page)  ❶
```

Что здесь произошло?

❶ `resolve` – это функция, которую Django использует внутренне для преобразования URL-адреса и нахождения функций представления, в соответствие которым они должны быть поставлены. Мы проверяем, чтобы `resolve`, когда ее вызывают с «/», то есть корнем сайта, нашла функцию под названием `home_page`.

❷ Что это за функция? Это функция представления, которую мы собираемся написать далее, и она вернет фактический HTML, который нам нужен. Из оператора `import` вы видите, что мы планируем сохранить ее в *lists/views.py*.

Итак, что, по вашему мнению, произойдет, когда мы выполним тесты?

```
$ python manage.py test
ImportError: невозможно импортировать имя 'home_page'
```

Очень предсказуемая и неинтересная ошибка: мы попытались импортировать то, чего еще даже не написали. Тем не менее это хорошие новости – для TDD исключение, которое было спрогнозированным, считается ожидаемой неполадкой. Поскольку у нас неработающий функциональный тест и неработающий модульный тест, мы получаем всемерное благословление от Билли-тестировщика программировать на всю катушку.

Наконец-то! Мы на самом деле напишем прикладной код!

Захватывающе, не правда ли? Берегитесь, TDD означает, что длительные периоды предвкушения разряжаются крайне постепенно и крошечными приращениями. Особенно если учесть, что мы учимся и только-только начинаем, то мы позволяем себе изменять (или добавлять) всего по одной строке кода за один раз – и каждый раз мы делаем лишь минимальное изменение, которое требуется для того, чтобы решить текущую неполадку теста.

Здесь я намеренно перегибаю палку, но какова же наша текущая неполадка теста? Мы не можем импортировать `home_page` из `lists.views`? Хорошо, исправим это, и только это. В *lists/views.py*:

lists/views.py

```
from django.shortcuts import render

# Создайте здесь представления свои.
home_page = None
```

«*Да вы шутите?!*» – скажете вы мне.

Я это слышу от вас, потому что именно так говорил я сам (с чувством), когда мои коллеги впервые продемонстрировали мне TDD. Так вот, потерпите. О том, заведет ли все это слишком далеко или нет, мы поговорим чуть позже. А пока позвольте себе идти дальше, пусть даже с некоторым раздражением, и убедитесь, что наши тесты помогают писать правильный код маленькими порциями, шаг за шагом.

Выполняем тесты еще раз:

```
$ python manage.py test
Creating test database for alias 'default'...
E
======================================================================
ERROR: test_root_url_resolves_to_home_page_view (lists.tests.HomePageTest)
```

```
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests.py", line 8, in
test_root_url_resolves_to_home_page_view
    found = resolve('/')
  File ".../django/urls/base.py", line 27, in resolve
    return get_resolver(urlconf).resolve(path)
  File ".../django/urls/resolvers.py", line 392, in resolve
    raise Resolver404({‘tried’: tried, 'path': new_path})
django.urls.exceptions.Resolver404: {'tried': [[<RegexURLResolver
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}

 ----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (errors=1)
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

Файл urls.py

Наши тесты сообщают, что нам требуется отображение URL-адреса на обработчик. Django использует файл под названием *urls.py* для отображения URL-адресов на функции представления. В папке *superlists/superlists* имеется основной файл *urls.py* для всего сайта. Пойдем и посмотрим:

superlists/urls.py

```
"""superlists URL Configuration

Список `urlpatterns` направляет URL-адреса к представлениям.
За дополнительной информацией обратитесь на:
    https://docs.djangoproject.com/en/dev/topics/http/urls/
Примеры:
Представления на основе функций
    1. Добавить импорт: from my_app import views
    2. Добавить URL в urlpatterns: url(r'^$', views.home, name='home')
Представления на основе классов
    1. Добавить импорт: from other_app.views import Home
    2. Добавить URL в urlpatterns: url(r'^$', Home.as_view(), name='home')
Включение еще одного URLconf
    1. Импортировать функцию include(): from django.conf.urls import url, include
    2. Добавить URL в urlpatterns: url(r'^blog/', include('blog.urls'))
"""
from django.conf.urls import url
```

```
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
]
```

Как обычно, много полезных комментариев и стандартных предложений от Django.

Запись `url` начинается регулярным выражением, которое устанавливает, к каким URL-адресам оно применяется, и далее говорит, куда оно должно отправить эти запросы – либо в функцию представления, которую вы импортировали, либо, возможно, в другой файл *urls.py* где-то в другом месте.

Первый пример содержит регулярное выражение `^$`, которое означает пустую строку. Может ли оно совпасть с корнем нашего сайта, который мы тестировали с «/»? Что произойдет, если эту запись включить?

Если вы никогда не сталкивались с регулярными выражениями, пока можете просто поверить мне на слово, но вам следует взять на заметку, что нужно все подробно о них разузнать.

Мы также избавимся от URL администратора, потому что пока мы не будем использовать сайт администратора Django.

superlists/urls.py

```
from django.conf.urls import url
from lists import views

urlpatterns = [
    url(r'^$', views.home_page, name='home'),
]
```

Командой `python manage.py test` выполните модульные тесты еще раз:

```
[...]
TypeError: представление должно быть вызываемым объектом или списком/кортежем
в случае include().
```

Какой-никакой, а прогресс! Мы больше не получаем ошибку 404.

Результат обратной трассировки выглядит запутанным, но сообщение в самом конце говорит о том, что происходит: модульные тесты фактически установили связь между URL-адресом `/` и `home_page=None` в файле *lists/views.py*, и теперь жалуются, что представление `home_page` не вызывается. И это дает нам оправдание для того, чтобы его изменить, поменяв `None` на фак-

тическую функцию. Каждое вносимое в код изменение выполняется под руководством тестов!

Чтение отчетов об обратной трассировке

Предлагаю ненадолго прерваться и поговорить о том, как читать обратные трассировки, поскольку в TDD нам приходится делать это часто. Вскоре вы научитесь их просматривать и подбирать соответствующие зацепки:

```
======================================================================
ERROR: test_root_url_resolves_to_home_page_view (lists.tests.HomePageTest)  ❷
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests.py", line 8, in
test_root_url_resolves_to_home_page_view
    found = resolve('/')  ❸
  File ".../django/urls/base.py", line 27, in resolve
    return get_resolver(urlconf).resolve(path)
  File ".../django/urls/resolvers.py", line 392, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
django.urls.exceptions.Resolver404: {'tried': [[<RegexURLResolver  ❶
<RegexURLPattern list> (admin:admin) ^admin/>]], 'path': ''}  ❶
 ----------------------------------------------------------------------
[...]
```

❶ Первое место, куда вы смотрите, – это обычно *сама ошибка*. Иногда это все, что нужно увидеть, и она позволит вам сразу идентифицировать проблему. Но иногда, как в данном случае, это не совсем самоочевидно.

❷ Следующее, что надо перепроверить, – это *какой тест не срабатывает*? Является ли он определенно тем, который мы ожидали, то есть который мы только что написали? В данном случае ответ «да».

❸ Затем мы ищем в тестовом коде место, которое выдало неполадку. Прослеживаем путь вниз от вершины результатов трассировки в поисках имени файла с тестами, чтобы установить, из какой тестовой функции и из какой строки кода поступает неполадка. В данном случае это строка, где мы вызываем функцию `resolve` для корневого («/») URL-адреса.

Обыкновенно существует и четвертый шаг, где мы идем далее вниз в поисках любого *нашего собственного прикладного кода*, который был связан с проблемой. В данном случае у нас весь код сгенерирован платформой Django, но позже в книге мы увидим много примеров этого четвертого шага.

Собрав все вместе, мы истолковываем результаты обратной трассировки как сообщение о том, что при попытке преобразовать «/» платформа Django подняла ошибку 404. Другими словами, Django не может найти отображение URL-адреса на обработчик для «/». Давайте ее выручим.

Назад в *lists/views.py*:

lists/views.py

```
from django.shortcuts import render

# Создайте ваши представления здесь.
def home_page():
    '''домашняя страница'''
    pass
```

И сейчас?

```
$ python manage.py test
Creating test database for alias 'default'...
.
 ----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

Ура! Наш самый первый модульный тест проходит! Этот тест настолько судьбоносный, что, думаю он, достоин фиксации:

```
$ git diff # покажет изменения в urls.py, tests.py и views.py
$ git commit -am "Первый модульный тест и преобразование url, фиктивное
представление"
```

Это была последняя команда `git commit`, которую я покажу вместе с флагами `a` и `m`. Она добавляет все изменения в отслеживаемые файлы и использует сообщение о фиксации из командной строки.

Команда `git commit -am` – это самая быстрая формула, но она предоставляет вам наименьший отзыв о том, что именно фиксируется, поэтому обязательно заранее выполните `git status` и `git diff` и убедитесь, что вам понятно, какие изменения будут сделаны.

Модульное тестирование представления

Переходим к написанию теста для нашего представления, который будет чем-то большим, чем пустой функцией, а именно, будет возвращать в браузер реальный отклик в виде HTML. Откройте *lists/tests.py* и добавьте новый *метод тестирования*. Я объясню каждый фрагмент:

lists/tests.py

```
from django.urls import resolve
from django.test import TestCase
from django.http import HttpRequest

from lists.views import home_page

class HomePageTest(TestCase):

    def test_root_url_resolves_to_home_page_view(self):
        '''тест: корневой url преобразуется в представление
           домашней страницы'''
        found = resolve('/')
        self.assertEqual(found.func, home_page)

    def test_home_page_returns_correct_html(self):
        '''тест: домашняя страница возвращает правильный html'''
        request = HttpRequest()  ❶
        response = home_page(request)  ❷
        html = response.content.decode('utf8')  ❸
        self.assertTrue(html.startswith('<html>'))  ❹
        self.assertIn('<title>To-Do lists</title>', html)  ❺
        self.assertTrue(html.endswith('</html>'))  ❹
```

Что в этом новом тесте происходит?

❶ Мы создаем объект `HttpRequest`, то есть то, что Django увидит, когда браузер пользователя запросит страницу.

❷ Мы передаем его представлению `home_page`, которое дает отклик. Вы не удивитесь, когда услышите, что этот объект является экземпляром класса под названием `HttpResponse`.

❸ Затем мы извлекаем содержимое `.content` отклика. Это необработанные байты, единицы и нули, которые будут отправлены по проводам в браузер пользователя. Мы вызываем `.decode()`, чтобы конвертировать их в символьную строку HTML, которая отправляется пользователю.

❹ Нам нужно, чтобы она начиналась с тега `<html>`, который закрывается в конце.

❺ И мы хотим разместить тег `<title>` со словами «to-do lists» где-нибудь в середине – потому что это то, что мы определили в нашем функциональном тесте.

Опять-таки, модульный тест находится под управлением функционального теста, но он также намного ближе к фактическому программному коду – сейчас мы думаем, как программисты.

Теперь выполним модульные тесты и посмотрим, что мы получаем:

```
TypeError: home_page() принимает 0 позиционных аргументов, но был предоставлен 1
```

Цикл «модульный-тест/программный-код»

Теперь мы можем начать осваивать цикл TDD «модульный-тест/программный-код»:

1. В терминале выполните модульные тесты и убедитесь, что они не срабатывают.
2. В редакторе внесите минимальное изменение в программный код, чтобы решить текущую неполадку теста.

И повторите!

Чем больше мы нервничаем по поводу приведения нашего кода в правильное состояние, тем меньше должно быть каждое изменение кода, которое мы вносим. Смысл в том, чтобы быть абсолютно уверенным, что каждый фрагмент кода обоснован тестом.

Такой процесс может показаться трудоемким, и сначала так и будет. Но как только вы войдете в ритм, вы обнаружите, что разрабатываете программный код быстро, даже если делаете микроскопические шаги. Так мы пишем весь наш производственный программный код на работе.

Посмотрим, насколько быстро мы сможем наладить этот цикл:

- Минимальное изменение кода:

lists/views.py

```
def home_page(request):
    '''домашняя страница'''
    pass
```

- Тесты:

```
html = response.content.decode('utf8')
AttributeError: у объекта'NoneType' нет атрибута 'content'
```

- Программный код – как и предсказывалось, мы используем `django.http.HttpResponse`:

lists/views.py

```
from django.http import HttpResponse

# Создайте здесь ваши представления.
def home_page(request):
    '''домашняя страница'''
    return HttpResponse()
```

- Снова тесты:

```
    self.assertTrue(html.startswith('<html>'))
AssertionError: Ложь не является истиной
```

- Снова программный код:

lists/views.py

```
def home_page(request):
    '''домашняя страница'''
    return HttpResponse('<html>')
```

- Тесты:

```
AssertionError: '<title>To-Do lists</title>' не найден в '<html>'
```

- Программный код:

lists/views.py

```
def home_page(request):
    '''домашняя страница'''
    return HttpResponse('<html><title>To-Do lists</title>')
```

- Тесты – почти у цели?

```
    self.assertTrue(html.endswith('</html>'))
AssertionError: Ложь не является истиной
```

- Ну давай, одно последнее усилие:

lists/views.py

```
def home_page(request):
    '''домашняя страница'''
    return HttpResponse('<html><title>To-Do lists</title></html>')
```

- Неужели?

```
$ python manage.py test
Creating test database for alias 'default'...
..
 ----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

Получилось! Теперь выполним функциональные тесты. Не забудьте снова завести сервер разработки, если он все еще не работает. Тут, похоже, наступает финальный этап гонки, несомненно, это он… Этого не может не быть!

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "functional_tests.py", line 19, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Закончить тест!')
AssertionError: Закончить тест!

 ----------------------------------------------------------------------
Ran 1 test in 1.609s

FAILED (failures=1)
```

Не сработало? Что? Или это просто наше собственное напоминание? И? Да! Получилось! У нас есть веб-страница!

Гм. Что ж, *я-то* полагал, что это захватывающий конец главы. Вы, возможно, по-прежнему немного озадачены, жаждете услышать обоснование для всех этих тестов. Не переживайте, все будет потом, но я надеюсь, что ближе к концу вы хотя бы почувствовали оттенок волнения.

Просто маленькая фиксация, чтобы успокоиться и поразмышлять над тем, что мы обсудили:

```
$ git diff # покажет новый тест в tests.py и представление в views.py
$ git commit -am "Базовое представление теперь возвращает минимальный HTML"
```

Это была настоящая глава! Почему бы не попробовать набрать `git log` – возможно, с использованием флага `--oneline`, чтобы получить напоминание о том, что же у нас тут получилось:

```
$ git log --oneline
a6e6cc9 Базовое представление теперь возвращает минимальный HTML
450c0f3 Первый модульный тест и преобразование url, фиктивное представление
ea2b037 Добавлено приложение для списков с намеренно неработающим модульным
тестом
[...]
```

Неплохо! Мы рассмотрели:

- Запуск приложения Django.
- Исполнителя модульных тестов Django.
- Разницу между ФТ и модульными тестами.
- Преобразование URL в Django и *urls.py*.
- Функции представления Django, объекты запроса и отклика.
- И возвращение элементарной HTML-разметки.

Полезные команды и понятия

Выполнение сервера разработки Django:

```
python manage.py runserver
```

Выполнение функциональных тестов:

```
python functional_tests.py
```

Выполнение модульных тестов:

```
python manage.py test
```

Цикл «модульный-тест/программный-код»:

1. Выполнить в терминале модульные тесты.
2. Внести в редакторе минимальное изменение в программный код.
3. Повторить!

Глава 4

И что же делать со всеми этими тестами (и рефакторизацией)?

Теперь, когда мы увидели азы методологии TDD в действии, пора остановиться и поговорить о том, почему мы это делаем.

Я представляю себе, как некоторые из вас, дорогие читатели, сдерживали кипящее разочарование – возможно, кто-то немного занимался модульным тестированием прежде, а кто-то просто торопится. Вы сдерживали себя, чтоб не задать такие вопросы, как:

- Разве все эти тесты не являются чрезмерными?
- Некоторые из них избыточны, не правда ли? Ну очевидно же дублирование части функциональных тестов модульными.
- Так чего вы добиваетесь этим импортом `django.core.urlresolvers` в своих модульных тестах? Не является ли это тестированием Django, то есть тестированием стороннего кода? Я считал, что это табу, разве нет?
- Эти модульные тесты, пожалуй, чересчур тривиальные – тестирование одной строки объявления и короткой функции, которая возвращает константу! Разве это не пустая трата времени? Может, оставить тесты для более сложных вещей?
- А как быть со всеми этими крошечными изменениями во время цикла «модульный-тест/программный-код»? Нельзя ли было просто перейти в конец? Я имею в виду, `home_page = None`!? Ведь это так?
- Только не говорите мне, что вы *на самом деле* разрабатываете такого рода программный код в реальных условиях.

Когда-то, будучи начинающим юнцом, я тоже задавался вопросами вроде этих. Но только потому, что они действительно хороши. По сути, я

по-прежнему все время задаю себе такие вопросы. Действительно ли все эти вещи имеют какое-то значение? Не является ли это чем-то вроде карго-культа?

Программировать – все равно что поднимать ведро с водой из колодца

По большому счету программировать нелегко. Как водится, мы умны и поэтому успешны. Методология TDD нужна для того, чтобы помогать нам, когда мы не так умны. Кент Бек (Kent Beck), который, в сущности, разработал TDD, использует метафору с подъемом ведра с водой из колодца на веревке: когда колодец не слишком глубокий и ведро не очень полное, это просто. И даже подъем полного ведра довольно легок поначалу. Но через некоторое время вы начинаете уставать. Методология TDD – это все равно что храповое колесо, которое позволяет вам сохранять темп, делать перерывы и убеждаться, что вы никогда не соскользнете назад. Благодаря этому вам не нужно все время быть умным.

Рис. 4.1. Тестируйте ВСЕ, что можно (Исходный источник иллюстрации: Элли Брош и гипербола с половиной – http://bit.ly/1iXxdYp)

Хорошо. Возможно, *в целом* вы готовы признать, что методология TDD – хорошая идеея. Но, может быть, вы по-прежнему думаете, что я перегибаю палку? Тестирую крошечную единицу программного кода и выполняю смехотворно большое количество маленьких шагов?

TDD – это *дисциплина*, а значит, это не то, что приходит естественным образом. Поскольку многие награды не достаются сразу, а приходят в длительной перспективе, от вас требуется заставлять себя это делать в данный момент. Это именно то, что образ Билли-тестировщика призван проиллюстрировать – вам нужно быть чуть-чуть зловредным.

О достоинствах примитивных тестов для примитивных функций

В краткосрочной перспективе написание тестов для простых функций и констант может показаться немного глупым.

Вполне возможно представить ситуацию, когда TDD «по большей части» выполняется, но при менее строгих правилах, где вы не тестируете модульно абсолютно все. Однако цель этой книги – продемонстрировать полную, строгую методологию TDD. Как и ката в боевых искусствах, идея состоит в том, чтобы изучить движения в управляемом контексте, когда противник отсутствует, чтобы приемы стали частью вашей мышечной памяти. Сейчас это кажется банальным, потому что мы начали с очень простого примера.

Проблема появляется, когда приложение становится сложным – вот когда вам действительно понадобятся тесты. И опасность состоит в том, что сложность имеет тенденцию подкрадываться исподтишка, постепенно. Вы можете не заметить, что это происходит, но довольно скоро окажетесь сваренной лягушкой.

Есть еще пара вещей, которые можно сказать в пользу крошечных, простых тестов для простых функций.

Во-первых, если это действительно примитивные тесты, то их написание не займет слишком много времени. Так что прекратите стонать и просто уже напишите их.

Во-вторых, всегда неплохо иметь заготовку (местозаполнитель, placeholder). Наличие там теста для простой функции означает преодоление гораздо меньшего психологического барьера, когда простая функция становится чуть-чуть сложнее – возможно, она прирастит оператор `if`. Затем, несколько недель спустя, прирастит цикл `for`. Не успеешь оглянуться, а она уже фабрика рекурсивных древовидных анализаторов на основе полиморфизма метаклассов. Но благодаря тому, что у нее были тесты с самого начала, добавление каждого нового теста ощущалось вполне естественным, и она хорошо протестирована. Альтернативная методика сопряжена с необходимостью принять решение, когда функция стала «достаточно сложной», что само по себе весьма субъективно. Но, что еще хуже, ввиду того, что заготовки нет, такое решение кажется слишком большим усилием, и вы каждый раз испытываете желание отодвинуть его принятие. И довольно скоро получаете суп из лягушки!

Вместо того чтобы пытаться придумывать какие-либо изменчивые субъективные правила о том, когда следует писать тесты, а когда можно обойтись без них, я предлагаю пока следовать дисциплине – как и в случае любой дисциплины, вам нужно, не торопясь, изучить правила, прежде чем их можно будет нарушать.

Теперь вернемся к нашей теме.

Использование Selenium для тестирования взаимодействий пользователя

На чем мы остановились в конце предыдущей главы? Давайте снова выполним тест и выясним:

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "functional_tests.py", line 19, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Закончить тест!')
AssertionError: Закончить тест!

 ----------------------------------------------------------------------
Ran 1 test in 1.609s

FAILED (failures=1)
```

Вы попробовали и получили ошибку с сообщением *Problem loading page* (Проблема при загрузке страницы) или *Unable to connect* (Не в состоянии установить соединение)? У меня то же самое. Это потому, что мы забыли сперва завести сервер разработки, применив `manage.py runserver`. Сделайте это, и вы получите сообщение о неполадке, которое нам нужно.

Одна из прелестей TDD состоит в том, что вы никогда не забудете, что делать дальше – просто повторно выполните свои тесты, и они скажут, с чем вам нужно работать.

«Закончить тест», – говорит TDD, поэтому давайте-ка этим и займемся! Откроем *functional_tests.py* и расширим наш ФТ:

functional_tests.py

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import time
import unittest

class NewVisitorTest(unittest.TestCase):

    def setUp(self):
```

```
        '''установка'''
        self.browser = webdriver.Firefox()

    def tearDown(self):
        '''демонтаж'''
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):
        '''тест: можно начать список и получить его позже'''
        # Эдит слышала про крутое новое онлайн-приложение со списком
        # неотложных дел. Она решает оценить его домашнюю страницу
        self.browser.get('http://localhost:8000')

        # Она видит, что заголовок и шапка страницы говорят о списках
        # неотложных дел
        self.assertIn('To-Do', self.browser.title)
        header_text = self.browser.find_element_by_tag_name('h1').text  ❶
        self.assertIn('To-Do', header_text)

        # Ей сразу же предлагается ввести элемент списка
        inputbox = self.browser.find_element_by_id('id_new_item')  ❶
        self.assertEqual(
            inputbox.get_attribute('placeholder'),
            'Enter a to-do item'
        )

        Она набирает в текстовом поле "Купить павлиньи перья"
        # (ее хобби – вязание рыболовных мушек)
        inputbox.send_keys('Купить павлиньи перья')  ❷

        # Когда она нажимает enter, страница обновляется, и теперь страница
        # содержит "1: Купить павлиньи перья" в качестве элемента списка
        inputbox.send_keys(Keys.ENTER)  ❸
        time.sleep(1)  ❹

        table = self.browser.find_element_by_id('id_list_table')
        rows = table.find_elements_by_tag_name('tr')  ❶
        self.assertTrue(
            any(row.text == '1: Купить павлиньи перья' for row in rows)
        )

        # Текствое поле по-прежнему приглашает ее добавить еще один элемент.
        # Она вводит "Сделать мушку из павлиньих перьев"
        # (Эдит очень методична)
```

```
        self.fail('Закончить тест!')

        # Страница снова обновляется и теперь показывает оба элемента
        # ее списка
        [...]
```

❶ Мы используем несколько методов, которые Selenium предоставляет для исследования веб-страниц: `find_element_by_tag_name`, `find_element_by_id` и `find_element`**`s`**`_by_tag_name` (обратите внимание на дополнительную `s`, то есть она вернет несколько элементов, а не один).

❷ Мы также используем `send_keys`, который является принятым в Selenium способом ввода данных в поля ввода `input`.

❸ Класс `Keys` (не забудьте его импортировать) позволяет нам отправлять специальные клавиши наподобие **Enter**[1].

❹ Когда мы нажмем **Enter**, страница обновится. Наличие `time.sleep` гарантирует, что браузер закончил загружать новую страницу, прежде чем мы сделаем какие-либо утверждения. Это называется явным ожиданием[2] (очень простым; мы его усовершенствуем в главе 6).

Не упустите разницу между функциями Selenium `find_element_...` и `find_elements_...`. Одна возвращает элемент и поднимает исключение, если она не может его найти, а другая возвращает список, который может быть пустым.

Кроме того, взгляните на функцию `any`. Это малоизвестная встроенная функция Python. Мне даже не нужно ее объяснять, не правда ли? Python – сплошное удовольствие.

Хотя, если вы – один из тех читателей, кто Python не знает, внутри этой функции находится *генераторное выражение*, которое похоже на списковое включение[3], но круче. Вам обязательно нужно о нем почитать. Если

[1] Кроме того, вы можете воспользоваться строкой «\n», но класс Keys также позволяет вам отправлять специальные клавиши, такие как **Ctrl**, поэтому посчитал, что стоит его показать.

[2] Функции ожидания (wait functions) дают возможность потоку блокировать собственное исполнение кода. Функции ожидания не возвращают значения до тех пор, пока не будут выполнены заданные критерии. – *Прим. перев.*

[3] Генератор (построитель) последовательности (англ. list comprehension, также «списковое включение») – это синтаксическая конструкция, результатом выполнения которой является последовательность (список, множество или словарь). Предназначена для создания последовательностей путем применения операций к существующим последовательностям, соответствует математической форме записи для построения множества и заменяет использование функций `map` и `filter`. – *Прим. перев.*

погуглить, то вы найдете самого Гвидо, который прекрасно его объясняет[4]. Когда вернетесь, удивлюсь, если скажете, что это не чистое удовольствие!

Давайте посмотрим, как поживает наш тест:

```
$ python functional_tests.py
[...]
selenium.common.exceptions.NoSuchElementException: Message: Не получается
локализовать элемент: h1
```

Если расшифровать, то тест говорит, что не может найти на странице элемент <h1>. Посмотрим, что можно сделать, чтобы его добавить в HTML-разметку домашней страницы.

Большие изменения в функциональном тесте сами по себе обычно являются достаточным основанием для фиксации. Мне не удалось сделать это в своем первом проекте, и позже я об этом пожалел, когда передумал и смешал это изменение с рядом других. Чем более атомарными являются ваши фиксации, тем лучше:

```
$ git diff # покажет изменения в functional_tests.py
$ git commit -am "ФТ теперь проверяет возможность ввести элемент списка"
```

Правило «Не тестировать константы» и шаблоны во спасение

Предлагаю взглянуть на наши модульные тесты, *lists/tests.py*. В настоящее время мы ищем конкретные строковые значения с HTML-разметкой, но это не особенно эффективный метод тестирования HTML. Одно из правил модульного тестирования гласит «Не тестировать константы», и тестирование HTML как текста во многом аналогично тестированию константы.

Другими словами, если у вас есть некий программный код, который говорит:

```
wibble = 3,
```

нет особого смысла в тесте, который говорит:

```
from myprogram import wibble
assert wibble == 3
```

Модульные тесты на самом деле занимаются проверкой логики, потока управления и конфигурации. Утверждения о том, какая именно последовательность символов имеется в нашей HTML-разметке, этого не дают.

Что более важно: выжимать сырые строки в Python – это на самом деле далеко не самый хороший способ работы с HTML. Существует намного бо-

[4] См. http://bit.ly/1iXxD18

лее хорошее решение, которое заключается в использовании шаблонов. Совершенно отдельно от чего-либо, если в файле, чье имя закачивается расширением *.html*, мы сможем держать HTML в одном месте, то получим более хорошую подсветку синтаксиса! В Python есть много систем шаблонизации, и у Django есть своя собственная, которая работает очень хорошо. Давайте ею воспользуемся.

Перестройка программного кода для использования шаблона

Теперь нам нужно заставить функцию представления вернуть точно тот же HTML, но с помощью другого процесса. Этот процесс – рефакторизация, или перестройка программного кода, то есть когда мы пытаемся улучшить программный код *без изменения его функциональности*.

Эта последняя часть определения действительно имеет очень важное значение. Попробовав перестроить код и добавить новую функциональность во время рефакторизации, вы с большой вероятностью столкнетесь с проблемой. Рефакторизация сама по себе фактически является отдельной дисциплиной, и она даже имеет свой справочник[5].

Первое правило рефакторизации состоит в том, что вы не можете ее выполнять без тестов. К счастью, мы занимаемся TDD, поэтому у нас большая фора. Предлагаю убедиться, что наши тесты проходят; они будут подтверждать, что наша рефакторизация не затрагивает поведение программного кода:

```
$ python manage.py test
[...]
OK
```

Здорово! Для начала возьмем строку с HTML-разметкой и поместим ее в отдельный файл. Создайте каталог по названием *lists/templates*, в котором будут содержаться шаблоны, а затем откройте файл в *lists/templates/home.html*, куда мы передадим наш HTML[6]:

lists/templates/home.html

```
<html>
    <title>To-Do lists</title>
</html>
```

[5] Рефакторизация (Refactoring), автор Мартин Фаулер (Martin Fowler), см. http://refactoring.com

[6] Некоторые предпочитают использовать еще одну подпапку, названную по имени приложения (то есть *lists/templates/lists*) и затем ссылаться на шаблон как *lists/home.html*. Это называется организацией пространства имен шаблонов (template namespacing). Я посчитал, что для такого небольшого проекта это будет чересчур сложным, но в более крупных проектах, возможно, оно того стоит. См. подробности в учебном руководстве по Django (http://bit.ly/1iXxWZL).

Ухты, выделение синтаксиса. Гораздо круче! Теперь изменим нашу функцию представления:

lists/views.py

```
from django.shortcuts import render

def home_page(request):
    '''домашняя страница'''
    return render(request, 'home.html')
```

Вместо создания собственного `HttpResponse` мы теперь используем функцию Django `render`. Она берет запрос в качестве своего первого параметра (причины мы обсудим позже) и название шаблона для вывода в виде HTML. Django будет автоматически искать папки с именем *templates* внутри всех каталогов вашего приложения. Затем создаст `HttpResponse` за вас, основываясь на содержимом шаблона.

Шаблоны являются очень мощным инструментом Django. Их основная сила – в подстановке переменных Python в текст HTML. Мы этот инструмент пока не используем, но обязательно обратимся к нему в следующих главах. Вот почему мы используем `render` и (позже) `render_to_string`, а не, скажем, ручное чтение файла с диска при помощи встроенной функции `open`..

Убедимся, что все работает, как задумано:

```
$ python manage.py test
[...]
======================================================================
ERROR: test_home_page_returns_correct_html (lists.tests.HomePageTest)  ❷
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests.py", line 17, in
test_home_page_returns_correct_html
    response = home_page(request)  ❸
  File "/.../superlists/lists/views.py", line 5, in home_page
    return render(request, 'home.html')  ❹
  File "/usr/local/lib/python3.6/dist-packages/django/shortcuts.py", line 48,
in render
    return HttpResponse(loader.render_to_string(*args, **kwargs),
  File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py",
line 170, in render_to_string
    t = get_template(template_name, dirs)
  File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py",
```

```
line 144, in get_template
    template, origin = find_template(template_name, dirs)
  File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py",
line 136, in find_template
    raise TemplateDoesNotExist(name)
django.template.base.TemplateDoesNotExist: home.html  ❶

 ----------------------------------------------------------------------
Ran 2 tests in 0.004s
```

Еще один шанс проанализировать обратную трассировку:

❶ Начинаем с ошибки: не удается найти шаблон.

❷ Перепроверяем, какой тест не работает. Конечно же, это наш тест представления HTML.

❸ Затем в тестах находим строку, которая вызвала неполадку: это происходит, когда мы вызываем функцию `home_page`.

❹ Наконец мы отыскиваем часть собственного прикладного кода, который вызвал неполадку: это когда мы пытаемся вызвать `render`.

Итак, почему Django не может найти шаблон? Ведь он именно там, где ему положено быть – в папке *lists/templates*.

Дело в том, что мы еще не зарегистрировали приложение `lists` в Django официально. К сожалению, недостаточно просто выполнить команду `startapp` и получить в своей папке с проектом то, что с очевидностью является приложением. Необходимо сказать Django, что вы действительно этого хотите, и также добавить его в *settings.py*. На всякий случай. Откройте его и отыщите переменную `INSTALLED_APPS`, в которую мы добавим приложение `lists`:

superlists/settings.py

```
# Определение приложений

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
]
```

Вы видите, что по умолчанию там уже имеется много приложений. Нам просто нужно добавить свое – *lists* – в конец списка приложений. Не

забывайте про замыкающую запятую – она не требуется, но в определенный момент вы будете сильно раздражены, когда забудете про нее и Python свяжет два строковых значения, расположенных на разных строках…

Теперь можно снова попытаться выполнить тесты:

```
$ python manage.py test
    [...]
    self.assertTrue(html.endswith('</html>'))
AssertionError: False is not true
```

Проклятье, пока не получается.

Если ваш текстовый редактор настаивает на добавлении символов новой строки в конце файлов, то вы даже не увидите эту ошибку. Если это так, можете спокойно проигнорировать следующий фрагмент и перейти прямо туда, где можно увидеть **OK** в распечатке кода.

Однако тест действительно продвинулся дальше! Похоже, ему удалось найти наш шаблон, но последнее из трех утверждений не сработало. Очевидно, что-то не так в конце вывода. Мне пришлось заняться небольшой отладкой при помощи `print(repr(html))`, чтобы отыскать неполадку. Оказалось, что в результате перехода на шаблоны в конец был внесен дополнительный символ новой строки (`\n`). Мы можем заставить их пройти успешно вот так:

lists/tests.py

```
self.assertTrue(html.strip().endswith('</html>'))
```

Это крошечный обманный прием, но пробел в конце файла с HTML-разметкой действительно не должен иметь для нас значения. Попробуем выполнить тесты еще раз:

```
$ python manage.py test
[...]
OK
```

Рефакторизация кода теперь завершена полностью, и тесты означают наше удовлетворение тем, что поведение осталось нетронутым. Теперь мы можем изменить тесты так, чтобы они больше не тестировали константы – вместо этого они должны проверять, что для преобразования в HTML-разметку мы используем правильный шаблон.

Тестовый клиент Django

Один из способов протестировать это, – вручную преобразовать шаблон в HTML-разметку в самом тесте и затем сравнить с тем, что возвращает представление. В Django есть функция `render_to_string`, которая позволяет это делать:

lists/tests.py

```
from django.template.loader import render_to_string
[...]

    def test_home_page_returns_correct_html(self):
        '''тест: домашняя страница возвращает правильный html'''
        request = HttpRequest()
        response = home_page(request)
        html = response.content.decode('utf8')
        expected_html = render_to_string('home.html')
        self.assertEqual(html, expected_html)
```

Однако такой способ удостовериться, что мы используем верный шаблон, слишком громоздкий, и вся эта суета с `.decode()` и `.strip()` сильно отвлекает. Вместо этого Django предоставляет инструмент под названием «тестовый клиент Django»[7], который имеет встроенные способы проверки использования шаблонов. Вот как он выглядит:

lists/tests.py

```
    def test_home_page_returns_correct_html(self):
        '''тест: домашняя страница возвращает правильный html'''
        response = self.client.get('/')  ❶

        html = response.content.decode('utf8')  ❷
        self.assertTrue(html.startswith('<html>'))
        self.assertIn('<title>To-Do lists</title>', html)
        self.assertTrue(html.strip().endswith('</html>'))

        self.assertTemplateUsed(response, 'home.html')  ❸
```

❶ Вместо создания объекта `HttpRequest` вручную и прямого вызова функции представления мы вызываем метод `self.client.get`, передавая ему URL-адрес, который хотим протестировать.

❷ Пока оставим старые тесты на месте, чтобы удостовериться, что все работает так, как мы планируем.

❸ `.assertTemplateUsed` – этот метод тестирования нам предоставляет класс Django `TestCase`. Он позволяет проверить, какой шаблон ис-

[7] См. https://docs.djangoproject.com/en/1.11/topics/testing/tools/%23the-test-client

пользовался для вывода отклика как HTML (работает только для откликов, которые были получены тестовым клиентом).

И этот тест по-прежнему будет проходить успешно:

```
Ran 2 tests in 0.016s

OK
```

Учитывая, что я всегда с подозрением отношусь к тесту, неполадки которого я не видел, давайте намеренно его нарушим.

lists/tests.py

```
        self.assertTemplateUsed(response, 'wrong.html')
```

Благодаря этому мы также узнаем, как выглядят сообщения теста об ошибках:

```
AssertionError: Ложь не является истиной: Шаблон 'wrong.html' не был
шаблоном, который использовался для вывода отклика в качестве HTML.
Фактически использовался шаблон(ы): home.html
```

Очень полезная информация! Теперь вернем утверждение в прежнее состояние. Пока мы тут, заодно можем удалить наши старые утверждения и старый тест `test_root_url_resolves`, потому что он неявно выполняется тестовым клиентом Django. Мы объединили два многословных теста в один!

lists/tests.py (ch04l010)

```
from django.test import TestCase

class HomePageTest(TestCase):
    '''тест домашней страницы'''

    def test_uses_home_template(self):
        '''тест: используется домашний шаблон'''
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html')
```

Хотя главное в том, что вместо тестирования константы мы тестируем нашу реализацию. Здорово![8]

[8] У вас не получается двигаться дальше, потому что вы не знаете, что это за приписки ch04l0xx рядом с некоторыми распечатками программного кода? Они обозначают конкретные фиксации (https://github.com/hjwp/book-example/commits/chapter_philosophy_and_refactoring) в репозитории с примерами из книги. Это связано с собственными тестами моей книги (https://github.com/hjwp/Book-TDD-Web-Dev-Python/tree/master/tests). Ну, вы знаете, тесты для тестов в книге о тестировании. Они имеют свои собственные тесты, разумеется.

Почему нельзя было использовать тестовый клиент Django с самого начала?

Вы, наверное, задаетесь вопросом, почему мы сразу не использовали тестовый клиент Django? В реальной ситуации именно так бы я и поступил. Но я хотел показать вам «ручную» работу, и тому есть несколько причин. Во-первых, это позволило мне вводить понятия постепенно, одно за другим и сохранять кривую обучения максимально плавной. Во-вторых, вы не всегда можете использовать Django для создания приложений и инструменты тестирования не всегда могут быть доступными, а вот прямой вызов функций и исследование их откликов всегда возможно!

К тому же тестовый клиент Django имеет недостатки; позже в книге мы обсудим разницу между полноизолированными модульными тестами и интегрированными тестами, к которым нас подталкивает тестовый клиент. Но пока это весьма прагматический выбор.

О рефакторизации

Это был абсолютно банальный пример рефакторизации. Но, по выражению Кента Бека в «*Разработке на основе тестирования: на своем примере*» «Рекомендую вам действительно работать таким способом. Точнее, уметь работать таким способом».

По сути, когда я это писал, мои первым инстинктивным желанием было взять и изменить сначала тест (заставить его сразу использовать функцию `assertTemplateUsed`, удалить три лишних утверждения, оставив только проверку содержимого относительно ожидаемого вывода в качестве HTML), а затем пойти вперед и внести изменение в код. Но обратите внимание, в действительности это открывало для меня широкое поле деятельности, чтобы все нарушить. Я мог определить шаблон, который содержал бы *любую* произвольную строку вместо строки с нужными тегами `<html>` и `<title>`.

Во время рефакторизации работайте либо с программным кодом, либо с тестами, но не с двумя сразу.

В ходе рефакторизации всегда существует желание сразу перейти на несколько шагов вперед, чтобы немного подрегулировать поведение. Однако довольно скоро у вас накопятся изменения в полдюжине разных файлов, в результате чего вы совершенно заблудитесь и больше уже ничто не будет работать. Если вы не хотите закончить как кот Том-рефакторщик

(см. ниже), делайте маленькие шаги; продолжайте выполнять рефакторизацию, а функциональные изменения – совершенно отдельно.

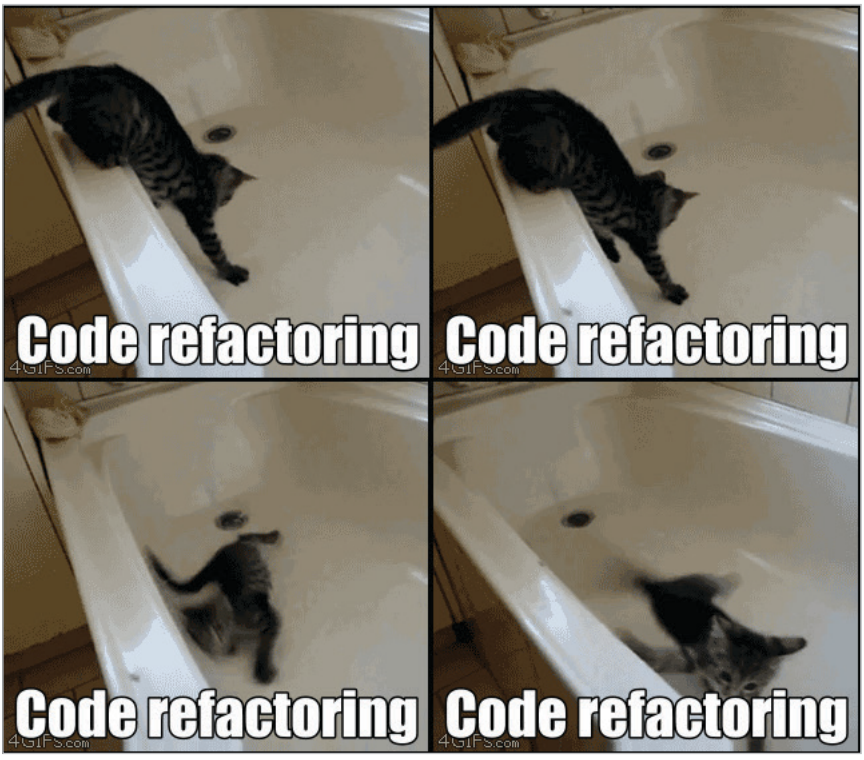


Рис. 4.2. Том-рефакторщик – обязательно посмотрите полный анимированный GIF (Источник: 4GIFs.com)

Мы еще увидим кота Тома-рефакторщика в этой книге как пример того, что происходит, когда мы увлечены и хотим изменить слишком много вещей сразу. Думайте о нем, как о небольшом мультяшном дьяволенке-двойнике Билли-тестировщика, который выскакивает из-за вашего плеча и дает плохие советы...

После любой перестройки кода лучше сразу выполнить фиксацию:

```
$ git status # см. tests.py, views.py, settings.py, + новую папку templates
$ git add . # также добавляем неотслеживаемую папку templates
$ git diff --staged # просматриваем изменения, которые будем фиксировать
$ git commit -m "Рефакторизовано представления домашней страницы для
использования шаблона"
```

Еще немного о главной странице

Тем временем наш функциональный тест по-прежнему не срабатывает. Давайте внесем в программный код фактическое изменение, которое заставит его проходить успешно. Ввиду того, что теперь наш HTML находится в шаблоне, мы можем не стесняться вносить в него изменения без написания любых дополнительных модульных тестов. Мы хотели `<h1>`:

lists/templates/home.html

```
<html>
    <head>
        <title>To-Do lists</title>
    </head>
    <body>
        <h1>Your To-Do list</h1>
    </body>
</html>
```

Теперь убедимся, что нашему функциональному тесту это нравится чуть больше:

```
selenium.common.exceptions.NoSuchElementException: Message: Не получается
локализовать элемент: [id="id_new_item"]
```

Хорошо...

lists/templates/home.html

```
    [...]
        <h1>Your To-Do list</h1>
        <input id="id_new_item" />
    </body>
    [...]
```

А теперь?

```
AssertionError: '' != 'Enter a to-do item'
```

Добавляем замещающий текст.

lists/templates/home.html

```
    <input id="id_new_item" placeholder="Enter a to-do item" />
```

Это дает:

```
selenium.common.exceptions.NoSuchElementException: Message: Не получается
локализовать элемент: [id="id_list_table"]
```

Поэтому мы можем пойти вперед и поместить таблицу на страницу. На данном этапе она будет пустой.

lists/templates/home.html

```
    <input id=”id_new_item” placeholder="Enter a to-do item" />
    <table id="id_list_table">
    </table>
</body>
```

А теперь что говорит ФТ?

```
  File "functional_tests.py", line 43, in
test_can_start_a_list_and_retrieve_it_later
    any(row.text == '1: Купить павлиньи перья' for row in rows)
AssertionError: Ложь не является истиной
```

Несколько загадочно. Воспользуемся номером строки, чтобы найти ошибку. Оказывается, это функция `any`, о которой ранее было сказано много самодовольных слов, или, точнее, утверждение `assertTrue`, у которого нет четкого сообщения о неполадке. В `unittest` большинству методов `assertX` в качестве аргумента можно передать индивидуализированное сообщение об ошибке:

functional_tests.py

```
self.assertTrue(
    any(row.text == '1: Купить павлиньи перья' for row in rows),
    "Новый элемент списка не появился в таблице"
)
```

Если вы снова выполните ФТ, то увидите сообщение:

```
AssertionError: False is not true : Новый элемент списка не появился в таблице
```

Теперь, чтобы заставить его пройти успешно, нам фактически потребуется обработать передачу пользовательской формы. Но это тема следующей главы.

А пока выполним фиксацию:

```
$ git diff
$ git commit -am "HTML главной страницы теперь генерируется из шаблона"
```

Благодаря небольшой рефакторизации мы имеем представление, которое выводит шаблон в качестве HTML. Мы прекратили тестировать константы и теперь находимся в удачном положении, чтобы начать обрабатывать вводимые пользователем данные.

Резюме: процесс TDD

Мы увидели все основные аспекты процесса TDD на практике:

- Функциональные тесты.
- Модульные тесты.
- Цикл «модульный-тест/программный-код».
- Рефакторизация.

Настало время для небольшого резюме и, возможно, даже нескольких блок-схем. Прошу прощения, годы, проведенные в качестве консультанта по управленческим вопросам, меня вконец угробили. Но, с другой стороны, это проявляется в рекурсии.

Каков полный процесс TDD? Взгляните на рис. 4.3.

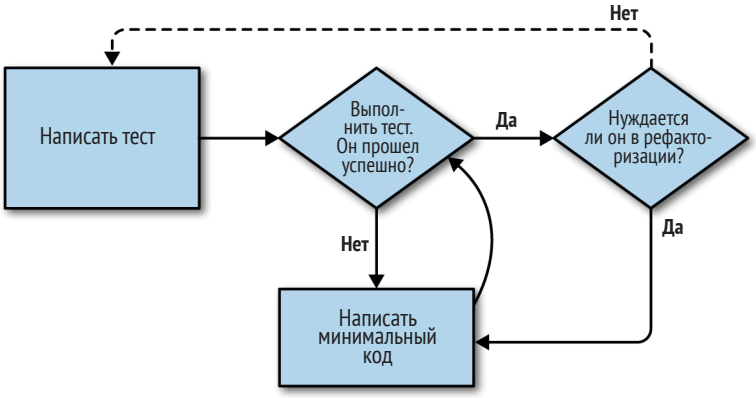


Рис. 4.3. Полный процесс TDD

Мы пишем тест. Мы его выполняем и видим, что он не срабатывает. Мы пишем какой-то минимальный программный код, чтобы продвинуться чуть-чуть дальше. Повторно выполняем тест и повторяем до тех пор, пока он не сработает. Затем, как дополнительный вариант, можно выполнить рефакторизацию программного кода, используя тесты, чтобы удостовериться, что мы ничего не повредили.

Но как это применимо в случае, когда у нас модульные *и* функциональные тесты? Дело в том, что функциональные тесты можно рассматривать как высокоуровневое представление данного цикла, где написание программного кода с целью заставить функциональные тесты проходить успешно фактически сопряжено с применением еще одного, меньшего цикла TDD, который использует модульные тесты. Посмотрите на рис. 4.4.

Мы пишем функциональный тест и видим, что он не срабатывает. Далее идет процесс написания программного кода с целью заставить его пройти успешно, который сам по себе является циклом мини-TDD: мы пишем один или несколько модульных тестов и входим в цикл «модульный-тест/

программный-код» до тех пор, пока модульные тесты не пройдут успешно. Затем возвращаемся к ФТ, чтобы проверить, что он продвинулся немного дальше, и можем чуть-чуть расширить наше приложение, используя больше модульных тестов и т. д.

А как быть с рефакторизацией в контексте функциональных тестов? Здесь она означает, что мы используем функциональный тест, чтобы проверить, что мы сохранили поведение приложения нетронутым, но при этом мы можем изменять или добавлять и удалять модульные тесты и использовать цикл модульных тестов для фактического изменения реализации.

Функциональные тесты – это финальный арбитр в вопросе, работает ваше приложение или нет. Модульные тесты – это инструмент, который помогает на этом пути.

Такой способ взглянуть на вещи иногда называется двухконтурным TDD. Один из моих выдающихся технических рецензентов, Эмили Бах, по данной теме написала пост в блоге[9], который рекомендую в качестве альтернативной точки зрения.

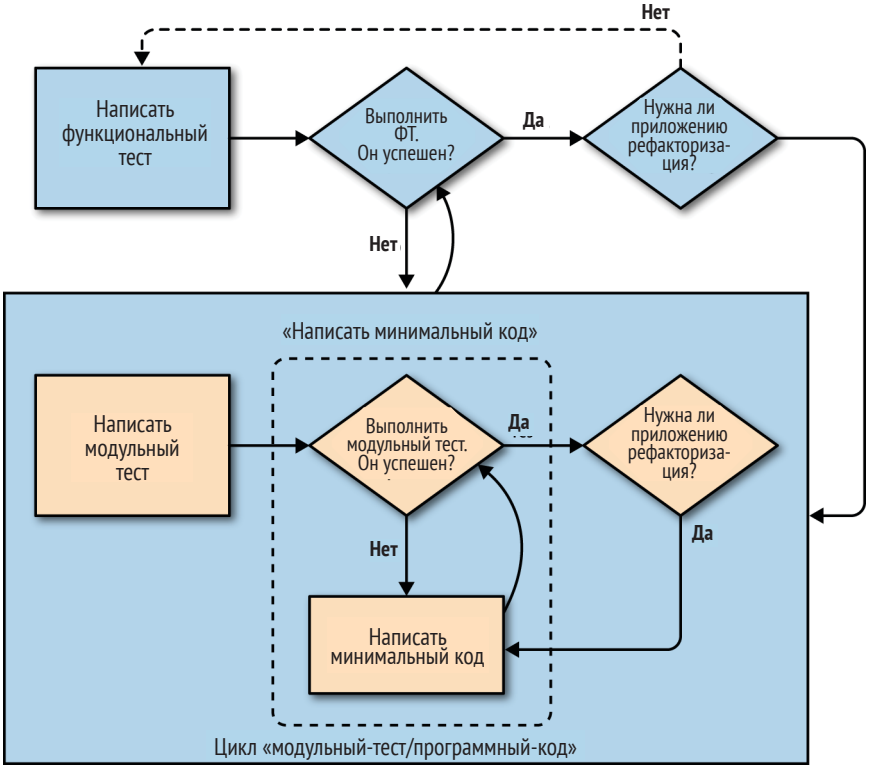


Рис. 4.4. TDD обрабатывает с тестами модуля и функциональным

[9] См. http://bit.ly/1iXzoLR

В следующих главах мы более подробно рассмотрим различные части этого потока операций.

Как проверить программный код либо сразу перескочить вперед (если нужно)

Все примеры программного кода, которые я использовал в книге, доступны в моем репозитории на GitHub[10]. Поэтому, если вы когда-нибудь захотите сравнить свой программный код с моим, можете заглянуть туда.

Каждая глава имеет свою ветку с укороченным названием. К примеру, ветку для данной главы можно найти по ссылке, приведенной в сноске[11]. Это снимок программного кода, каким он должен быть в конце главы.

Вы можете найти полный их список в приложении J вместе с инструкциями о том, как их скачивать или использовать Git для сравнения вашего программного кода с моим.

[10] См. https://github.com/hjwp/book-example/

[11] См. https://github.com/hjwp/book-example/tree/chapter_philosophy_and_refactoring

Глава 5

Сохранение вводимых пользователем данных: тестирование базы данных

Мы хотим взять у пользователя элемент списка неотложных дел и и отправить его на сервер, чтобы каким-то образом его сохранить и позже отобразить пользователю.

Начав писать эту главу, я сразу перешел к тому, что, по моему мнению, является верной структурой программного кода: многочисленным моделям для списков и элементов списков, куче разных URL-адресов для добавления новых списков и элементов, трем новым функциям представления и примерно полудюжине новых модульных тестов для всего вышеупомянутого. Но сумел вовремя остановиться. Несмотря на то что я вполне уверен в своей достаточной осведомленности, чтобы решить все эти задачи сразу, смысл методологии TDD в том, чтобы делать по одной операции за один раз, когда это требуется. Поэтому я решил быть намеренно близоруким и в любой конкретный момент делать только необходимое для того, чтобы продвинуть функциональные тесты немного дальше.

Это демонстрация того, как методология TDD поддерживает итеративный стиль разработки – он может оказаться не самым быстрым маршрутом, но в конце вы обязательно достигнете своей цели. К тому же есть дополнительная выгода: он позволяет мне вводить новые понятия, такие как модели, работающие с POST-запросами, шаблонные теги Django и т. д., *по одному за один раз,* а не выгружать их все на вас одновременно.

Это вовсе не значит, что вы *не должны* пытаться думать наперед и быть умными. В следующей главе мы будем использовать чуть больше конструктивного и опережающего мышления, и покажем, как это укладывается в TDD. Но пока предлагаю пахать с тупым упорством и просто делать то, что нам говорят тесты.

Подключение формы для отправки POST-запроса

В конце предыдущей главы тесты сообщили, что мы не можем сохранять данные, вводимые пользователем. Пока мы будем использовать стандартный HTML-запрос методом POST. Немного скучный, но его довольно хорошо и просто доставлять – мы можем использовать разного рода заводной код HTML5 и JavaScript, об этом я расскажу чуть позже.

Чтобы заставить браузер отправлять POST-запрос, нужно сделать две вещи:

1. Назначить элементу `<input>` атрибут `name=`;
2. Обернуть его в тег `<form>` с методом `method="POST"`.

Скорректируем наш шаблон в *lists/templates/home.html*:

lists/templates/home.html

```
<h1>Your To-Do list</h1>
<form method=”POST”>
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
</form>

<table id="id_list_table">
```

Теперь выполнение ФТ выдает немного загадочную, неожиданную ошибку:

```
$ python functional_tests.py
[...]
Traceback (most recent call last):
  File "functional_tests.py", line 40, in
test_can_start_a_list_and_retrieve_it_later
    table = self.browser.find_element_by_id('id_list_table')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_list_table"]
```

Когда функциональный тест не срабатывает с неожиданной неполадкой, можно предпринять несколько мер для его отладки:

- Добавить операторы `print`, чтобы показать, например, текст текущей страницы.
- Улучшить *сообщение об ошибке*, чтобы показать дополнительную информацию о текущем состоянии.
- Посетить сайт вручную.

- Использовать `time.sleep`, чтобы приостановить тест во время его исполнения.

По ходу книги мы рассмотрим все эти меры, но чаще всего я замечаю, что использую вариант с `time.sleep`. Предлагаю его опробовать.

Что удобно, у нас уже есть задержка `sleep` непосредственно до того, как ошибка происходит, давайте ее просто немного расширим.

functional_tests.py

```
# Когда она нажимает enter, страница обновляется, и теперь страница
# содержит "1: Купить павлиньи перья" в качестве элемента списка
inputbox.send_keys(Keys.ENTER)
time.sleep(10)

table = self.browser.find_element_by_id('id_list_table')
```

В зависимости от того, насколько быстро Selenium работает на вашем ПК, вы уже могли это мельком увидеть. Но когда мы выполним функциональные тесты еще раз, у нас будет время понаблюдать за тем, что происходит: вы увидите страницу, которая выглядит как на рис. 5.1 с большим количеством отладочной информации Django.

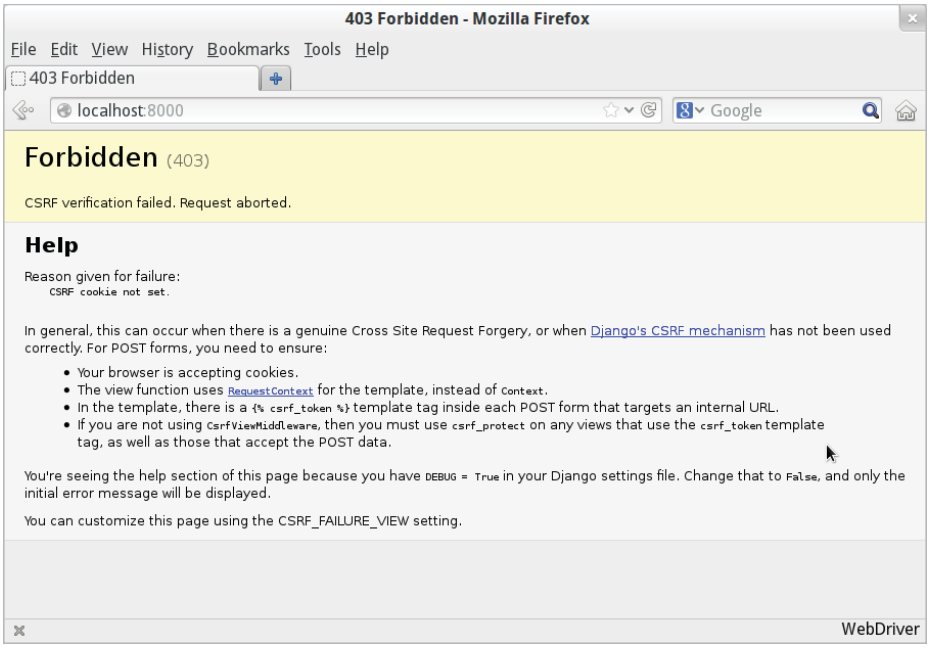


Рис. 5.1. Страница DEBUG в Django, показывающая ошибку CSRF (подделка межсайтовых запросов)

Безопасность: на удивление забавно!

Если вы никогда не слышали о сетевом вторжении с использованием подделки межсайтовых запросов (CSRF от англ. *cross-site request forgery*), почему бы не познакомиться с ним сейчас. Как и обо всех методах взлома системы безопасности, об этом забавно почитать, поскольку они представляют собой использование системы самыми неожиданными способами…

Когда я вернулся в университет, чтобы получить степень в области Computer Science, я записался на модуль по безопасности из чувства долга. *Так и быть, наверное, он будет очень сухим и скучным, но все же, думаю, надо его взять*. Он оказался одним из самых захватывающих модулей всего курса – переполненный радостью взламывания, особым умонастроением, которое требуется, чтобы думать об использовании системы оригинальными способами.

Хочу порекомендовать учебник для своего курса, «*Технологии бесопасности*» Росса Андерсона. Он довольно прост в части чистой криптографии, но переполнен интересными описаниями неожиданных тем, таких как подбор ключей, подделка банкнот, экономия картриджей для струйного принтера и перехват зашифрованных сигналов сверхзвуковых истребителей южноафриканских ВВС на основе атак с повторным навязыванием сообщений. Это огромный том порядка восьми сантиметров толщиной, и я гарантирую вам абсолютно увлекательное чтение.

CSRF-защита Django связана с размещением небольшого автоматически генерируемого маркера в каждую сгенерированную форму, чтобы идентифицировать POST-запросы, прибывающие с исходного сайта. До сих пор нашим шаблоном был чистый HTML, и на этом шаге мы сначала используем шаблонное волшебство Django. Чтобы добавить маркер CSRF, применяем *шаблонный тег*, который синтаксически оформляется фигурными-скобками/знаками процента, `{% ... %}`, которые знамениты тем, что представляют собой самую раздражающую в мире двухклавишную сенсорную комбинацию:

lists/templates/home.html

```
<form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
    {% csrf_token %}
</form>
```

Во время вывода в виде HTML Django их заменит на `<input type="hidden">` с маркером CSRF. Повторное выполнение функционального теста выдаст ожидаемую неполадку:

```
AssertionError: False is not true : Новый элемент списка не появился в таблице
```

Поскольку продолжительный `time.sleep` по-прежнему присутствует, этот тест приостановится на заключительном экране, показывая, что текст нового элемента исчезает после того, как форма отправлена, и страница обновляется, снова показывая пустую форму. Все потому, что мы еще не подключили сервер для работы с POST-запросом – он просто его игнорирует и выводит на экран обычную домашнюю страницу.

Правда, теперь мы можем снова разместить нормальный короткий `time.sleep`:

functional_tests.py

```
# Когда она нажимает Enter, страница обновляется, и теперь страница
# содержит "1: Купить павлиньи перья" в качестве элемента списка
inputbox.send_keys(Keys.ENTER)
time.sleep(1)

table = self.browser.find_element_by_id('id_list_table')
```

Обработка POST-запроса на сервере

Поскольку в форме мы не указали атрибут `action=`, он отправляется назад по тому же URL-адресу, откуда он был передан по умолчанию (то есть `/`) и который обрабатывается функцией `home_page`. Адаптируем это представление с учетом обработки POST-запроса.

Это означает новый модульный тест для представления `home_page`. Откройте *lists/tests.py* и добавьте новый метод в `HomePageTest`:

lists/tests.py (ch05l005)

```
def test_uses_home_template(self):
    '''тест: используется домашний шаблон'''
    response = self.client.get('/')
    self.assertTemplateUsed(response, 'home.html')

def test_can_save_a_POST_request(self):
    '''тест: можно сохранить post-запрос''
    response = self.client.post('/', data={'item_text': 'A new list item'})
    self.assertIn('A new list item', response.content.decode())
```

Чтобы выполнить POST-запрос, мы вызываем `self.client.post`, и, как видите, он принимает аргумент `data` с данными формы, которые мы хотим отправить. Затем мы удостоверяемся, что текст из нашего POST-запроса появляется в сгенерированном HTML. Это дает ожидаемую неполадку:

```
$ python manage.py test
[...]
```

```
AssertionError: 'A new list item' not found in '<html>\n    <head>\n
<title>To-Do lists</title>\n    </head>\n    <body>\n    <h1>Your To-Do
list</h1>\n       <form method="POST">\n          <input name="item_text"
[...]
</body>\n</html>\n'
```

Можно заставить тест пройти успешно, добавив `if` и предоставив другую ветвь кода для POST-запроса. В типичном стиле TDD мы начинаем с намеренно глупого возвращаемого значения:

lists/views.py

```
from django.http import HttpResponse
from django.shortcuts import render

def home_page(request):
    '''домашняя страница'''
    if request.method == 'POST':
        return HttpResponse(request.POST['item_text'])
    return render(request, 'home.html')
```

Это заставляет наши модульные тесты пройти успешно, на самом деле мы добиваемся не этого. На самом деле мы хотим добавить передачу POST-данных в таблицу в шаблоне домашней страницы.

Передача переменных Python для вывода в шаблоне

Выше мы уже слегка коснулись этой темы, но теперь пришла пора познакомиться с истинной мощью шаблонного синтаксиса Django, то есть передачей переменных из программного кода представления в шаблоны HTML.

Для начала посмотрим, как шаблонный синтаксис позволяет включать объект Python в шаблон. Форма записи `{{...}}` отображает объект в виде строкового значения:

lists/templates/home.html

```
<body>
    <h1>Your To-Do list</h1>
    <form method="POST">
        <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
        {% csrf_token %}
    </form>

    <table id="id_list_table">
        <tr><td>{{ new_item_text }}</td></tr>
```

```
    </table>
</body>
```

Скорректируем модульный тест, чтобы он проверял, что мы по-прежнему используем шаблон:

lists/tests.py

```
    def test_can_save_a_POST_request(self):
        '''тест: можно сохранить post-запрос'''
        response = self.client.post('/', data={'item_text': 'A new list item'})
        self.assertIn('A new list item', response.content.decode())
        self.assertTemplateUsed(response, 'home.html')
```

Как и ожидалось, этот программный код вызовет неполадку:

```
AssertionError: Не использован шаблон для вывода отклика в качестве HTML
```

Хорошо, наше преднамеренно глупое возвращаемое значение больше не обманывает тесты, поэтому можно переписать это представление и сообщить ему, чтобы он передавал параметр POST в шаблон. Функция `render` в качестве третьего аргумента принимает словарь, который отображает имена переменных на их значения:

lists/views.py (ch05l009)

```
def home_page(request):
    '''домашняя страница'''
    return render(request, 'home.html', {
        'new_item_text': request.POST['item_text'],
    })
```

Снова выполняем модульные тесты:

```
ERROR: test_uses_home_template (lists.tests.HomePageTest)
[...]
  File "/.../superlists/lists/views.py", line 5, in home_page
    'new_item_text': request.POST['item_text'],
[...]
django.utils.datastructures.MultiValueDictKeyError: "'item_text'"
```

Неожиданная неполадка.

Если вы помните правила чтения отчетов об обратных трассировках, вы сразу определите, что на самом деле это неполадка в *другом* тесте. Мы заставили фактический тест, с которым работали, пройти успешно, но модульные тесты засекли неожиданное последствие, регрессию: мы повредили ветвь кода, где нет POST-запроса.

В этом вся суть использования тестов. Да, мы могли предсказать, что это произойдет, но представьте ситуацию, что у нас был плохой день или мы работали невнимательно: тесты только что спасли нас от случайного повреждения приложения и благодаря TDD мы сразу же об этом узнали. Нам не пришлось ждать группу контроля качества либо переходить на веб-браузер и кликать вручную. Можно продолжить работу, исправив все на месте. Вот так:

lists/views.py

```
def home_page(request):
    '''домашняя страница'''
    return render(request, 'home.html', {
        'new_item_text': request.POST.get('item_text', ''),
    })
```

Ищите `dict.get`[1], если вы не уверены в том, что именно там происходит.

Теперь модульные тесты должны пройти успешно. Посмотрим, что говорят функциональные тесты:

```
AssertionError: False is not true: Новый элемент списка не появился в таблице
```

Если в этом или любом другом месте этой главы ваши функциональные тесты показывают другую ошибку, жалуясь на `StaleElementReferenceException`, возможно, потребуется увеличить явное ожидание `time.sleep` – попробуйте 2 или 3 секунды вместо 1; затем продолжайте читать – в следующей главе будет дано более надежное решение.

Хм, эта ошибка не очень-то помогает. Давайте воспользуемся еще одним способом отладки ФТ – улучшением сообщения об ошибке. Это, вероятно, самый конструктивный метод, потому что улучшенные сообщения об ошибках всегда рядом и помогают в отладке любых будущих ошибок:

functional_tests.py (ch05l011)

```
self.assertTrue(
    any(row.text == '1: Купить павлиньи перья' for row in rows),
    f"Новый элемент списка не появился в таблице. Содержимым было:\
n{table.text}" ❶
)
```

❶ Если вы не встречали такую синтаксическую конструкцию раньше, то знайте: это новый синтаксис Python для f-строк (вероятно,

[1] См. http://docs.python.org/3/library/stdtypes.html%23dict.get

самое захватывающее нововведение в Python 3.6). Просто начинаете строку литерой `f` и затем можете использовать синтаксис с фигурной скобкой для вставки локальных переменных. Дополнительную информацию смотрите в сопроводительных записках о версии Python 3.6[2].

Это дает нам более полезное сообщение об ошибке:

```
AssertionError: False is not true : Новый элемент списка не появился в таблице.
Содержимым было:
Купить павлиньи перья
```

Знаете, что может быть лучше? Сделать это утверждение чуть менее умным. Возможно, вы помните, как я был доволен собой, что использовал функцию `any`. Но один из моих предрелизных читателей (спасибо, Джейсон!) предложил намного более простую реализацию. Мы можем заменить все четыре строки `assertTrue` единственным `assertIn`:

functional_tests.py (ch05l012)

```
self.assertIn('1: Купить павлиньи перья', [row.text for row in rows])
```

Теперь гораздо лучше. Если нам вдруг начинает казаться, что мы очень умны, то это прекрасный повод для беспокойства. На самом деле, скорее всего, мы слишком все усложняем. И в итоге запросто получаем сообщение об ошибке:

```
    self.assertIn('1: Купить павлиньи перья', [row.text for row in rows])
AssertionError: '1: Купить павлиньи перья' not found in ['Купить павлиньи перья']
```

Считайте, что я был должным образом наказан.

Если вместо этого ФТ говорит, что таблица пуста ("not found in []"), проверьте тег `<input>` – имеет ли он правильный атрибут `name="item_text"`? Без него вводимые пользователем данные не будут связаны с верным ключом в `request.POST`.

Дело в том, что ФТ хочет, чтобы мы перечисляли элементы списка со строковым значением «1:» в начале первого элемента. Самое быстрое, что заставит его пройти успешно, – это обманным путем внести оперативное изменение в шаблон:

lists/templates/home.html

```
    <tr><td>1: {{ new_item_text }}</td></tr>
```

[2] См. https://docs.python.org/3/whatsnew/3.6.html%23pep-498-formatted-string-literals

Правило «Красный/зеленый/рефакторизуй» и триангуляция

Цикл «модульный-тест/программный-код» иногда подается как правило «*красный*, *зеленый*, *рефакторизуй*»:

- Начать с написания модульного теста, который не срабатывает (красный уровень).
- Написать простейший код, который заставляет его пройти успешно (зеленый уровень), *даже если это означает обман*.
- Выполнить рефакторизацию, чтобы получить более хороший программный код, в котором больше смысла.

Итак, что же мы делаем во время этапа рефакторизации? Что является причиной перейти от реализации, в которой мы прибегаем к различным уловкам, к той, которой мы довольны?

Одним таким методом является *устранение дублирований*: если ваш тест использует волшебную константу (как константа «1:» в начале элемента списка) и прикладной код ее тоже использует, это считается дублированием и такая ситуация оправдывает рефакторизацию. Удаление волшебной константы из прикладного кода обычно означает, что вам больше не надо прибегать к уловкам.

Считаю, что это выглядит немного расплывчато, поэтому мне больше нравится использовать второй метод, который называется *триангуляцией*: если ваши тесты позволяют писать «обманный» код, которым вы недовольны (например, возвращение волшебной константы), *напишете еще один тест*, который вынуждает написать более хороший код. Это как раз то, что мы делаем, когда расширяем ФТ, чтобы убедиться, что мы получаем «2:» при вводе *второго* элемента списка.

Теперь мы добрались до `self.fail (' Закончить тест!')`. Если мы расширяем ФТ, чтобы проверить добавление второго элемента в таблицу (копипаста – всегда в помощь), мы видим, что «примерочный» вариант решения в действительности не прокатывает:

functional_tests.py

```
# Текстовое поле по-прежнему приглашает ее добавить еще один элемент. Она
# вводит "Сделать мушку из павлиньих перьев" (Эдит очень методична)
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Сделать мушку из павлиньих перьев')
inputbox.send_keys(Keys.ENTER)
time.sleep(1)

# Страница снова обновляется и теперь показывает оба элемента ее списка
table = self.browser.find_element_by_id('id_list_table')
rows = table.find_elements_by_tag_name('tr')
```

```
self.assertIn('1: Купить павлиньи перья', [row.text for row in rows])
self.assertIn(
    '2: Сделать мушку из павлиньих перьев',
     [row.text for row in rows]
)

# Эдит интересно, запомнит ли сайт ее список. Далее она видит, что
# сайт сгенерировал для нее уникальный URL-адрес – об этом
# выводится небольшой текст с пояснениями.
self.fail('Закончить тест!')

# Она посещает этот URL-адрес – ее список по-прежнему там.
```

Разумеется, функциональные тесты возвращают ошибку:

```
AssertionError: '1: Купить павлиньи перья' not found in ['1: Сделать мушку
из павлиньих перьев']
```

Если клюнуло трижды, рефакторизуй

Прежде чем мы пойдем дальше – в этом ФТ есть плохой *код с душком*[3]. У нас три почти идентичных блока кода, которые проверяют новые элементы в таблице списка. Есть принцип, который гласит *«не повторяйся»* (DRY – don't repeat yourself), и нам нравится его применять, следуя мантре «*если клюнуло трижды – рефакторизуй*». Вы можете скопировать и вставить программный код один раз, и может случиться так, что попытка удалить получившееся в результате дублирования будет преждевременной. Но если произошло три таких случая – пора удалять дублирование.

Начинаем с фиксации того, что у нас есть к этому моменту. Хотя мы знаем, что наш сайт имеет главный дефект – он может обрабатывать всего один элемент списка, – все же это больше, чем было. Возможно, нам придется все это переписать, а может и нет, но правило гласит, что прежде чем делать какую-либо перестройку кода, надо выполнить фиксацию:

```
$ git diff
# должна показать изменения в functional_tests.py, home.html,
# tests.py и views.py
$ git commit -a
```

Возвращаясь к рефакторизации функционального теста: мы можем применить локальную (inline) функцию, но это слегка нарушит последо-

[3] Если вы не сталкивались с данным понятием, «код с душком» имеет отношение к фрагменту программного кода, который заставляет вас его переписывать. У Джеффа Этвуда (Jeff Atwood) имеется подборка в его блоге Coding Horror (http://www.codinghorror.com/blog/2006/05/code-smells.html). Чем больше опыта вы накапливаете как программист, тем чувствительнее становится ваш нос к коду с душком…

вательность операций в тесте. Воспользуемся вспомогательным методом (напомню, что в качестве тестов будут выполняться только те методы, которые начинаются с префикса `test_`, в результате чего другие методы можно применять в собственных целях):

functional_tests.py

```
    def tearDown(self):
        '''демонтаж'''
        self.browser.quit()

    def check_for_row_in_list_table(self, row_text):
        '''подтверждение строки в таблице списка'''
        table = self.browser.find_element_by_id('id_list_table')
        rows = table.find_elements_by_tag_name('tr')
        self.assertIn(row_text, [row.text for row in rows])

    def test_can_start_a_list_and_retrieve_it_later(self):
        '''тест: можно начать список и получить его позже'''
        [...]
```

Мне нравится помещать вспомогательные методы ближе к вершине класса, между `tearDown` и первым тестом. Давайте применим это в ФТ:

functional_tests.py

```
# Когда она нажимает enter, страница обновляется, и теперь страница
# содержит "1: Купить павлиньи перья" в качестве элемента таблицы списка
inputbox.send_keys(Keys.ENTER)
time.sleep(1)
self.check_for_row_in_list_table('1: Купить павлиньи перья')

# Текствое поле по-прежнему приглашает ее добавить еще один элемент.
# Она вводит "Сделать мушку из павлиньих перьев"
# (Эдит очень методична)
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Сделать мушку из павлиньих перьев')
inputbox.send_keys(Keys.ENTER)
time.sleep(1)

# Страница снова обновляется и теперь показывает оба элемента ее списка
self.check_for_row_in_list_table('1: Купить павлиньи перья')
self.check_for_row_in_list_table('2: Сделать мушку из павлиньих перьев')

# Эдит интересно, запомнит ли сайт ее список. Далее она видит, что
[...]
```

Выполняем ФТ еще раз, чтобы проверить, что он ведет себя так же…

```
AssertionError: '1: Купить павлиньи перья' not found in ['1: Сделать мушку
из павлиньих перьев']
```

Хорошо. Теперь мы можем зафиксировать рефакторизацию ФТ, как свое собственное небольшое, атомарное изменение:

```
$ git diff # покажет изменения в functional_tests.py
$ git commit -a
```

И вернемся к работе. Если когда-либо придется обрабатывать более одного элемента списка, нам потребуется какой-то метод долговременного хранения. Базы данных – надежное решение этой задачи.

Django ORM и первая модель

Объектно-реляционный преобразователь (ORM от англ. *Object-Relational Mapper*) – это уровень абстракции данных, хранящихся в базе данных с таблицами, строками и столбцами. Он позволяет работать с базами данных, используя знакомые объектно-ориентированные метафоры, которые хорошо работают с кодом. Классы отображаются на таблицы базы данных, атрибуты отображаются на столбцы, и отдельный экземпляр класса представляет строку данных в базе данных.

Django поставляется с превосходным ORM, и написание модульного теста, который его использует, фактически представляет собой превосходный способ его изучить, поскольку он реализует код через указание того, как мы хотим, чтобы он работал.

Создадим новый класс в *lists/tests.py*:

lists/tests.py

```
from lists.models import Item
[...]

class ItemModelTest(TestCase):
    '''тест модели элемента списка'''

    def test_saving_and_retrieving_items(self):
        '''тест сохранения и получения элементов списка'''
        first_item = Item()
        first_item.text = 'The first (ever) list item'
        first_item.save()

        second_item = Item()
        second_item.text = 'Item the second'
```

```
    second_item.save()

    saved_items = Item.objects.all()
    self.assertEqual(saved_items.count(), 2)

    first_saved_item = saved_items[0]
    second_saved_item = saved_items[1]
    self.assertEqual(first_saved_item.text, 'The first (ever) list item')
    self.assertEqual(second_saved_item.text, 'Item the second')
```

Вы видите, что создание новой записи в базе данных является относительно простым процессом создания объекта, присвоения нескольких атрибутов и вызова функции `.save()`. Django также предоставляет API для опроса базы данных через атрибут класса, `.objects`, и мы используем самый простой запрос – `.all()`, который получает все записи для заданной таблицы. Результаты возвращаются в виде похожего на список объекта под названием `QuerySet`, из которого можно извлекать отдельные объекты, а также вызвать дальнейшие функции, такие как `.count()`. Затем мы проверяем, сохранились ли объекты в базе данных, чтобы убедиться, что информация сохранена корректно .

ORM Django имеет много других полезных и интуитивно понятных возможностей; сейчас, пожалуй, самое время пробежаться по учебному руководству Django[4], в котором дается превосходное введение в эти возможности.

Я написал этот модульный тест в очень многословном стиле как способ представить Django ORM. Не рекомендую писать подобного рода тесты своей модели в реальной ситуации. Позже, в главе 15 мы этот тест перепишем, чтобы он стал намного короче.

Терминология 2: модульные тесты против интегрированных и база данных

Поборники чистоты вам скажут, что реальный модульный тест никогда не должен касаться базы данных и тест, который я только что написал, следует назвать скорее интегрированным, потому что он не только тестирует программный код, но и опирается на внешнюю систему – базу данных.

Ничего страшного, если мы пока проигнорируем эту разницу – у нас два типа тестов: высокоуровневые функциональные, которые тестируют приложение с точки зрения пользователя, и более низкоуровневые, которые тестируют его с точки зрения программиста.

Мы вернемся к этому вопросу и поговорим о модульных и интегрированных тестах ближе к концу книги, в главе 23.

[4] См. https://docs.djangoproject.com/en/1.11/intro/tutorial01/

Теперь попытаемся выполнить модульный тест. Наступает еще один цикл «модульный-тест/программный-код»:

```
ImportError: невозможно импортировать имя 'Item'
```

Очень хорошо, давайте дадим ему что-нибудь для импорта из *lists/models.py*. Мы чувствуем себя уверенно, поэтому пропустим шаг `Item = None` и сразу приступим к созданию класса:

lists/models.py

```
from django.db import models

class Item(object):
    '''элемент списка'''
    pass
```

Это приводит тест к:

```
    first_item.save()
AttributeError: У объекта 'Item' нет атрибута 'save'
```

Чтобы предоставить нашему классу `Item` метод `save` и превратить его в реальную модель Django, мы заставляем его наследовать от класса `Model`:

lists/models.py

```
from django.db import models

class Item(models.Model):
    '''элемент списка'''
    pass
```

Первая миграция базы данных

На следующем шаге происходит ошибка базы данных:

```
django.db.utils.OperationalError: нет такой таблицы: lists_item
```

В Django задача ORM – моделировать базу данных, однако еще существует вторая система, которая отвечает за фактическое создание базы данных, – *миграция*. Ее задача – давать возможность добавлять и удалять таблицы и столбцы, основываясь на изменениях, вносимых в файлы *models.py*.

Ее можно представить как систему управления версиями базы данных. Как мы позже увидим, она особенно полезна в ситуации, когда нужно обновить базу данных, развернутую на работающем сервере.

Пока нам нужно только знать, как создать первую миграцию базы данных, что мы и делаем при помощи команды `makemigrations`[5]:

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0001_initial.py
    - Create model Item
$ ls lists/migrations
0001_initial.py __init__.py __pycache__
```

Если вам любопытно, можете взглянуть на файл миграций. Вы увидите, что он представляет собой наши дополнения в *models.py*.

Тем временем мы должны убедиться, что наши тесты продвинулись немного дальше.

Тест продвинулся удивительно далеко

Тест на самом деле продвинулся удивительно далеко:

```
$ python manage.py test lists
[...]
    self.assertEqual(first_saved_item.text, 'The first (ever) list item')
AttributeError: У объекта 'Item' нет атрибута 'text'
```

Это на целых восемь строк ниже, чем последняя неполадка – мы прошли весь процесс сохранения этих двух элементов, проверили, что они сохранены в базе данных, однако платформа Django, похоже, не запомнила атрибут `.text`.

К слову, если вы в Python новичок, вероятно, вас удивило, что можно вообще присваивать атрибут `.text`. В таких языках, как Java, вы бы, наверное, получили ошибку компиляции. В Python правила менее строгие.

Классы, которые наследуют от `models.Model`, отображаются в таблицы базы данных. По умолчанию они получают автоматически сгенерированный атрибут ID, который будет столбцом первичного ключа в базе данных. Но вам нужно в явном виде задать любые другие столбцы. Вот как мы организуем текстовое поле:

lists/models.py

```
class Item(models.Model):
    '''элемент списка'''
    text = models.TextField()
```

В Django имеется много других типов полей, таких как `IntegerField`, `CharField`, `DateField` и т. д. Я выбрал `TextField` вместое `CharField`, потому

[5] Вам интересно, когда мы займемся выполнением команд «migrate» и «makemigrations»? Продолжайте читать, они ждут вас уже в этой главе.

что в последнем есть ограничение длины, которое в этой точке кажется необоснованным. Подробнее о типах полей вы можете почитать в учебном руководстве Django и в документации[6].

Новое поле означает новую миграцию

Тесты указывают на еще одну ошибку базы данных:

```
django.db.utils.OperationalError: нет такого столбца: lists_item.text
```

Дело в том, что мы добавили в базу данных дополнительное поле, а значит, нужно создать еще одну миграцию. Круто, когда тесты об этом сообщают!

Попробуем:

```
$ python manage.py makemigrations
You are trying to add a non-nullable field 'text' to item without a default;
we can't do that (the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a
null value for this column)
 2) Quit, and let me add a default in models.py
Select an option:2
```

Ой. Эта команда не позволяет добавлять столбец без значения по умолчанию. Давайте выберем опцию 2 (выйти и позволить мне добавить в models.py значение по умолчанию) и установим в *models.py* значение по умолчанию. Полагаю, синтаксис не требует объяснений:

lists/models.py

```
class Item(models.Model):
    '''элемент списка'''
    text = models.TextField(default='')
```

И теперь миграция должна завершиться:

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0002_item_text.py
    - Add field text to item
```

Итак, две новые строки в *models.py*, две миграции базы данных и, как результат, атрибут `.text` на модельных объектах теперь распознается как специальный атрибут, поэтому он сохраняется в базе данных и тесты проходят…

[6] См. http://bit.ly/1slDAGH и https://docs.djangoproject.com/en/1.11/ref/models/fields/.

```
$ python manage.py test lists
[...]

Ran 3 tests in 0.010s
OK
```

Сделаем фиксацию самой первой модели!

```
$ git status # см. tests.py, models.py и 2 неотслеживаемые миграции
$ git diff # покажет изменения в tests.py и models.py
$ git add lists
$ git commit -m "Модель для элементов списка и связанная с ней миграция"
```

Сохранение POST-запроса в базу данных

Скорректируем тест для POST-запроса нашей домашней страницы и укажем, что представление должно сохранять новый элемент в базе данных, а не передавать его прямо в соответствующий отклик. Для этого добавим в существующий тест `test_home_page_can_save_a_POST_request` три новые строки:

lists/tests.py

```
def test_can_save_a_POST_request(self):
    '''тест: можно сохранить post-запрос'''
    response = self.client.post('/', data={'item_text': 'A new list item'})

    self.assertEqual(Item.objects.count(), 1)  ❶
    new_item = Item.objects.first()  ❷
    self.assertEqual(new_item.text, 'A new list item')  ❸

    self.assertIn('A new list item', response.content.decode())
    self.assertTemplateUsed(response, 'home.html')
```

❶ Мы убеждаемся, что один новый объект `Item` был сохранен в базе данных. `objects.count()` – сокращение для `objects.all().count()`.

❷ `objects.first()` – то же самое, что и `objects.all()[0]`.

❸ Проверяем, что текст элемента правильный.

Этот тест становится многословным. Похоже, он тестирует много разных вещей. Это еще один *код с душком* – длинный модульный тест нужно разделить пополам, иначе он будет говорить, что тестируемый элемент слишком сложный. Добавим это в наш собственный рабочий список неотложных дел, например в блокноте:

Записав это в таком вот блокноте, мы гарантируем себе, что не забудем, и чувствуем себя уверенно, возвращаясь к тому, с чем мы работали. Выполняем тесты повторно и видим ожидаемую неполадку:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

Скорректируем наше представление:

lists/views.py

```
from django.shortcuts import render
from lists.models import Item

def home_page(request):
    '''домашняя страница'''
    item = Item()
    item.text = request.POST.get('item_text', '')
    item.save()
    return render(request, 'home.html', {
        'new_item_text': request.POST.get('item_text', ''),
    })
```

Я запрограммировал очень наивное решение, и вы наверняка заметили весьма очевидную проблему: с каждым запросом к домашней странице мы будем сохранять пустые элементы. Добавим это в список задач, подлежащих исправлению. У нас все еще нет способа предоставлять разным людям разные списки. Пока мы это проигнорируем.

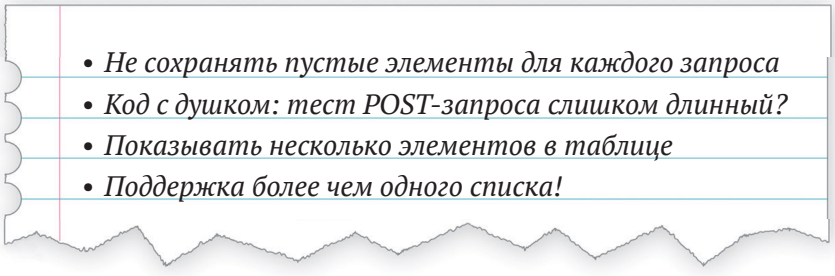
Однако я не утверждаю, что такие вопиющие проблемы всегда следует игнорировать в «реальной жизни». Каждый раз, когда мы обнаруживаем проблемы, требуется принимать решение на свое усмотрение – остановить ли то, что вы делаете, и начать снова, или же отложить решение проблемы на потом. Иногда следует довести начатое до конца, не обращая внимание на выявленные сложности, но иногда проблема столь велика, что требует остановки и переосмысления.

Посмотрим, как поживают модульные тесты. Они проходят! Хорошо. Можно сделать небольшую рефакторизацию:

lists/views.py

```
    return render(request, 'home.html', {
        'new_item_text': item.text
    })
```

Теперь заглянем в наш блокнот. Я добавил еще пару вещей, которые нас интересуют:



Начнем с первого. Мы можем закрепить тестирующее утверждение за существующим тестом, но лучше давать модульным тестам тестировать по одной вещи за раз, поэтому предлагаю добавить новый тест:

lists/tests.py

```
class HomePageTest(TestCase):
    '''тест домашней страницы'''
    [...]

    def test_only_saves_items_when_necessary(self):
        '''тест: сохраняет элементы, только когда нужно'''
        self.client.get('/')
        self.assertEqual(Item.objects.count(), 0)
```

Это дает нам неполадку `1 != 0`. Исправим ее. Будьте начеку: хотя такое изменение логики представления является совершенно незначительным, в реализацию кода вносится довольно много мелких правок:

lists/views.py

```
def home_page(request):
    '''домашняя страница'''
    if request.method == 'POST':
        new_item_text = request.POST['item_text']  ❶
        Item.objects.create(text=new_item_text)  ❷
    else:
        new_item_text = ''  ❶
```

```
    return render(request, 'home.html', {
        'new_item_text': new_item_text,  1
    })
```

❶ Мы используем переменную `new_item_text`, которая будет хранить содержимое POST-запроса либо пустую строку.

❷ `.objects.create` – это лаконично сокращенное создание нового объекта `Item` без необходимости вызывать метод `.save()`.

Теперь тест проходит:

```
Ran 4 tests in 0.010s

OK
```

Переадресация после POST-запроса

Но вот в чем дело: вся эта свистопляска вокруг `new_item_text = ''` сильно огорчает. К счастью, теперь мы можем все исправить. Функция представления имеет две задачи: обработать вводимые пользователем данные и вернуть надлежащий отклик. Мы позаботились о первой части – сохранили вводимые пользователем данные в базе. Теперь предлагаю поработать со второй частью.

Нам говорят: «Всегда переадресуйте после POST-запроса»[7]. Что ж, так и поступим. Еще раз внесем изменение в модульный тест, чтобы сохранить POST-запрос, сообщая, что вместо вывода отклика в качестве HTML вместе с элементом внутри он должен переадресовать назад, на домашнюю страницу:

lists/tests.py

```
def test_can_save_a_POST_request(self):
    '''тест: можно сохранить post-запрос'''
    response = self.client.post('/', data={'item_text': 'A new list item'})

    self.assertEqual(Item.objects.count(), 1)
    new_item = Item.objects.first()
    self.assertEqual(new_item.text, 'A new list item')

    self.assertEqual(response.status_code, 302)
    self.assertEqual(response['location'], '/')
```

Мы больше не ожидаем отклика с содержимым `.content`, сгенерированным шаблоном, поэтому утверждения, которые за этим следят, исключаются. Теперь отклик будет представлять собой HTTP-*переадресацию*, кото-

[7] См. https://en.wikipedia.org/wiki/Post/Redirect/Get

рая имеет код состояния 302 и направляет браузер в новое место.

Это дает нам ошибку `200 != 302`. Мы можем существенно подчистить представление:

lists/views.py (ch05l028)

```
from django.shortcuts import redirect, render
from lists.models import Item

def home_page(request):
    '''домашняя страница'''
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/')

    return render(request, 'home.html')
```

И теперь тесты должны пройти:

```
Ran 4 tests in 0.010s

OK
```

Лучшие приемы модульного тестирования: каждый тест должен проверять одну единицу кода

Теперь представление выполняет переадресацию после POST-запроса, что является плюсом, и мы несколько сократили модульный тест, но можем добиться еще большего успеха.

Положительная практика модульного тестирования говорит, что один тест должен проверять всего одну единицу кода – так легче отслеживать дефекты. Наличие многочисленных тестирующих утверждений означает, что, если тест не срабатывает на раннем утверждении, вы не знаете, каково состояние более поздних утверждений. Если мы случайно повредим это представление, нам нужно знать, что именно нарушилось: сохранение объектов или же тип отклика.

Еще не факт, что вы всегда с первой попытки будете писать идеальные модульные тесты с одиночными тестирующими утверждениями. Но теперь, кажется, самое время разделить компетенции:

lists/tests.py

```
def test_can_save_a_POST_request(self):
    '''тест: можно сохранить post-запрос'''
    self.client.post('/', data={'item_text': 'A new list item'})
```

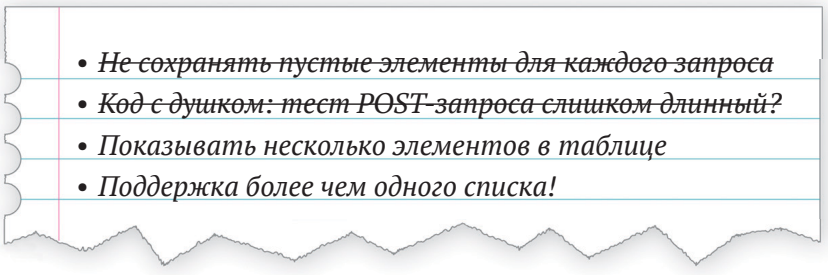
```
    self.assertEqual(Item.objects.count(), 1)
    new_item = Item.objects.first()
    self.assertEqual(new_item.text, 'A new list item')

def test_redirects_after_POST(self):
    '''тест: переадресует после post-запроса'''
    response = self.client.post('/', data={'item_text': 'A new list item'})
    self.assertEqual(response.status_code, 302)
    self.assertEqual(response['location'], '/')
```

Теперь мы должны увидеть, что вместо четырех тестов проходят пять:

```
Ran 5 tests in 0.010s

OK
```

Генерирование элементов в шаблоне

Теперь намного лучше! Вернемся к нашему рабочему списку неотложных дел:



От вычеркивания пунктов из списка получаешь такое же удовлетворение, как и от работающих тестов!

Третий элемент – последний из «простых». Давайте создадим новый модульный тест, который проверяет, что шаблон также может отображать многочисленные элементы списка:

lists/tests.py

```
class HomePageTest(TestCase):
    '''тест домашней страницы'''
    [...]

    def test_displays_all_list_items(self):
        '''тест: отображаются все элементы списка'''
```

```
        Item.objects.create(text='itemey 1')
        Item.objects.create(text='itemey 2')

        response = self.client.get('/')

        self.assertIn('itemey 1', response.content.decode())
        self.assertIn('itemey 2', response.content.decode())
```

Интересуетесь межстрочными интервалами в тесте? Я группирую тест следующим образом: две строки в начале – они настраивают тест, одна строка в середине – она фактически вызывает программный код, подвергаемый проверке, и в конце – утверждения. Не обязательно должно быть именно так, но это действительно помогает видеть структуру теста. Настроить, осуществить, утвердить – вот типичная схема модульного теста.

Тест не срабатывает, как и ожидалось:

```
AssertionError: 'itemey 1' not found in '<html>\n <head>\n [...]
```

Шаблонный синтаксис Django имеет тег для итеративного обхода списков, `{% for .. in ..%}`. Мы можем применить его следующим образом:

lists/templates/home.html

```
<table id="id_list_table">
    {% for item in items %}
        <tr><td>1: {{ item.text }}</td></tr>
    {% endfor %}
</table>
```

Это один из главных плюсов системы шаблонной обработки. Теперь шаблон сгенерирует HTML с многочисленными строками `<tr>`, по одной для каждого элемента в переменной `items`. Отлично! По ходу я представлю еще несколько фрагментов шаблонного волшебства Django, но в определенный момент вы захотите почитать об остальных в документации Django[8].

Простое изменение шаблона не делает наши тесты зелеными; теперь нам нужно фактически передать ему элементы из представления домашней страницы:

lists/views.py

```
def home_page(request):
    '''домашняя страница'''
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
```

[8] См. https://docs.djangoproject.com/en/1.11/topics/templates/

```
        return redirect('/')

    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})
```

А вот это действительно заставляет модульные тесты пройти успешно… Близок момент истины: пройдет ли функциональный тест?

```
$ python functional_tests.py
[...]
AssertionError: 'To-Do' не найдено в 'OperationalError в /'
```

Что ж, это очевидно. Раз так, давайте применим еще один метод отладки функционального теста, один из самых прямолинейных – посещение сайта в ручном режиме! Откройте браузер и направьте его на `http://localhost:8000`. Вы увидите страницу отладки Django с сообщением `no such table: lists_item` (Такой таблицы нет: элемент объекта `lists_item`), как на рис. 5.2.

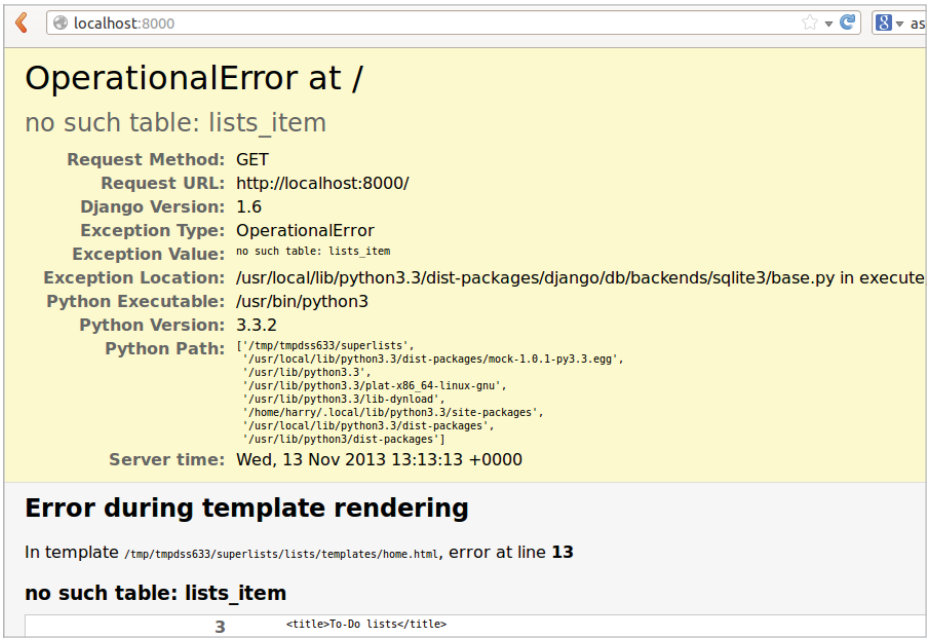


Рис. 5.2. Еще одно полезное отладочное сообщение

Создание производственной базы данных при помощи команды migrate

Это еще одно полезное сообщение об ошибке из Django, которое в сущности жалуется на то, что мы должным образом не настроили базу данных. Отсюда следует совершенно очевидный вопрос: тогда почему все хорошо

работало в модульных тестах? Все из-за того, что для модульных тестов Django создает специальную *тестовую базу данных*. Это одна из волшебных вещей, которые в Django делает класс `TestCase`.

Чтобы настроить нашу «реальную» базу данных, нужно ее создать. Базы данных SQLite – это просто файл на диске, и в *settings.py* вы увидите, что Django по умолчанию поместит ее в файл *db.sqlite3,* в основной каталог проекта:

uperlists/settings.py

```
[...]
# База данных
# https://docs.djangoproject.com/en/1.11/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Мы сообщили Django все, что нужно для создания базы данных: сначала через *models.py* и затем при создании файла миграций. Чтобы фактически использовать Django для получения реальной базы данных, мы применим еще один швейцарский нож Django – команду *manage.py* `migrate`:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, lists, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying lists.0001_initial... OK
  Applying lists.0002_item_text... OK
  Applying sessions.0001_initial... OK
```

Теперь можно обновить страницу на *localhost*, убедиться, что ошибка пропала, и снова попытаться выполнить функциональные тесты:

```
AssertionError: '2: Сделать мушку из павлиньих перьев' not found in ['1: Купить
павлиньи перья', '1: Сделать мушку из павлиньих перьев']
```

Уже близко! Нам нужно просто выправить нумерацию списка. Здесь поможет еще один потрясающий шаблонный тег Django – `forloop.counter`:

lists/templates/home.html

```
    {% for item in items %}
        <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
    {% endfor %}
```

Если вы попробуете еще раз, то увидите, что ФТ добрался до конца:

```
    self.fail('Закончить тест!')
AssertionError: Закончить тест!
```

Однако при его выполнении, вы, возможно, заметите что-то неладное, как показано на рис. 5.3.

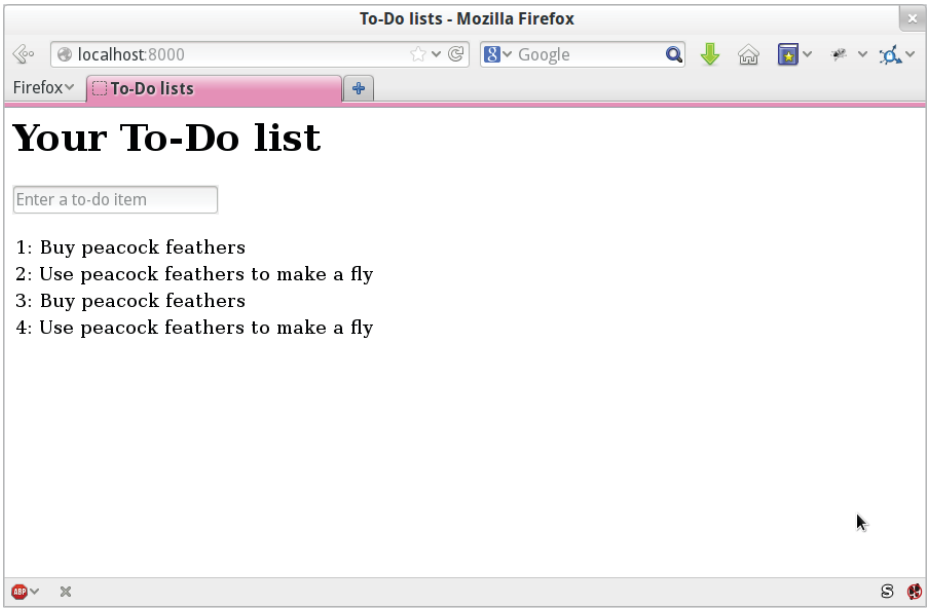


Рис. 5.3. Наличие элементов списка, перенесенных из последнего выполнения теста

Ну дела! Похоже, предыдущие запуски теста накапливают результаты в базе данных. По сути, если выполнить тесты еще раз, вы увидите, что ситуация только ухудшится:

```
1: Купить павлиньи перья
2: Сделать мушку из павлиньих перьев
3: Купить павлиньи перья
4: Сделать мушку из павлиньих перьев
5: Купить павлиньи перья
6: Сделать мушку из павлиньих перьев
```

Бр-р-р. А цель была так близка! Нужен какой-то автоматизированный способ подчистки после сеанса. А пока, если есть желание, можно сделать это вручную, удалив базу данных и воссоздав ее заново командой `migrate`:

```
$ rm db.sqlite3
$ python manage.py migrate --noinput
```

Подбодрите себя тем, что ФТ по-прежнему проходит.

За исключением этого небольшого дефекта в функциональном тесте, у нас все-таки есть программный код, который более-менее работает. Выполним фиксацию.

Начните с выполнения команд `git status` и `git diff`, и вы увидите изменения в *home.html*, *tests.py* и *views.py*. Давайте их добавим:

```
$ git add lists
$ git commit -m "Переадресация после POST и показ всех элементов в шаблоне"
```

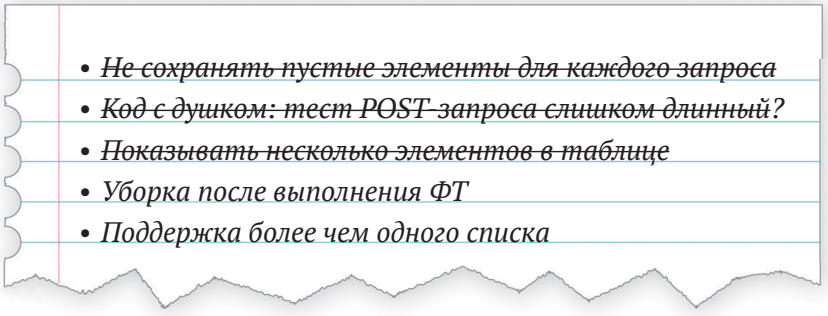
Вы можете найти для себя полезным добавлять маркеры для конца каждой главы, к примеру `git tag end-of-chapter-05`.

Резюме

К чему мы пришли?

- Настроили форму для добавления в список новых элементов используя POST-запрос.
- Настроили простую модель в базе данных для сохранения элементов списка.
- Узнали о создании миграций базы данных, в том числе тестовой базы данных (где миграции применяются автоматически) и реальной базы данных (где нужно применять миграции вручную).
- Использовали наши первые два шаблонных тега Django: `{% csrf_token %}` и цикл `{% for ... endfor %}`.
- Применили по крайней мере три разных метода отладки ФТ: встроенные операторы печати, оператор приостановки выполнения `time.sleep` и улучшение сообщений об ошибках.

Но в нашем списке неотложных дел осталось несколько пунктов: заставить ФТ подчищать после себя и (что, возможно, более важно) добавить поддержку более чем одного списка.



Стоит сказать, что мы *могли бы* уже предложить сайт заказчику, но люди сочтут странным, что всему народонаселению придется совместно использовать один-единственный список неотложных дел. Представляю себе, как это заставит людей погрузиться в размышления о том, насколько взаимосвязаны мы все друг с другом, как делим общую судьбу здесь, на космическом корабле «Земля», и как мы должны сотрудничать в решении глобальных проблем, с которыми сталкивается все человечество.

Но в практическом плане такой сайт будет не очень полезен.

Полезные понятия TDD

Регрессия

Когда новый программный код повреждает какой-нибудь аспект приложения, который раньше работал.

Неожиданная неполадка

Когда тест не срабатывает неожиданным образом. Это означает, что в наших тестах мы допустили ошибку либо что тесты помогли нам найти регрессию и мы должны что-то исправить в коде.

Красный/зеленый/рефакторизуй

Еще один способ описания процесса TDD. Написать тест и убедиться, что он не срабатывает (красный уровень), написать некий программный код, чтобы заставить его пройти успешно (зеленый уровень), затем выполнить перестройку кода, чтобы улучшить реализацию.

Триангуляция

Добавление контрольного случая с новым конкретным примером для существующего фрагмента кода, чтобы обосновать обобщение реализации (которое до этого момента, возможно, было получено «обманным путем»).

Если клюнуло трижды – рефакторизуй

Эмпирическое правило для ситуаций, когда из программного кода следует удалять дублирование. Если два фрагмента кода выглядят очень похожими, стоит подождать, пока вы не встретите третий случай, чтобы убедиться, какой фрагмент кода действительно является общим, повторно используемым фрагментом, который можно рефакторизовать.

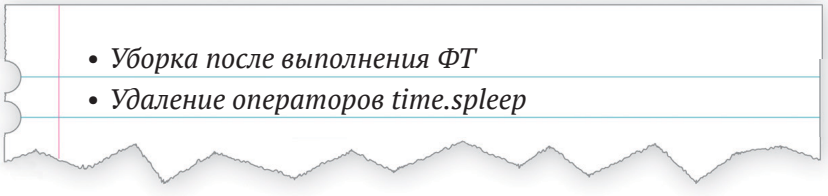
Блокнот с рабочим списком неотложных дел

Место для фиксации рабочих моментов, которые происходят с нами во время программирования, чтобы можно было закончить то, что мы делаем, и вернуться к ним позже.

Глава 6

Усовершенствование функциональных тестов: обеспечение изоляции и удаление методов sleep

Прежде чем мы возьмемся за исправление реальной проблемы, следует позаботиться о нескольких служебных операциях. В конце предыдущей главы мы отметили, что разные прогоны тестов влияют друг на друга, и мы это исправим. Также я недоволен операторами `time.sleep`, разбросанными по всему программному коду – они выглядят несколько ненаучно, поэтому заменим их на что-то более надежное.



Оба этих изменения приблизят нас к опробыванию «лучших приемов тестирования», которые сделают наши тесты детерминированнее и надежнее.

Обеспечение изоляции в функциональных тестах

Предыдущую главу мы завершили классической проблемой тестирования: как обеспечить изоляцию между тестами. Каждое выполнение функцио-

нальных тестов накапливало элементы списка в базе данных, и это влияло на результаты последующих тестов.

Когда мы запускаем *модульные* тесты, исполнитель тестов Django автоматически создает совершенно новую тестовую базу данных (отдельно от реальной), которую он может безопасно обнулять перед каждым запуском теста и отбрасывать в конце. Однако наши функциональные тесты выполняются относительно «реальной» базы данных, *db.sqlite3*.

Один из способов уладить этот вопрос состоит в том, чтобы – «собрать» собственное решение и добавить программный код в *functional_tests.py*, который выполнит очистку. Методы `setUp` и `tearDown` для такого рода задач просто идеальны.

Правда, начиная с Django 1.4 появился новый класс под названием `LiveServerTestCase`, который может сделать эту работу за вас. Он автоматически создает тестовую базу данных (точно так же, как в прогоне модульного теста) и запускает сервер разработки, относительно которого будут выполняться функциональные тесты. Несмотря на то что как инструмент он имеет некоторые ограничения, которые нам придется обойти позже, на данном этапе он невероятно полезен, поэтому давайте его задействуем.

Класс `LiveServerTestCase` ожидает, что он будет выполнен исполнителем тестов Django с использованием *manage.py*. Начиная с Django версии 1.6 исполнитель тестов отыскивает любые файлы, имя которых начинается с *test*. Чтобы поддержать чистоту и порядок, предлагаю создать папку для функциональных тестов, чтобы она выглядела как приложение. Для Django всего лишь нужно, чтобы это был допустимый каталог пакета Python (т. е. с файлом *__init__.py* внутри):

```
$ mkdir functional_tests
$ touch functional_tests/__init__.py
```

Затем мы *перемещаем* функциональные тесты из автономного файла *functional_tests.py* в файл *tests.py* приложения `functional_tests`. Используем команду `git mv`, чтобы Git отследил, что мы переместили файл:

```
$ git mv functional_tests.py functional_tests/tests.py
$ git status # показывает переименование functional_tests/tests.py и __init__.py
```

В этой точке ваше дерево каталогов должно выглядеть так:

```
.
├── db.sqlite3
├── functional_tests
│   ├── __init__.py
│   └── tests.py
├── lists
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   ├── 0002_item_text.py
│   │   ├── __init__.py
│   │   └── __pycache__
│   ├── models.py
│   ├── __pycache__
│   ├── templates
│   │   └── home.html
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── __pycache__
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Файл *functional_tests.py* исчез и превратился в *functional_tests/tests.py*. Теперь всякий раз, когда мы хотим выполнить функциональные тесты, вместо команды `python functional_tests.py` мы будем использовать `python manage.py test functional_tests`.

Вы можете смешать свои функциональные тесты с тестами для приложения `lists`. Но я все же предпочитаю их разделять, потому что функциональные тесты обычно имеют пересекающиеся компетенции, которые совпадают друг с другом в различных приложениях. ФТ предназначены для того, чтобы следить за происходящим с точки зрения пользователей. И пользователей не волнует, как вы разделили работу между различными приложениями!.

Теперь давайте отредактируем *functional_tests/tests.py* и изменим класс `NewVisitorTest`, чтобы он использовал класс Django `LiveServerTestCase`:

functional_tests/tests.py (ch06l001)

```
from django.test import LiveServerTestCase
from selenium import webdriver
```

```
from selenium.webdriver.common.keys import Keys
import time

class NewVisitorTest(LiveServerTestCase):
    '''тест нового посетителя'''

    def setUp(self):
        '''установка'''
        [...]
```

Далее вместо жесткого кодирования адресации localhost в порту 8000 `LiveServerTestCase` предоставляет нам атрибут `live_server_url`:

functional_tests/tests.py (ch06l002)

```
def test_can_start_a_list_and_retrieve_it_later(self):
    '''тест: можно начать список и получить его позже'''
    # Эдит слышала про крутое новое онлайн-приложение со списком
    # неотложных дел
    self.browser.get(self.live_server_url)
```

При желании мы можем удалить `if __name__ == '__main__'`, поскольку для запуска ФТ мы будем использовать исполнитель тестов Django.

Теперь мы в состоянии выполнить наши функциональные тесты при помощи исполнителя тестов Django, поручив ему выполнять тесты только для нашего нового приложения *functional_tests*:

```
$ python manage.py test functional_tests
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 65, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Закончить тест!')
AssertionError: Закончить тест!

 ----------------------------------------------------------------------
Ran 1 test in 6.578s

FAILED (failures=1)
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

ФТ доходит до `self.fail`, точно так же, как он это делал до рефакторизации. Вы также заметите, что, если выполнить тесты во второй раз, от предыдущего теста не останется никаких старых элементов списка – он подчистил после себя. Отлично! Мы должны зафиксировать это как атомарное изменение:

```
$ git status # functional_tests.py переименованный + модифицированный, новый __init__.py
$ git add functional_tests
$ git diff --staged -M
$ git commit # сообщение, напр."сделать functional_tests приложением,
использовать LiveServerTestCase"
```

Флаг `-M` в команде `git diff` бывает полезным. Он означает «обнаруживать перемещения», поэтому он заметит, что `functional_tests.py` и `functional_tests/tests.py` являются одним и тем же файлом, и покажет вам более осмысленную разницу `diff` (попробуйте выполнить ее без флага!).

Выполнение только модульных тестов

Теперь, если мы выполним `manage.py test`, Django выполнит и функциональный, и модульный тест:

```
$ python manage.py test
Creating test database for alias 'default'...
......F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later
[...]
AssertionError: Закончить тест!

 ----------------------------------------------------------------------
Ran 7 tests in 6.732s

FAILED (failures=1)
```

Чтобы выполнять только модульные тесты, можно указать, что мы хотим выполнить только тесты для приложения `lists`:

```
$ python manage.py test lists
Creating test database for alias 'default'...
......
 ----------------------------------------------------------
Ran 6 tests in 0.009s

OK
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

Полезные команды, обновлено

Выполнение функциональных тестов

```
python manage.py test functional_tests
```

Выполнение модульных тестов

```
python manage.py test lists
```

Что делать, если я говорю «выполнить тесты», а вы не уверены, какие тесты я имею в виду? Взгляните еще раз на блок-схему в конце главы 4 и попытайтесь понять, где мы находимся. В качестве эмпирического правила мы обычно выполняем функциональные тесты, только когда все модульные тесты проходят. Так что если есть сомнения – попробуйте оба!

Ремарка: обновление Selenium и Geckodriver

Сегодня я просматривал эту главу и обнаружил, что ФТ зависают при попытке их выполнить.

Оказывается, за ночь Firefox автообновился, и мои версии Selenium и Geckodriver тоже нуждались в обновлении. Беглый просмотр страницы с описанием релиза geckodriver[1] подтвердил, что вышла новая версия. Таким образом, накопилось несколько скачиваний и обновлений:

- Сначала быстрое обновление `pip install --upgrade selenium`.
- Затем недолгое скачивание нового драйвера geckodriver.
- Я сохранил резервную копию старой версии и поместил новую на ее место в системном пути PATH.
- Быстрая проверка командой `geckodriver --version` подтверждает, что новая версия готова к работе.

После этого ФТ снова заработали, как и ожидалось.

Нет никакой определенной причины, почему в книге это произошло именно в этой точке, и на самом деле маловероятно, что это произойдет именно сейчас и у вас тоже. Но в определенный момент это может случиться, поэтому показалось наиболее уместным упомянуть об этом именно тут, ведь как раз сейчас мы занимаемся служебными операциями.

Это один из тех моментов, с которыми придется смириться при использовании Selenium. Несмотря на то что версии вашего браузера и Selenium можно закрепить (на CI-сервере, например, то есть сервере под нужды непрерывной интеграции), в реальном мире версии браузеров не стоят на месте, и вам нельзя отставать от того, что есть у пользователей.

[1] См. https://github.com/mozilla/geckodriver/releases

Если с вашими ФТ происходит что-то странное, всегда стоит попробовать обновить Selenium..

А теперь вернемся к нашему обычному программированию.

О неявных и явных ожиданиях и методе time.sleep

Предлагаю поговорить о `time.sleep` в нашем ФТ:

functional_tests/tests.py

```
# Когда она нажимает enter, страница обновляется, и теперь страница
# содержит "1: Купить павлиньи перья" в качестве элемента таблицы списка
inputbox.send_keys(Keys.ENTER)
time.sleep(1)

self.check_for_row_in_list_table('1: Купить павлиньи перья')
```

Это называется явным ожиданием. Оно противопоставляется неявным ожиданиям – в некоторых случаях Selenium пробует ждать «автоматически» за вас, когда он считает, что страница загружается. Он даже предоставляет метод `implicitly_wait`, который дает вам возможность указать, сколько времени ему ждать, если вы запросите у него элемент, который, скорее всего, еще отсутствует на странице.

По сути дела, в первой редакции книги я мог целиком опираться на неявные ожидания. Но проблема в том, что неявные ожидания всегда немного капризны, и с выпуском Selenium 3 они стали еще более ненадежными. Вместе с тем, общим мнением команды разработчиков Selenium было то, что неявные ожидания – просто плохая идея и их надо избегать.

Поэтому данная редакция с самого начала содержит явно заданные ожидания. Но методы `time.sleep` имеют свои собственные проблемы. В настоящее время мы ожидаем в течение одной секунды, но кто сказал, что это правильный срок? Для большинства тестов, которые мы выполняем на нашей собственной машине, одна секунда – слишком долго и действительно замедляет выполнение наших ФТ. 0,1 секунды – было бы в самый раз. Но сложность заключается в том, что, если вы установите слишком короткий срок, то периодически будете получать побочную неполадку по любой причине, из-за которой ноутбук именно в тот момент работал медленнее. И даже с 1 секундой вы никогда не можете быть полностью уверены, что не получите случайные неполадки, которые не будут говорить о настоящей проблеме, а ложно-положительные исходы в тестах – реальный

источник раздражения. (Гораздо более подробно этот вопрос обсуждается в статье Мартина Фаулера[2]).

Неожиданные ошибки `NoSuchElementException` и `StaleElementException` являются обычными признаками того, что вы забыли про явные ожидания. Попробуйте удалить `time.sleep` и убедиться, что получите одну такую ошибку.

Поэтому давайте заменим наши методы `sleep` инструментом, который будет ожидать столько, сколько нужно, вплоть до длительного тайм-аута, достаточного для того, чтобы отловить любые незначительные сбои. Мы переименуем функцию `check_for_row_in_list_table` в `wait_for_row_in_list_table` и добавим в нее операции, связанные с опросом/повторной попыткой:

functional_tests/tests.py (ch06l004)

```
from selenium.common.exceptions import WebDriverException

MAX_WAIT = 10  ❶
[...]

    def wait_for_row_in_list_table(self, row_text):
        '''ожидать строку в таблице списка'''
        start_time = time.time()
        while True:  ❷
            try:
                table = self.browser.find_element_by_id('id_list_table')  ❸
                rows = table.find_elements_by_tag_name('tr')
                self.assertIn(row_text, [row.text for row in rows])
                return  ❹
            except (AssertionError, WebDriverException) as e:  ❺
                if time.time() - start_time > MAX_WAIT:  ❻
                    raise e  ❻
                time.sleep(0.5)  ❺
```

❶ Мы будем использовать константу под названием `MAX_WAIT` с установленным максимальным количеством времени, которое мы готовы ожидать. 10 секунд более чем достаточно для отлавливания любых незначительных сбоев или случайных замедлений.

❷ Это цикл, который будет продолжаться бесконечно, до тех пор, пока мы не придем к одному из двух возможных выходов.

❸ Здесь находятся наши три строки тестирующих утверждений из старой версии метода.

[2] См. https://martinfowler.com/articles/nonDeterminism.html

❹ Если мы и наше тестирующее утверждение их проходит, мы возвращаемся из функции и выходим из цикла.

❺ Но если мы отлавливаем исключение, то ожидаем в течение короткого периода и в цикле делаем повторную попытку. Мы хотим отловить два типа исключений: `WebDriverException` для случая, когда страница не загрузилась и Selenium не может найти табличный элемент на странице, и `AssertionError` – для случая, когда таблица имеется, но это, возможно, таблица до перезагрузок страницы, поэтому в ней пока нет нашей строки.

❻ Это второй путь выхода из цикла. Если мы доходим до этой точки, значит, наш программный код продолжил поднимать исключения всякий раз, когда мы делали попытку, пока мы не превысили тайм-аут. Тем самым на этот раз мы повторно поднимаем исключение и даем ему «всплыть» в нашем тесте. В итоге, скорее всего, мы получим его в отчете об обратной трассировке, сообщающем, почему тест не сработал.

Вам кажется, что этот программный код немного уродлив и затрудняет понимание процесса? Соглашусь. Позже мы перестроим обобщенную вспомогательную функцию `wait_for`, чтобы отделить последовательность операций по хронометражу и повторному вызову исключений от тестирующих утверждений. Но во многих случаях мы будем ожидать, пока нам это нужно.

Если раньше вы уже использовали Selenium, то знаете, что в нем есть несколько вспомогательных функций, которые выполняют ожидания. Я не большой их поклонник. На протяжении этой книги мы создадим пару ожидающих вспомогательных инструментов, которые, полагаю, будут содействовать хорошо читаемому программному коду. Но, разумеется, в свободное время вам следует ознакомиться со свойственными Selenium вариантами ожиданий и составить о них собственное мнение.

Теперь мы можем переименовать вызовы методов и удалить методы `time.sleeps`:

functional_tests/tests.py (ch06l005)

```
[...]
# Когда она нажимает enter, страница обновляется, и теперь страница
# содержит "1: Купить павлиньи перья" в качестве элемента таблицы списка
inputbox.send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Купить павлиньи перья')

# Текствое поле по-прежнему приглашает ее добавить еще один элемент.
```

```
# Она вводит "Сделать мушку из павлиньих перьев" (Эдит очень методична)
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Сделать мушку из павлиньих перьев')
inputbox.send_keys(Keys.ENTER)

# Страница снова обновляется, и теперь показывает оба элемента
# ее списка
self.wait_for_row_in_list_table('2: Сделать мушку из павлиньих перьев')
self.wait_for_row_in_list_table('1: Купить павлиньи перья')
[...]
```

И повторно выполнить тесты:

```
$ python manage.py test
Creating test database for alias 'default'...
......F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 73, in
test_can_start_a_list_and_retrieve_it_later
    self.fail('Закончить тест!')
AssertionError: Закончить тест!

 ----------------------------------------------------------------------
Ran 7 tests in 4.552s

FAILED (failures=1)
System check identified no issues (0 silenced).
Destroying test database for alias 'default'...
```

Мы добираемся до этого же места и замечаем, что также сбросили несколько секунд с времени исполнения. Прямо сейчас может показаться, что это не слишком много, но все это суммируется.

Просто для проверки, что мы делаем все верно, давайте преднамеренно нарушим тест несколькими способами и посмотрим на некоторые ошибки. Сначала проверим, что мы получим нужную ошибку, если будем отыскивать текст, который никогда не появится:

functional_tests/tests.py (ch06l006)

```
        rows = table.find_elements_by_tag_name('tr')
        self.assertIn('foo', [row.text for row in rows])
        return
```

Мы видим, что по-прежнему получаем хорошо говорящее само за себя тестовое сообщение о неполадке:

```
    self.assertIn('foo', [row.text for row in rows])
AssertionError: 'foo' not found in ['1: Купить павлиньи перья']
```

Давайте вернем программный код в прежний вид и нарушим кое-что еще:

functional_tests/tests.py (ch06l007)

```
    try:
        table = self.browser.find_element_by_id('id_nothing')
        rows = table.find_elements_by_tag_name('tr')
        self.assertIn(row_text, [row.text for row in rows])
        return
    [...]
```

Разумеется, мы тоже будем получать ошибки, когда страница не содержит элемент, который мы ищем.

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_nothing"]
```

Похоже, все в порядке. Теперь вернем программный код в тот вид, который должен быть, и сделаем завершающий прогон теста.

```
$ python manage.py test
[...]
AssertionError: Закончить тест!
```

Великолепно! Закончив эту небольшую интермедию, продолжим работу над приведением приложения в то состояние, когда оно фактически сможет работать с многочисленными списками.

Тестирование лучших практических приемов, примененных в этой главе

Обеспечение изоляции тестов и управление глобальным состоянием

Разные тесты не должны влиять друг на друга. Для этого нужно обнулять любое постоянное состояние в конце каждого теста. Исполнитель тестов Django помогает это делать путем создания тестовой базы данных, которая полностью очищается в промежутках между тестами. (См. также главу 23.)

Не допускать методов sleep

Всякий раз, когда мы ожидаем загрузки страниц, рука тянется использовать черновой вариант с `time.sleep`. Но проблема в том, что время ожидания всегда похоже на выстрел в темноту: либо слишком короткий и уязвимый для побочных неполадок, либо слишком долгий и замедляющий

прогоны тестов. Отдавайте предпочтение циклу с повторной попыткой, который опрашивает приложение и как можно скорее идет дальше.

Не опираться на неявные ожидания в Selenium

Теоретически Selenium выполняет неявные ожидания, но их реализация варьируется между браузерами и на момент написания настоящей книги была очень ненадежной в драйвере Selenium 3 для Firefox. «Явное ожидание лучше, чем неявное», как утверждает Дзэн Python, поэтому отдавайте предпочтение явному ожиданию.

Глава 7

Работа в инкрементном режиме

Сейчас предлагаю обратиться к реальной проблеме, заключающейся в том, что наша структура кода допускает только один глобальный список. В этой главе я продемонстрирую критический метод TDD: как адаптировать существующий программный код, используя инкрементный, пошаговый процесс, который проводит вас из одного рабочего состояния в другое. Билли-тестировщик – это вовсе не Том-рефакторщик.

Маломасштабные конструктивные изменения по мере необходимости

Давайте подумаем о том, как должна работать поддержка многочисленных списков. В настоящее время ФТ (который мы максимально приблизили к проектному заданию) говорит следующее:

functional_tests/tests.py

```
# Эдит интересно, запомнит ли сайт ее список. Далее она видит, что
# сайт сгенерировал для нее уникальный URL-адрес – по этому поводу
# выводится небольшой текст с объяснениями.
self.fail('Закончить тест!')

# Она посещает этот URL-адрес – ее список по-прежнему там.

# Удовлетворенная, она снова ложится спать.
```

Но в действительности здесь мы хотим дать более развернутую информацию, отметив, что разные пользователи не видят списки друг друга и каждый получает свой собственный URL-адрес как способ вернуться к своим сохраненным спискам. Как бы могла выглядеть новая структура кода?

В первую очередь маломасштабные конструктивные изменения

Методология TDD тесно связана с гибкой динамикой разработки программного обеспечения, а та, в свою очередь, с резко отрицательной реакцией на *крупномасштабные конструктивные изменения в первую очередь*, то есть на традиционную практику разработки программного обеспечения, в силу которой после продолжительного сбора технических требований наступает одинаково продолжительная стадия разработки структуры ПО, когда оно планируется на бумаге. Гибкая философия заключается в том, что вы получаете больше знаний в результате решения практических задач, чем при изучении теории, особенно если вы как можно раньше сталкиваете свое приложение с реальными пользователями. Вместо долгой первичной стадии выработки структуры кода мы пытаемся выпустить *минимально дееспособное приложение* на ранней стадии и даем структуре развиваться постепенно, основываясь на обратной связи от использования в реальной ситуации.

Но это не означает, что такой взгляд на конструирование кода запрещается с порога! В предыдущей большой главе мы увидели, как бездумное забегание вперед способно *в конечном счете* вывести нас к верному ответу, но зачастую небольшие размышления о структуре кода помогают получить его быстрее. Поэтому давайте подумаем о минимальном дееспособном приложении `lists` – какую структуру кода нам нужно реализовать.

- Мы хотим, чтобы каждый пользователь мог сохранять свой собственный список – как минимум один на данном этапе.
- Список состоит из нескольких элементов, где главный атрибут – небольшой описательный текст.
- Мы должны сохранять списки от одного посещения к другому. На данный момент мы можем дать каждому пользователю уникальный URL-адрес своего списка. Позже нам может понадобиться некий способ автоматического распознавания пользователей и показа им их списков.

Чтобы предоставлять пользователю текущие элементы, по всей видимости, мы будем сохранять списки и их элементы в базе данных. Каждый список будет иметь уникальный URL, а каждый элемент списка будет выражен небольшим описанием, связанным с определенным списком.

Вам это никогда не понадобится!

Как только вы начинаете думать о структуре кода, такие размышления трудно остановить. Нас начинают обуревать самые разнообразные мыс-

ли: можно дать каждому списку имя или заголовок, можно распознавать пользователей по имени и паролю, можно добавить в список более длинное поле для примечаний, а также краткие описания, можно предоставить некоторое упорядочивание и т. д. Но мы повинуемся другому принципу гибкого евангелия: «YAGNI» (произносится «ягни»), которое расшифровывается так: You Ain't Gonna Need It! То есть «вам это никогда не понадобится».

Как разработчики программного обеспечения, мы получаем удовольствие от создания, и иногда трудно сопротивляться порыву создавать просто потому, что нас посетила идея и она нам, *возможно*, пригодится. Проблема в том, что, как правило (независимо от того, насколько классной была идея), вы *не будете* ее использовать. Наоборот, у вас останется куча неиспользуемого кода, усложняющего ваше приложение. YAGNI – это мантра, которую мы используем, чтобы противостоять своим сверхвосторженным творческим порывам.

REST (-овский) подход

У нас есть представление о структуре данных, которую мы хотим реализовать, – элемент «Модель» шаблона проектирования «Модель – Представление – Контроллер» (MVC). А как быть с элементами «Представление» и «Контроллер»? Как пользователь должен взаимодействовать со списками `List` и их элементами `Item` через веб-браузер?

Передача состояния представления (REST, от англ. *Representational State Transfer*) – это подход к разработке веб-приложений, который обычно используется для того, чтобы руководить проектированием прикладных программных интерфейсов для веб или веб-API. При разработке сайта, ориентированного на пользователя, невозможно *строго* следовать правилам REST, но они все же обеспечивают некоторое полезное вдохновение (обратитесь к приложению F, если хотите увидеть реальный REST API).

Технология REST предполагает, что у нас есть структура URL, которая соответствует нашей структуре данных, в данном случае – спискам и элементам списков. Каждый список может иметь собственный URL:

```
lists/<list identifier>/
```

Таким образом будет выполнено техническое требование, которое мы определили в ФТ. Для просмотра списка мы используем GET-запрос (типичное посещение браузером страницы).

Для создания совершенно нового списка у нас будет специальный URL, который принимает POST-запросы:

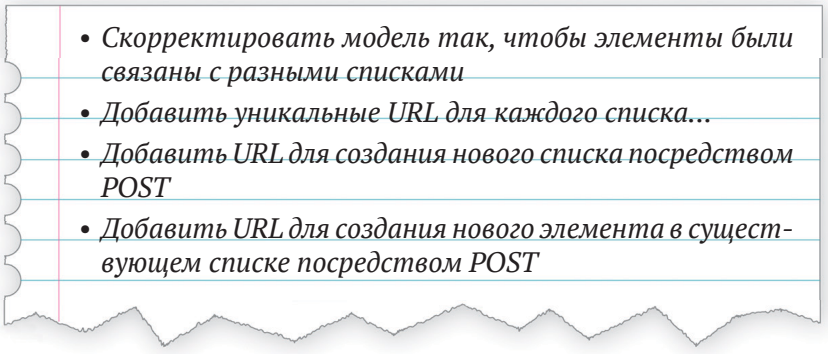
```
/lists/new
```

Для добавления нового элемента к существующему списку будет отдельный URL, на который мы можем отправлять POST-запросы:

```
/lists/<list identifier>/add_item
```

(Опять-таки, мы не пытаемся идеально следовать правилам REST, который здесь использовал бы PUT-запрос, – мы просто используем REST для вдохновения. Помимо всего прочего, вы не можете использовать PUT в стандартной HTML-форме.)

После подведения итогов наш блокнот для этой главы выглядит примерно так:



Инкрементная реализация новой структуры кода на основе TDD

Как мы используем методологию TDD для реализации новой структуры кода? Давайте еще раз посмотрим на блок-схему процесса TDD на рис. 7.1.

На верхнем уровне мы будем использовать комбинацию, состоящую из добавления новой функциональности (путем добавления нового ФТ и написания нового прикладного кода) и рефакторизации приложения, то есть написания части существующей реализации заново так, чтобы это обеспечило пользователю ту же функциональность, но с учетом особенностей новой структуры кода. Мы сможем применять существующий функциональный тест для проверки на отсутствие каких-то повреждений того, что уже работает, и новый функциональный тест для управления новыми возможностями.

На уровне модульных тестов мы будем добавлять новые тесты или модифицировать существующие для проверки требуемых изменений и сможем аналогично использовать модульные тесты, которых мы не касаемся, чтобы проверить, что по ходу мы ничего не повредили.

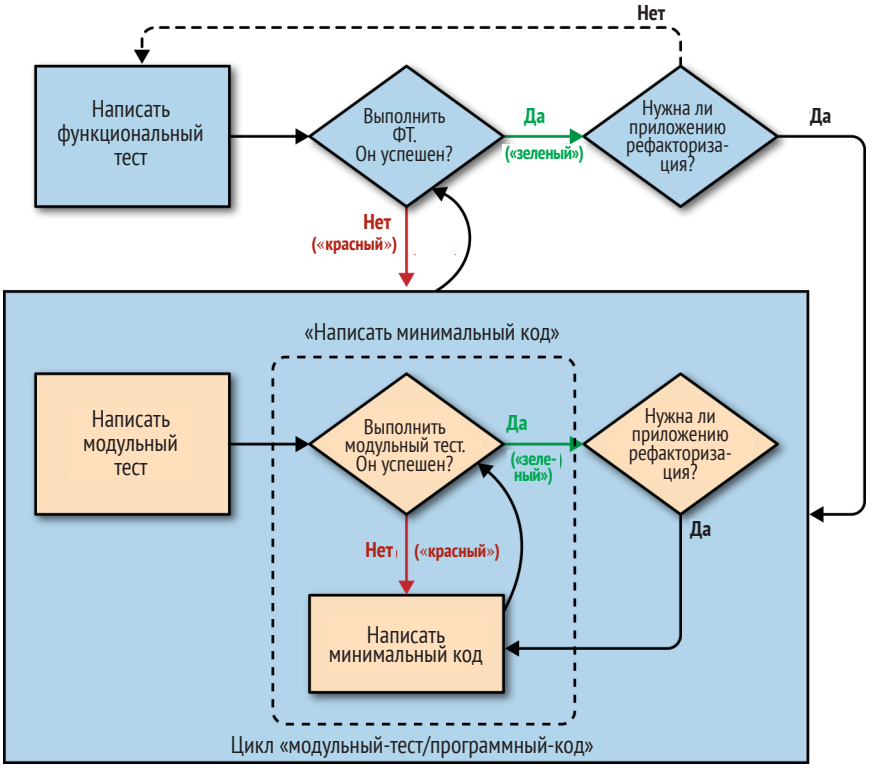


Рис. 7-1. Процесс TDD обрабатывает с функциональными и модульными тестами

Обеспечение теста на наличие регрессии

Давайте преобразуем содержимое нашего блокнота в новый метод функционального теста, который вводит второго пользователя и проверяет, что его список неотложных дел не связан со списком Эдит.

Мы начинаем почти так же, как и с первым: Эдит добавляет первый элемент, чтобы создать список неотложных дел. Однако тут мы вводим первое новое утверждение: список Эдит должен находиться по своему собственному уникальному URL-адресу:

functional_tests/tests.py (ch07l005)

```
def test_can_start_a_list_for_one_user(self):
    '''тест: можно начать список для одного пользователя'''
    # Эдит слышала про крутое новое онлайн-приложение со списком
    [...]
    # Страница снова обновляется и теперь показывает оба элемента ее списка
    self.wait_for_row_in_list_table('2: Сделать мушку из павлиньих перьев')
```

```
    self.wait_for_row_in_list_table('1: Купить павлиньи перья')

    # Удовлетворенная, она снова ложится спать.

def test_multiple_users_can_start_lists_at_different_urls(self):
    '''тест: многочисленные пользователи могут начать списки по разным url'''
    # Эдит начинает новый список
    self.browser.get(self.live_server_url)
    inputbox = self.browser.find_element_by_id('id_new_item')
    inputbox.send_keys('Купить павлиньи перья')
    inputbox.send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table('1: Купить павлиньи перья')

    # Она замечает, что ее список имеет уникальный URL-адрес
    edith_list_url = self.browser.current_url
    self.assertRegex(edith_list_url, '/lists/.+')  ❶
```

❶ `assertRegex` – это вспомогательная функция из `unittest`, которая проверяет, соответствует ли строка регулярному выражению. Мы используем ее, чтобы проверить реализацию нашей новой REST'овской конструкции. Дополнительную информацию можно почерпнуть в документации по `unittest`[1].

Далее мы воображаем, что приходит новый пользователь. Мы хотим проверить, что при посещении домашней страницы он не видит ни одного элемента списка Эдит и что он получает уникальный URL-адрес для своего списка.

functional_tests/tests.py (ch07l006)

```
[...]
self.assertRegex(edith_list_url, '/lists/.+')  ❶

# Теперь новый пользователь, Фрэнсис, приходит на сайт.

## Мы используем новый сеанс браузера, тем самым обеспечивая, чтобы никакая
## информация от Эдит не прошла через данные cookie и пр.
self.browser.quit()
self.browser = webdriver.Firefox()

# Фрэнсис посещает домашнюю страницу. Нет никаких признаков списка Эдит
self.browser.get(self.live_server_url)
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Купить павлиньи перья', page_text)
```

[1] См. http://docs.python.org/3/library/unittest.html

```
self.assertNotIn('Сделать мушку', page_text)

# Фрэнсис начинает новый список, вводя новый элемент. Он менее
# интересен, чем список Эдит...
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Купить молоко')
inputbox.send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Купить молоко')

# Фрэнсис получает уникальный URL-адрес
francis_list_url = self.browser.current_url
self.assertRegex(francis_list_url, '/lists/.+')
self.assertNotEqual(francis_list_url, edith_list_url)

# Опять-таки, нет ни следа от списка Эдит
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Купить павлиньи перья', page_text)
self.assertIn('Купить молоко', page_text)

# Удовлетворенные, они оба ложатся спать
```

❶ Я использую форму записи с двойными хешами (##), чтобы обозначить *метакомментарии* (комментарии о том, как работает тест и почему), чтобы мы могли отличать их от регулярных комментариев в ФТ, которые объясняют историю пользователя. Они являются посланием к нам самим в будущем, чтобы потом не удивляться, с какой это стати мы выходим из браузера и запускаем новый...

В остальном новый тест довольно хорошо говорит сам за себя. Давайте посмотрим, что получится, когда мы выполним наши ФТ:

```
$ python manage.py test functional_tests
[...]
.F
======================================================================
FAIL: test_multiple_users_can_start_lists_at_different_urls
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File “/.../superlists/functional_tests/tests.py”, line 83, in
test_multiple_users_can_start_lists_at_different_urls
    self.assertRegex(edith_list_url, '/lists/.+')
AssertionError: Regex didn't match: '/lists/.+' not found in
'http://localhost:8081/'

 ----------------------------------------------------------------------
```

```
Ran 2 tests in 5.786s

FAILED (failures=1)
```

Хорошо, наш первый тест по-прежнему проходит, и второй не срабатывает там, где мы и ожидали. Выполним фиксацию, а затем создадим несколько новых моделей и представлений:

```
$ git commit -a
```

Итеративное движение в сторону новой структуры кода

Воодушевленный новой структурой кода, в этой точке я испытывал сильный порыв взять и начать изменять *models.py*, что повредило бы половину модульных тестов, и затем вломиться и изменить почти каждую строку кода за один присест. Это естественное желание, и методология TDD как дисциплина постоянно с ним борется. Повинуйтесь Билли-тестировщику, а не Тому-рефакторщику! От нас никто не требует реализовывать новую, сверкающую глянцем структуру кода революционными методами. Давайте вносить небольшие изменения, которые переводят нас из одного рабочего состояния в другое, где наша структура мягко направляет нас на каждом этапе.

В нашем рабочем списке неотложных дел имеется четыре элемента. ФТ со своим сообщением `Regexp didn't match` (регулярное выражение не совпало) говорит, что второй элемент – назначение спискам их собственного URL-адреса и идентификатора – будет темой нашей работы на следующем этапе. Давайте попробуем исправить эту ошибку, и только эту, ничего другого.

URL-адрес получается из переадресации после POST-запроса. В *lists/tests.py* найдите `test_redirects_after_POST` и замените ожидаемое место переадресации:

lists/tests.py

```
self.assertEqual(response.status_code, 302)
self.assertEqual(response['location'], '/lists/единственный-в своем-роде-
список-в-мире/')
```

Не кажется ли это немного странным? Безусловно, */lists/один-единственный-список-в-мире* не является URL-адресом, который появится в окончательном варианте нашего приложения. Но мы стараемся вносить по одному изменению за раз. Пока наше приложение поддерживает всего один список, это единственный URL-адрес, который имеет смысл. Мы по-прежнему движемся вперед в том, что у нас будет разный URL-адрес для списка

и домашней страницы, что является шагом по пути к более REST-образной конструкции. Позже, когда у нас будут многочисленные списки, вносить изменения будет просто.

На это можно посмотреть и по-другому, как на технику принятия решений: наша новая структура URL-адресов в настоящее время не реализована поэтому она работает для 0 элементов. В итоге мы хотим решить задачу для *n* элементов, но решение для 1 элемента является хорошим шагом в этом направлении.

Выполнение модульных тестов дает ожидаемую неполадку:

```
$ python manage.py test lists
[...]
AssertionError: '/' != '/lists/один-единственный-список-в-мире/'
```

Мы можем скорректировать представление `home_page` в *lists/views.py*:

lists/views.py

```
def home_page(request):
    '''домашняя страница'''
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/lists/один-единственный-список-в-мире/')

    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})
```

Конечно же, это полностью повредит функциональные тесты, потому что такого URL-адреса на нашем сайте еще нет. Разумеется, если вы их выполните, вы обнаружите, что они не срабатывают сразу после попытки передать первый элемент, сообщая, что они не могут найти таблицу списка; все потому, что URL-адрес */один-единственный-список-в-мире/* еще не существует!

```
  File "/.../superlists/functional_tests/tests.py", line 57, in
test_can_start_a_list_for_one_user
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_list_table"]

[...]

  File "/.../superlists/functional_tests/tests.py", line 79, in
test_multiple_users_can_start_lists_at_different_urls
```

```
    self.wait_for_row_in_list_table('1: Купить павлиньи перья')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_list_table"]
```

Мало того что новый тест не срабатывает, но ведь и старый тоже! Это говорит о том, что мы внесли *регрессию*. Попробуем как можно быстрее вернуться в рабочее состояние, создав URL-адрес для нашего одного-единственного списка.

Первый самодостаточный шаг: один новый URL-адрес

Откройте *lists/tests.py* и добавьте новый тестовый класс под названием `ListViewTest`. Затем скопируйте метод `test_displays_all_list_items` напротив `HomePageTest` в новый класс, переименуйте его и немного подправьте:

lists/tests.py (ch07l009)

```
class ListViewTest(TestCase):
    '''тест представления списка'''

    def test_displays_all_items(self):
        '''тест: отображаются все элементы списка'''
        Item.objects.create(text='itemey 1')
        Item.objects.create(text='itemey 2')

        response = self.client.get('/lists/один-единственный-список-в-мире/')

        self.assertContains(response, 'itemey 1')  ❶
        self.assertContains(response, 'itemey 2')  ❶
```

❶ Это новый вспомогательный метод: вместо слегка раздражающей свистопляски с `assertIn/response.content.decode()`, Django предоставляет метод `assertContains`, который знает, как иметь дело с откликами и байтами их содержимого.

Теперь попытаемся выполнить этот тест:

```
    self.assertContains(response, 'itemey 1')
[...]
AssertionError: 404 != 200 : Невозможно получить содержимое: Код отклика был 404
```

Мы видим хороший побочный эффект от использования `assertContains`: эта ошибка сразу же сообщает, что тест не проходит, потому что новый URL-адрес еще не существует, и возвращает код состояния 404.

Новый URL-адрес

URL-адрес одноэлементного списка еще не существует. Мы исправляем это в *superlists/urls.py*.

Будьте аккуратны с закрывающими косыми чертами в URL-адресах – как здесь в тестах, так и в *urls.py*. Это частая причина дефектов.

superlists/urls.py

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/один-единственный-список-в-мире/$', views.view_list,
name='view_list'),
]
```

Выполнив тест снова, мы получаем:

```
AttributeError: module 'lists.views' has no attribute 'view_list'
```

Новая функция представления

Прекрасно и само говорит за себя. Давайте создадим фиктивную функцию представления в *lists/views.py*:

lists/views.py

```
def view_list(request):
    '''представление списка'''
    pass
```

Сейчас мы получаем:

```
ValueError: Представление lists.views.view_list не вернуло объект
HttpResponse. Вместо этого оно вернуло None.

[...]
FAILED (errors=1)
```

Теперь всего одна неполадка, и она указывает нам нужное направление. Давайте скопируем две последние строки из представления `home_page` и посмотрим, удастся ли этот трюк:

lists/views.py

```
def view_list(request):
    '''представление списка'''
```

```
    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})
```

Выполните модульные тесты повторно, они должны пройти:

```
Ran 7 tests in 0.016s

OK
```

Теперь снова попробуем функциональные тесты и посмотрим, что они нам скажут:

```
FAIL: test_can_start_a_list_for_one_user
[...]
  File "/.../superlists/functional_tests/tests.py", line 67, in
test_can_start_a_list_for_one_user
[...]
AssertionError: '2: Сделать мушку из павлиньих перьев' not found in ['1:
Купить павлиньи перья']

FAIL: test_multiple_users_can_start_lists_at_different_urls
[...]
AssertionError: Купить павлиньи перья' unexpectedly found in 'Your To-Do
list\n1: Купить павлиньи перья'
[...]
```

Оба продвигаются чуть дальше, чем были прежде, но по-прежнему не срабатывают. Было бы хорошо вернуть тесты в рабочее состояние и заставить первый из них пройти успешно. Что он пытается нам сказать?

Ошибка возникает, когда мы пробуем добавить второй элемент. Здесь нужно засучить рукава и приступить к отладке. Мы знаем, что домашняя страница работает, потому что тест прошел весь путь вниз до строки 67 в ФТ, так что мы как минимум добавили первый элемент. И все наши модульные тесты проходят, поэтому мы вполне уверены, что URL-адреса и представления делают то, что должны делать – домашняя страница отображает правильный шаблон и может обрабатывать POST-запросы, а представление для *одного-единственного-списка-в-мире* знает, как отображать все элементы… Но оно не знает, как обрабатывать POST-запросы. Ага, это дает нам ключ к разгадке.

Второй подсказкой является эмпирическое правило, которое гласит: когда все модульные тесты проходят, а функциональные тесты – нет, нередко это говорит о проблеме, которая не была охвачена модульными тестами. В нашем случае эта проблема часто связана с шаблоном.

Разгадка в том, что наша форма ввода в *home.html* в настоящее время не задает явный URL-адрес, куда делать POST-запрос.

lists/templates/home.html

```
<form method="POST">
```

По умолчанию браузер передает POST-данные обратно на тот же URL-адрес, где он в настоящее время находится. Когда мы находимся по адресу домашней страницы, все работает хорошо, но когда мы на нашей странице *одного-единственного-списка-в-мире*, это не срабатывает.

Теперь мы можем добавить в наше новое представление обработку POST-запроса, но это будет сопряжено с написанием еще большего числа тестов и программного кода, а мы хотим как можно скорее вернуться в рабочее состояние. По сути дела, самое быстрое, что можно сделать для наладки, – просто использовать существующее представление домашней страницы, которое уже работает для всех POST-запросов:

lists/templates/home.html

```
<form method="POST" action="/">
```

Попробуйте, и мы увидим, что наши ФТ возвращаются в более удачное место:

```
FAIL: test_multiple_users_can_start_lists_at_different_urls
[...]
AssertionError: 'Купить павлиньи перья' unexpectedly found in 'Your To-Do
list\n1: Купить павлиньи перья'

Ran 2 tests in 8.541s

FAILED (failures=1)
```

Наш первоначальный тест проходит еще раз, значит, мы вернулись в рабочее состояние. Новая функциональность может еще не работать, но по крайней мере старый материал работает так же хорошо, как и прежде.

Зеленый? Рефакторизуй!

Пора навести небольшой порядок.

В свистопляске *Красный/Зеленый/Рефакторизация* мы дошли до зеленого, так что нужно глянуть, что нужно, чтобы выполнить рефакторизацию. Сейчас у нас есть два представления: одно для домашней страницы, другое для отдельного списка. Оба используют один тот же шаблон и передают ему все элементы списка, находящиеся в базе данных. Если мы просмотрим наши методы модульного тестирования, то увидим какие-то фрагменты, которые мы, вероятно, захотим изменить:

```
$ grep -E "class|def" lists/tests.py
class HomePageTest(TestCase):
    def test_uses_home_template(self):
    def test_displays_all_list_items(self):
    def test_can_save_a_POST_request(self):
    def test_redirects_after_POST(self):
    def test_only_saves_items_when_necessary(self):
class ListViewTest(TestCase):
    def test_displays_all_items(self):
class ItemModelTest(TestCase):
    def test_saving_and_retrieving_items(self):
```

Мы определенно можем удалить метод `test_displays_all_list_items` из `HomePageTest`, он больше не нужен. Если теперь выполнить команду **`manage.py test lists`**, она сообщит, что было выполнено 6 тестов вместо 7:

```
Ran 6 tests in 0.016s
OK
```

Раз нам больше не нужно, чтобы шаблон домашней страницы отображал все элементы списка, он должен просто показывать единственное поле ввода, приглашающее начать новый список.

Еще один шажок: отдельный шаблон для просмотра списков

Поскольку домашняя страница и представление списка теперь являются отдельными страницами, они должны использовать разные HTML-шаблоны; *home.html* будет иметь единственное поле ввода, тогда как новый шаблон, *list.html*, позаботится о показе таблицы существующих элементов.

Давайте добавим новый тест, чтобы проверить, что он использует другой шаблон:

lists/tests.py

```
class ListViewTest(TestCase):
    '''тест представления списка'''

    def test_uses_list_template(self):
        '''тест: используется шаблон списка'''
        response = self.client.get('/lists/единственный-список-в-мире/')
        self.assertTemplateUsed(response, 'list.html')

    def test_displays_all_items(self):
        '''тест: отображаются все элементы списка'''
        [...]
```

`assertTemplateUsed` – одна из наиболее полезных функций тестового клиента Django. Посмотрим, что она говорит:

```
AssertionError: False is not true : Шаблон 'list.html' не являлся шаблоном,
который использовался для вывода отклика в качестве HTML. Фактически
использовался шаблон(ы): home.html
```

Великолепно! Теперь изменим представление:

lists/views.py

```
def view_list(request):
    '''новый список'''
    items = Item.objects.all()
    return render(request, 'list.html', {'items': items})
```

Но, очевидно, этот шаблон еще не существует. Если мы выполним модульные тесты, то получим:

```
django.template.exceptions.TemplateDoesNotExist: list.html
```

Давайте создадим новый файл в *lists/templates/list.html*:

```
$ touch lists/templates/list.html
```

Пустой шаблон, который дает нам следующую ниже ошибку – приятно осознавать, что есть тесты, которые проверят его наполнение:

```
AssertionError: False is not true : Невозможно найти 'itemey 1' в отклике
```

Шаблон для отдельного списка будет использовать повторно многое из того, что сейчас есть в *home.html*, поэтому для начала просто копируем весь материал:

```
$ cp lists/templates/home.html lists/templates/list.html
```

Это снова приводит к тому, что тесты проходят (зеленый уровень). Теперь наведем небольшой порядок (рефакторизация). Мы сказали, что домашней странице не нужно перечислять элементы списка – ей нужно только поле ввода для нового списка, поэтому мы можем удалить несколько строк из *lists/templates/home.html* и, может быть, немного подправить `h1` – скажем, назначив заголовок «Start a new To-Do list» («Начать новый список неотложных дел»):

lists/templates/home.html

```
<body>
  <h1>Start a new To-Do list</h1>
```

```
  <form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
    {% csrf_token %}
  </form>
</body>
```

Повторно выполняем модульные тесты, чтобы проверить, что ничего не испортили – хорошо…

На самом деле вовсе не нужно передавать все элементы в шаблон *home.html* в представлении `home_page`, поэтому можно это упростить:

lists/views.py

```
def home_page(request):
    '''домашняя страница'''
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/lists/единственный-список-в-мире/')
    return render(request, 'home.html')
```

Выполните модульные тесты еще раз; они по-прежнему проходят. Пора приступать к выполнению функциональных тестов:

```
AssertionError: '1: Купить молоко' not found in ['1: Купить павлиньи перья', '2:
Купить молоко']
```

Неплохо! Тест на наличие регрессии (первый ФТ) проходит, и новый тест теперь продвинулся немного дальше. Это говорит о том, что Фрэнсис не получает свою собственную страницу со списком (он по-прежнему видит некоторые элементы списка Эдит.)

Может показаться, что мы не продвинулись ни на шаг, функционально сайт по-прежнему ведет себя почти так же, как в начале главы. Но это действительно прогресс. Мы начали движение по пути к новой структуре и реализовали много плацдармов, *не ухудшив ничего из того, что было раньше*. Давайте зафиксируем прогресс, которого мы достигли к этому моменту:

```
$ git status # должна показать 4 измененных файла и 1 новый, list.html
$ git add lists/templates/list.html
$ git diff # должна показать, чтобы упростили home.html,
           # переместили один тест в новый класс in lists/tests.py добавили
           # новое представление в views.py и упростили home_page и
           # сделали одно добавление в urls.py
$ git commit -a # добавляет сообщение, резюмирующее все, что выше, может быть
                # чем-то вроде "new URL, view and template to display lists"
```

Третий шажок: URL-адрес для добавления элементов списка

Где мы находимся в нашем собственном рабочем списке неотложных дел?

- *Скорректировать модель, чтобы элементы были связаны с разными списками*
- *Добавить уникальные URL для каждого списка...*
- *Добавить URL для создания нового списка посредством POST*
- *Добавить URL для добавления нового элемента в существующий список посредством POST*

Мы вроде как сделали прогресс на втором элементе, несмотря на то что имеется всего один список в мире. Первый элемент немного страшноват. Можем ли мы сделать что-то с элементом 3 или 4?

Давайте займемся новым URL-адресом для добавления новых элементов списка. Если нет ничего другого, это по крайней мере упростит представление домашней страницы.

Тестовый класс для создания нового списка

Откройте *lists/tests.py* и переместите методы `test_can_save_a_POST_request` и `test_redirects_after_POST` в новый класс, затем измените URL-адрес, на который они делают POST-запрос:

lists/tests.py (ch07l021-1)

```
class NewListTest(TestCase):
    '''тест нового списка'''

    def test_can_save_a_POST_request(self):
        '''тест: можно сохранить post-запрос'''
        self.client.post('/lists/new', data={'item_text': 'A new list item'})
        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new list item')

    def test_redirects_after_POST(self):
        '''тест: переадресует после post-запроса'''
```

```
        response = self.client.post('/lists/new', data={'item_text': 'A new
list item'})
        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['location'], '/lists/единственный-список-
в-мире/')
```

Кстати, это еще одно место, где следует обратить внимание на закрывающие косые черты. Речь идет о `/new` без закрывающей косой черты. Я использую форму записи, суть которой в том, что URL-адреса без такой черты являются URL-адресами «действия», которое изменяет базу данных.

Раз уж мы здесь, давайте научимся использовать новый метод тестового клиента Django, `assertRedirects`:

lists/tests.py (ch07l021-2)

```
def test_redirects_after_POST(self):
    '''тест: переадресует после post-запроса'''
    response = self.client.post('/lists/new', data={'item_text': 'A new list item'})
    self.assertRedirects(response, '/lists/единственный-список-в-мире/')
```

Тут не о чем особо рассказывать, он просто приятно заменяет два утверждения на одно...

Попробуйте выполнить:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
[...]
    self.assertRedirects(response, '/lists/единственный-список-в-мире/')
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

Первая неполадка говорит о том, что мы не сохраняем новый элемент в базе данных, а вторая – что вместо того, чтобы вернуть переадресацию с кодом состояния 302, наше представление возвращает код состояния 404. Все потому, что мы не создали URL-адрес для */lists/new*, поэтому `client.post` просто получает отклик «не найден».

Вы помните, как ранее мы разделили его на два теста? Если бы у нас был всего один тест, который проверяет сохранение и переадресацию, то он завершился бы на неполадке `0 != 1`, которую намного труднее отладить. Спросите у меня, я знаю.

URL-адрес и представление для создания нового списка

Теперь предлагаю создадать новый URL-адрес:

superlists/urls.py

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/new$', views.new_list, name='new_list'),
    url(r'^lists/единственный-список-в-мире/$', views.view_list, name='view_list'),
]
```

Мы получаем `no attribute 'new_list'`. Исправим это в *lists/views.py*:

lists/views.py (ch07l023-1)

```
def new_list(request):
    '''новый список'''
    pass
```

Затем получаем «The view lists.views.new_list didn't return an HttpResponse object» («Представление lists.views.new_list не вернуло объект HttpResponse»). (Уже становится привычным делом!) Можно было бы вернуть необработанный `HttpResponse`, но, поскольку мы знаем, что нам потребуется переадресация, давайте позаимствуем строку в `home_page`:

lists/views.py (ch07l023-2)

```
def new_list(request):
    '''новый список'''
    return redirect('/lists/единственный-список-в-мире/')
```

Это дает:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

Выглядит довольно прямолинейно. Позаимствуем еще одну строку:

lists/views.py (ch07l023-3)

```
def new_list(request):
    '''новый список'''
    Item.objects.create(text=request.POST['item_text'])
    return redirect('/lists/единственный-список-в-мире/')
```

И теперь все проходит:

```
Ran 7 tests in 0.030s

OK
```

Функциональные тесты показывают, что мы вернулись в рабочее состояние:

```
[...]
AssertionError: '1: Купить молоко' not found in ['1: Купить павлиньи перья',
'2: Купить молоко']
Ran 2 tests in 8.972s
FAILED (failures=1)
```

Удаление теперь уже избыточного кода и тестов

Мы выглядим неплохо. Поскольку новые представления делают большую часть работы, которую раньше делало представление `home_page`, следует максимально его упростить. К примеру, нельзя ли удалить весь блок `if request.method == 'POST'`?

lists/views.py

```
def home_page(request):
    return render(request, 'home.html')
```

Отлично!

```
OK
```

И раз уж мы здесь, мы также можем удалить теперь уже избыточный тест `test_only_saves_items_when_necessary`!

Разве это не замечательно? Функции представления выглядят намного проще. Мы снова выполняем тесты, чтобы убедиться…

```
Ran 6 tests in 0.016s
OK
```

А что насчет функциональных тестов?

Регрессия! Наведение форм на новый URL-адрес

Ой:

```
ERROR: test_can_start_a_list_for_one_user
[...]
  File "/.../superlists/functional_tests/tests.py", line 57, in
test_can_start_a_list_for_one_user
    self.wait_for_row_in_list_table('1: Купить павлиньи перья')
File "/.../superlists/functional_tests/tests.py", line 23, in
wait_for_row_in_list_table
```

```
    table = self.browser.find_element_by_id('id_list_table')
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_list_table"]

ERROR: test_multiple_users_can_start_lists_at_different_urls
[...]
  File "/.../superlists/functional_tests/tests.py", line 79, in
test_multiple_users_can_start_lists_at_different_urls
    self.wait_for_row_in_list_table('1: Купить павлиньи перья')
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_list_table"]
[...]

Ran 2 tests in 11.592s
FAILED (errors=2)
```

Все потому, что наши формы по-прежнему показывают на старый URL-адрес. Изменим *home.html* и *lists.html* на:

lists/templates/home.html, lists/templates/list.html

```
<form method="POST" action="/lists/new">
```

Это должно вернуть нас к работе:

```
AssertionError: '1: Купить молоко' not found in ['1: Купить павлиньи перья',
'2: Купить молоко']
[...]
FAILED (failures=1)
```

Назрела еще одна совершенно самодостаточная фиксация, в которой мы сделали кучу изменений в URL-адресах, *views.py* выглядит намного аккуратнее и чище, и мы уверены, что приложение по-прежнему работает так же хорошо, как и прежде. А ведь у нас неплохо получается с этой суетой вокруг перехода из одного рабочего состояния в другое!

```
$ git status # 5 измененных файлов
$ git diff # URL-адреса для 2 форм, перемещен код в представления
           # + тесты, новый URL-адрес
$ git commit -a
```

И мы можем вычеркнуть пункт в рабочем списке неотложных дел:

- *Скорректировать модель, чтобы элементы были связаны с разными списками*
- *Добавить уникальные URL для каждого списка...*
- ~~*Добавить URL для создания нового списка посредством POST*~~
- *Добавить URL для добавления нового элемента в существующий список посредством POST*

Стиснув зубы: корректировка моделей

Хватит заниматься служебными операциями с URL-адресами. Пора стиснуть зубы и изменить модели. Давайте скорректируем модульный тест модели. Просто для разнообразия я представлю изменения в виде разницы:

lists/tests.py

```
@@ -1,5 +1,5 @@
from django.test import TestCase
-from lists.models import Item
+from lists.models import Item, List

 class HomePageTest(TestCase):
@@ -44,22 +44,32 @@ class ListViewTest(TestCase):

-class ItemModelTest(TestCase):
+class ListAndItemModelsTest(TestCase):

     def test_saving_and_retrieving_items(self):
+        list_ = List()
+        list_.save()
+
         first_item = Item()
         first_item.text = 'Первый (самый) элемент списка'
+        first_item.list = list_
         first_item.save()

         second_item = Item()
         second_item.text = 'Элемент второй'
+        second_item.list = list_
```

```
        second_item.save()

+       saved_list = List.objects.first()
+       self.assertEqual(saved_list, list_)
+
        saved_items = Item.objects.all()
        self.assertEqual(saved_items.count(), 2)

        first_saved_item = saved_items[0]
        second_saved_item = saved_items[1]
        self.assertEqual(first_saved_item.text, 'Первый (самый) элемент списка')
+       self.assertEqual(first_saved_item.list, list_)
        self.assertEqual(second_saved_item.text, 'Элемент второй')
+       self.assertEqual(second_saved_item.list, list_)
```

Мы создаем новый объект `List` и назначаем ему каждый элемент, присваивая их его свойству `.list`. Проверяем, что список должным образом сохранен и что эти два элемента тоже сохранили свою связь со списком. Вы также заметите, что можно непосредственно сравнивать объекты списков друг с другом (`saved_list` и `list_`) – за кулисами они будут сравниваться путем проверки, что их первичный ключ (атрибут `.id`) одинаков.

Я использую имя переменной `list_`, чтобы избежать «затенения» встроенной функции Python `list`. Выглядит уродливо, но все другие варианты, которые я испробовал, были такими же уродливыми или хуже (`my_list`, `the_list`, `listl`, `listey`...).

Самое время для еще одного цикла «модульный-тест/программный-код».

Что касается первой пары итераций, то вместо того, чтобы явно показывать, какой код вводить между каждым прогоном теста, я покажу только ожидаемые сообщения об ошибке, появляющиеся в результате выполнения тестов. Предоставлю вам возможность самостоятельно выяснить, каким должно быть каждое минимальное изменение кода в отдельности:

Нужна подсказка? Вернитесь и взгляните на шаги, которые мы предприняли в предпоследней главе, чтобы внедрить модель `Item`.

Вашей первой ошибкой должна быть:

```
ImportError: cannot import name 'List'
```

Исправьте ее, затем вы увидите:

```
AttributeError: 'List' object has no attribute 'save'
```

Далее:

```
django.db.utils.OperationalError: no such table: lists_list
```

Поэтому мы выполняем `makemigrations`:

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0003_list.py
    - Create model List
```

И затем вы должны увидеть:

```
    self.assertEqual(first_saved_item.list, list_)
AttributeError: 'Item' object has no attribute 'list'
```

Связь по внешнему ключу

Каким образом передать объекту `Item` атрибут списка? Давайте просто попытаемся наивно передать его как атрибут `text` (между прочим, это хорошая возможность проверить, что ваше решение похоже на мое):

lists/models.py

```
from django.db import models

class List(models.Model):
    '''список'''
    pass

class Item(models.Model):
    '''элемент'''
    text = models.TextField(default='')
    list = models.TextField(default='')
```

Как обычно, тесты говорят о том, что требуется миграция:

```
$ python manage.py test lists
[...]
django.db.utils.OperationalError: no such column: lists_item.list

$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0004_item_list.py
    - Add field list to item
```

Посмотрим, что это нам дает:

```
AssertionError: 'List object' != <List: List object>
```

Мы не совсем готовы. Внимательно посмотрите на каждую сторону выражения `!=`. Средствами Django было сохранено только строковое представление объекта `List`. Чтобы сохранить связь с самим объектом, мы сообщаем Django о связи между этими двумя классами, используя внешний ключ `ForeignKey`:

lists/models.py

```
from django.db import models

class List(models.Model):
    '''список'''
    pass

class Item(models.Model):
    '''элемент'''
    text = models.TextField(default='')
    list = models.ForeignKey(List, default=None)
```

Это тоже потребует миграции. Поскольку последнее было отвлекающим маневром, давайте удалим его и заменим на новое:

```
$ rm lists/migrations/0004_item_list.py
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0004_item_list.py
    - Add field list to item
```

Удаление миграций чревато опасностями. Потому что не всегда удается с первого раза написать работающий программный код для моделей. Но если вы удаляете миграцию, которая уже где-нибудь применена к базе данных, веб-платформа Django будет озадачена тем, в каком состоянии она находится и как применять будущие миграции. Вы должны это делать, только если уверены, что миграция не использовалась. Хорошее эмпирическое правило: никогда не удалять или изменять миграцию, которая уже была зафиксирована в системе управления версиями (VCS).

Адаптация остальной части мира к новым моделям

Вернемся к нашим тестам. Что же теперь происходит?

```
$ python manage.py test lists
[...]
ERROR: test_displays_all_items (lists.tests.ListViewTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_can_save_a_POST_request (lists.tests.NewListTest)
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id

Ran 6 tests in 0.021s

FAILED (errors=3)
```

Вот это да!

Но есть немного хороших новостей. Хотя их и трудно увидеть, наши тесты модели проходят, правда три теста представления показывают грубые неполадки.

Причина – в новой связи, которую мы внесли между элементами и списками. Она требует, чтобы каждый элемент имел родительский список, к чему наши старые тесты и программный код не подготовлены.

Однако именно для этого нам нужны тесты! Давайте приведем их снова в рабочий вид. Самым простым является тест представления списка `ListViewTest`; мы просто создаем родительский список для двух тестируемых элементов:

lists/tests.py (ch07l031)

```
class ListViewTest(TestCase):
    '''тест представления списка'''

    def test_displays_all_items(self):
        '''тест: отображаются все элементы списка'''
        list_ = List.objects.create()
        Item.objects.create(text='itemey 1', list=list_)
        Item.objects.create(text='itemey 2', list=list_)
```

Это приводит нас всего к двум не работающим тестам, оба связаны с попыткой выполнить POST-запрос к представлению `new_list`. Расшифруем обратную трассировку с помощью нашей обычной техники, проходя назад – от ошибки к закопанной, где-то внутри, строке с тестовым кодом

и далее к строке с нашим собственным программным кодом, которая вызвала неполадку:

```
File "/.../superlists/lists/views.py", line 9, in new_list
Item.objects.create(text=request.POST['item_text'])
```

Это происходит при попытке создать элемент без родительского списка. Поэтому вносим соответствующее изменение в представление:

lists/views.py

```
from lists.models import Item, List
[...]

def new_list(request):
    '''новый список'''
    list_ = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect('/lists/единственный-список-в-мире/')
```

И это снова заставляет наши тесты пройти успешно:

```
Ran 6 tests in 0.030s

OK
```

В этой точке вы испытали внутреннюю досаду, не так ли? *Аргумент! Это выглядит совершенно неправильным: мы создаем новый список для каждого нового представления элемента и по-прежнему просто выводим на экран все элементы, как будто они принадлежат одному и тому же списку!* Согласен, я чувствую то же самое. Постепенный подход, в котором вы идете от рабочего кода к рабочему коду, противоречит здравому смыслу. Я всегда испытываю желание просто взять и все исправить за один раз вместо того, чтобы идти из одного странного, наполовину законченного состояния в другое. Но помните про Билли-тестировщика! Когда вы карабкаетесь на гору, вы очень тщательно продумываете, куда ставить свою ногу, и делаете по одному шагу за один раз, на каждом этапе проверяя, что место, куда вы ее поставили, не приведет вас к падению с утеса.

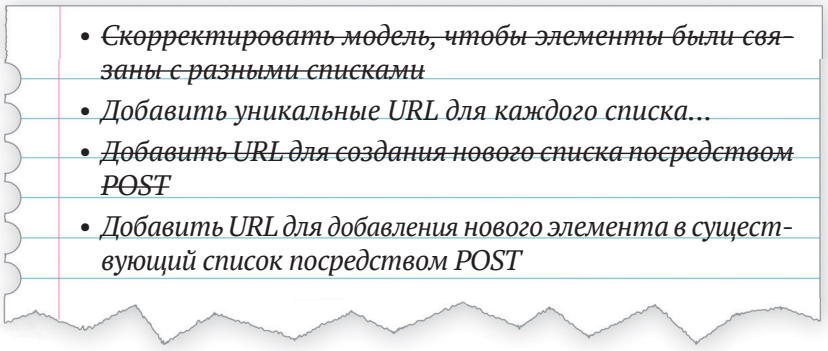
Поэтому, только чтобы обнадежить себя, что все работает, мы повторно выполняем ФТ.

```
AssertionError: '1: Купить молоко' not found in ['1: Купить павлиньи перья',
'2: Купить молоко']
[...]
```

Разумеется, он спокойно добирается туда, где мы были прежде. Мы ничего не повредили и внесли изменение в базу данных. И это можно только поприветствовать! Зафиксируем:

```
$ git status # 3 измененных файла плюс 2 миграции
$ git add lists
$ git diff --staged
$ git commit
```

Можем вычеркнуть еще один пункт из списка неотложных дел:



Каждому списку – свой URL-адрес

Что мы будем использовать в качестве уникального идентификатора для списков? Вероятно, самым простым пока будет использование автоматически генерируемого поля `id` из базы данных. Давайте изменим `ListViewTest`, чтобы два теста указывали на новые URL-адреса.

Мы также изменим старый тест `test_displays_all_items`. Назовем его `test_displays_only_items_for_that_list` и заставим проверять, что отображаются лишь элементы определенного списка:

lists/tests.py (ch07l033)

```
class ListViewTest(TestCase):
     '''тест представления списка'''

    def test_uses_list_template(self):
        '''тест: используется шаблон списка'''
        list_ = List.objects.create()
        response = self.client.get(f'/lists/{list_.id}/')
        self.assertTemplateUsed(response, 'list.html')

    def test_displays_only_items_for_that_list(self):
        '''тест: отображаются элементы только для этого списка'''
        correct_list = List.objects.create()
        Item.objects.create(text='itemey 1', list=correct_list)
```

```
        Item.objects.create(text='itemey 2', list=correct_list)
        other_list = List.objects.create()
        Item.objects.create(text='другой элемент 1 списка', list=other_list)
        Item.objects.create(text='другой элемент 2 списка', list=other_list)

        response = self.client.get(f'/lists/{correct_list.id}/')

        self.assertContains(response, 'itemey 1')
        self.assertContains(response, 'itemey 2')
        self.assertNotContains(response, 'другой элемент 1 списка')
        self.assertNotContains(response, 'другой элемент 2 списка')
```

Еще парочка этих прекрасных f-string в распечатке! Если они по-прежнему кажутся вам чем-то загадочным, обратитесь к документации[2] (хотя, если ваше формальное образование по Computer Science такое же плохое, как и мое, вы, вероятно, формальную грамматику пропустите).

Выполнение модульных тестов дает ожидаемую ошибку 404 и еще одну, связанную с ней:

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was
404 (expected 200)
[...]
FAIL: test_uses_list_template (lists.tests.ListViewTest)
AssertionError: Не использовался ни один шаблон для вывода отклика в
качестве HTML
```

Извлечение параметров из URL-адресов

Самое время узнать, как передавать параметры из URL-адресов в представления:

superlists/urls.py

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/new$', views.new_list, name='new_list'),
    url(r'^lists/(.+)/$', views.view_list, name='view_list'),
]
```

Скорректируем регулярное выражение для нашего URL-адреса, включив в его состав *группу захвата*, `(.+)`, которая будет отождествлять любые

[2] См. https://docs.python.org/3/reference/lexical_analysis.html%23f-strings

символы вплоть до следующего `/`. Полученный текст будет передан представлению в виде аргумента.

Другими словами, если мы перейдем по URL-адресу */lists/1/*, то `view_list` получит второй аргумент после нормального аргумента `request`, а именно – строку «1». Если мы переходим по */lists/foo/*, то получаем `view_list(request, "foo")`.

Но наше представление еще не готово получить аргумент! Разумеется, это провоцирует проблемы:

```
ERROR: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
[...]
ERROR: test_uses_list_template (lists.tests.ListViewTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
[...]
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
[...]
TypeError: view_list() takes 1 positional argument but 2 were given
FAILED (errors=3)
```

Это легко исправить с помощью фиктивного параметра в *views.py*:

lists/views.py

```
def view_list(request, list_id):
    '''представление списка'''
    [...]
```

Приходим к ожидаемой неполадке:

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
[...]
AssertionError: 1 != 0 : Response should not contain 'другой элемент 1 списка'
```

Давайте заставим наше представление различать, какие элементы оно отправляет в шаблон:

lists/views.py

```
def view_list(request, list_id):
    '''представление списка'''
    list_ = List.objects.get(id=list_id)
    items = Item.objects.filter(list=list_)
    return render(request, 'list.html', {'items': items})
```

Адаптирование new_list к новому миру

М-да, теперь появились ошибки еще в одном тесте:

```
ERROR: test_redirects_after_POST (lists.tests.NewListTest)
ValueError: недопустимый литерал для int() в десятичной системе:
'единственный-список-в-мире'
```

Тогда давайте взглянем на этот тест, раз уж он кряхтит:

lists/tests.py

```
class NewListTest(TestCase):
    '''тест нового списка'''
    [...]

    def test_redirects_after_POST(self):
        '''тест: переадресуется после post-запроса'''
        response = self.client.post('/lists/new', data={'item_text': 'A new
list item'})
        self.assertRedirects(response, '/lists/единственный-список-в-мире/')
```

Похоже, он не был адаптирован к новому миру `Lists` и `Items`. Этот тест должен сообщать, что представление переадресовывает на URL-адрес конкретного нового списка, который оно только что создало:

lists/tests.py (ch07l036-1)

```
def test_redirects_after_POST(self):
    '''тест: переадресуется после post-запроса'''
    response = self.client.post('/lists/new', data={'item_text': 'A new
list item'})
    new_list = List.objects.first()
    self.assertRedirects(response, f'/lists/{new_list.id}/')
```

Этот тест по-прежнему дает ошибку о наличии *недопустимого литерала*. Смотрим на само представление и изменяем его так, чтобы оно переадресовывало в допустимое место:

lists/views.py (ch07l036-2)

```
def new_list(request):
    '''новый список'''
    list_ = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect(f'/lists/{list_.id}/')
```

Это возвращает нас к состоянию, когда модульные тесты проходят.

```
$ python3 manage.py test lists
[...]
.....
 ----------------------------------------------------------------------
Ran 6 tests in 0.033s

OK
```

А как быть с функциональными тестами? Мы почти у цели?

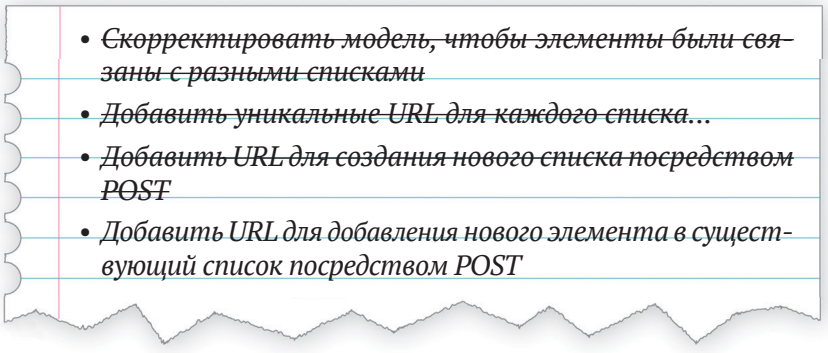
Функциональные тесты обнаруживают еще одну регрессию

Ну, почти:

```
F.
======================================================================
FAIL: test_can_start_a_list_for_one_user
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 67, in
test_can_start_a_list_for_one_user
    self.wait_for_row_in_list_table('2: Сделать мушку из павлиньих перьев')
[...]
AssertionError: '2: Сделать мушку из павлиньих перьев' not found in ['1:
Сделать мушку из павлиньих перьев']

 ----------------------------------------------------------------------
Ran 2 tests in 8.617s

FAILED (failures=1)
```

Новый тест фактически проходит, и разные пользователи могут получать разные списки, но старый тест предупреждает нас о регрессии. Похоже, больше нельзя добавлять второй элемент в список. Все из-за нашего чернового маневра, где мы создаем новый список для каждой отдельной отправки POST-запроса. Именно для этого и существуют функциональные тесты!

И это приятно коррелирует с последним элементом в нашем рабочем списке неотложных дел:



Еще одно представление для добавления элементов в существующий список

Нам нужен URL-адрес и представление для обработки добавления нового элемента в существующий список (*/lists/<list_id>/add_item*). У нас уже довольно неплохо получается, поэтому набросаем на скорую руку:

lists/tests.py

```
class NewItemTest(TestCase):
    '''тест нового элемента списка'''

    def test_can_save_a_POST_request_to_an_existing_list(self):
        '''тест: можно сохранить post-запрос в существующий список'''
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            f'/lists/{correct_list.id}/add_item',
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new item for an existing list')
        self.assertEqual(new_item.list, correct_list)

    def test_redirects_to_list_view(self):
        '''тест: переадресуется в представление списка'''
        other_list = List.objects.create()
```

```
        correct_list = List.objects.create()

        response = self.client.post(
            f'/lists/{correct_list.id}/add_item',
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertRedirects(response, f'/lists/{correct_list.id}/')
```

Вас интересует `other_list`? Почти так же, как в тестах для просмотра конкретного списка, очень важно, чтобы мы добавляли элементы в конкретный список. Добавление этого второго объекта в базу данных не дает мне использовать в реализации маневр `List.objects.first()`. Это было бы глупо делать, и можно зайти слишком далеко в тестировании все этих глупостей, которые вы не должны делать (в конце концов, их бесконечно много). Такое решение принимается по собственному усмотрению, но конкретно это, кажется, стоит того. Во второй части книги по этому поводу имеется дополнительное обсуждение в разделе «Ремарка о том, когда тестировать разработчика на глупость».

Мы получаем:

```
AssertionError: 0 != 1
[...]
AssertionError: 301 != 302 : Отклик не переадресовал, как ожидалось: код
состояния отклика 301 (ожидался 302)
```

Остерегайтесь «жадных» регулярных выражений!

Немного странно. Мы еще фактически не указывали URL-адрес для */lists/1/add_item*, поэтому ожидаемая неполадка должна быть `404 != 302`. Почему же мы получаем 301?

Загадка! Все дело в том, что мы использовали очень «жадное» регулярное выражение в URL:

superlists/urls.py

```
url(r'^lists/(.+)/$', views.view_list, name='view_list'),
```

В Django есть встроенный код для выдачи постоянной переадресации (301), когда кто-то запрашивает URL, который *почти* верный, за исключением недостающей косой черты. В данном случае адрес */lists/1/add_item/* соответствовал бы `lists/(.+)/`, где `(.+)` захватывает `1/add_item`. Поэтому

Django «с готовностью» догадывается, что мы на самом деле хотели URL-адрес с закрывающей косой чертой.

Применив регулярное выражение `\d` мы можем это исправить, заставив шаблон URL-адреса в явном виде захватывать только знаки цифр:

superlists/urls.py

```
url(r'^lists/(\d+)/$', views.view_list, name='view_list'),
```

Это дает неполадку, которую мы ожидали:

```
AssertionError: 0 != 1
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected:
Response
code was 404 (expected 302)
```

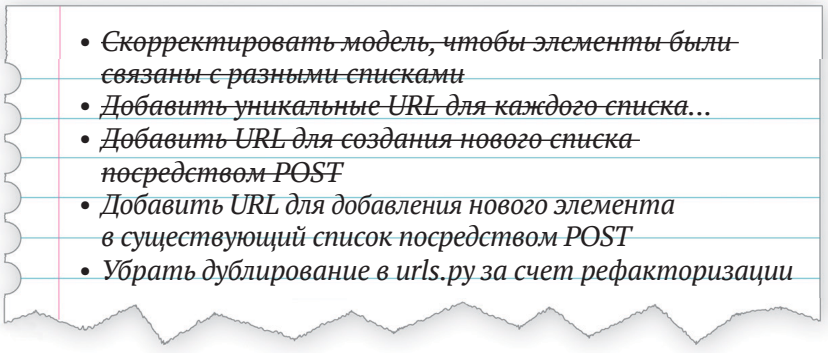
Последний новый URL-адрес

Теперь у нас есть ожидаемый код состояния 404, давайте добавим новый URL для создания новых элементов в существующих списках:

superlists/urls.py

```
urlpatterns = [
    url(r'^$', views.home_page, name='home'),
    url(r'^lists/new$', views.new_list, name='new_list'),
    url(r'^lists/(\d+)/$', views.view_list, name='view_list'),
    url(r'^lists/(\d+)/add_item$', views.add_item, name='add_item'),
]
```

Здесь три очень похожих URL. Пометим это в нашем рабочем списке неотложных задач – это хорошие кандидаты на рефакторизацию.



Возвращаясь к тестам, мы получаем обычные недостающие объекты представления модуля:

```
AttributeError: module 'lists.views' has no attribute 'add_item'
```

Последнее новое представление

Давайте попробуем:

lists/views.py

```
def add_item(request):
    '''добавить элемент'''
    pass
```

Так:

```
TypeError: add_item() принимает 1 позиционный аргумент, но было передано 2
```

lists/views.py

```
def add_item(request, list_id):
    '''добавить элемент'''
    pass
```

И затем:

```
ValueError: The view lists.views.add_item didn't return an HttpResponse object.
It returned None instead.
```

Можно скопировать `redirect` из `new_list` и `List.objects.get` из `view_list`:

lists/views.py

```
def add_item(request, list_id):
    '''добавить элемент'''
    list_ = List.objects.get(id=list_id)
    return redirect(f'/lists/{list_.id}/')
```

Это приводит нас к:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

Наконец мы заставляем его сохранить новый элемент списка:

lists/views.py

```
def add_item(request, list_id):
    '''добавить элемент'''
    list_ = List.objects.get(id=list_id)
```

```
    Item.objects.create(text=request.POST['item_text'], list=list_)
    return redirect(f'/lists/{list_.id}/')
```

И возвращаемся к состоянию, когда тесты проходят.

```
Ran 8 tests in 0.050s

OK
```

Тестирование контекстных объектов отклика напрямую

У нас есть новое представление и URL для добавления элементов в существующие списки. Теперь нужно просто им воспользоваться в нашем шаблоне *list.html*. Поэтому мы открываем его для корректировки тега формы…

lists/templates/list.html

```
<form method="POST" action="но что же нам сюда поместить?">
```

О! Чтобы получить URL для добавления в текущий список, шаблон должен знать, какой список он генерирует, а также каковы его элементы. Мы хотели бы сделать что-то вроде этого:

lists/templates/list.html

```
<form method="POST" action="/lists/{{ list.id }}/add_item">
```

Чтобы это работало, представление должно передать список шаблону. Давайте создадим новый модульный тест в `ListViewTest`:

lists/tests.py (ch07l041)

```
def test_passes_correct_list_to_template(self):
    '''тест: передается правильный шаблон списка'''
    other_list = List.objects.create()
    correct_list = List.objects.create()
    response = self.client.get(f'/lists/{correct_list.id}/')
    self.assertEqual(response.context['list'], correct_list)  ❶
```

❶ `response.context` представляет контекст, который мы собираемся передать в функцию генерирования HTML `render` – тестовый клиент Django помещает его для нас в объект `response`, чтобы помочь с тестированием.

Это дает нам:

```
KeyError: 'list'
```

потому что мы не передаем список в шаблон. На самом же деле это позволяет немного упростить код:

lists/views.py

```
def view_list(request, list_id):
    '''представление списка'''
    list_ = List.objects.get(id=list_id)
    return render(request, 'list.html', {'list': list_})
```

В результате это, конечно же, нарушит один из наших старых тестов, потому что шаблону нужны `items`:

```
FAIL: test_displays_only_items_for_that_list (lists.tests.ListViewTest)
[...]
AssertionError: False is not true : Couldn't find 'itemey 1' in response
```

Но мы можем исправить это в *list.html*, а также скорректировать атрибут action формы с POST-запросом:

lists/templates/list.html (ch07l043)

```
<form method="POST" action="/lists/{{ list.id }}/add_item">  ❶

  [...]

  {% for item in list.item_set.all %}  2
    <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
  {% endfor %}
```

❶ Это действие нашей новой формы.

❷ `.item_set` называется обратным просмотром[3]. Это один из невероятно полезных элементов Django ORM, который позволяет находить связанные с объектом элементы из другой таблицы…

Таким образом, это решение приводит к тому, что модульные тесты проходят:

```
Ran 9 tests in 0.040s

OK
```

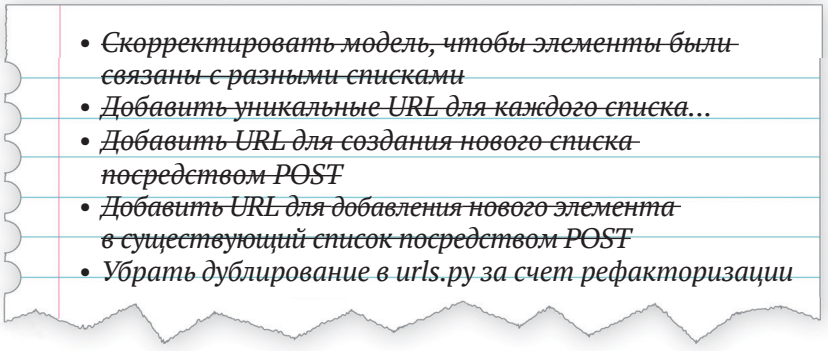
А что насчет функциональных тестов?

```
$ python manage.py test functional_tests
[...]
```

[3] См. https://docs.djangoproject.com/en/1.11/topics/db/queries/%23following-relationships-backward

```
..
 ----------------------------------------------------------------------
Ran 2 tests in 9.771s

OK
```

Ура! Да, и сразу же быстрая сверка с нашим рабочим списком неотложных дел:



Особенно раздражает то, что Билли-тестировщик – еще и ярый сторонник доведения дел до конца, поэтому мы должны выполнить эту заключительную работу.

Для начала выполним фиксацию – всегда следите за тем, что вы зафиксировали рабочее состояние перед тем, как заняться рефакторизацией:

```
$ git diff
$ git commit -am "Новый URL + представление для добавления в существующие
списки. ФТ проходит :-)"
```

Финальная рефакторизация с использованием URL-включений

superlists/urls.py в действительности предназначен для URL-адресов, которые применяются ко всему сайту. Для URL-адресов, которые применяются только к приложению `lists`, Django рекомендует использовать отдельный файл *lists/urls.py*, чтобы придать приложению более самодостаточный вид. Самый простой способ – использовать копии существующего *urls.py*:

```
$ cp superlists/urls.py lists/
```

Заменяем три строки в *superlists/urls.py* на `include`.

superlists/urls.py

```
from django.conf.urls import include, url
from lists import views as list_views  ❶
from lists import urls as list_urls  ❶

urlpatterns = [
  url(r'^$', list_views.home_page, name='home'),
  url(r'^lists/', include(list_urls)),  ❷
]
```

❶ Пока мы тут, используем синтаксическую конструкцию `import x as y` для назначения псевдонима `views` и `urls`. Это хорошая практика для высокоуровневого *urls.py*, она позволит нам, если нужно, импортировать представления и URL-адреса из многочисленных приложений – позже нам это понадобится.

❷ Это включение `include`. Обратите внимание, что оно может быть частью регулярного выражения в URL как префикс, который будет применяться ко всем включенным URL-адресам (это то место, где мы сокращаем дублирование и придаем программному коду более удачную структуру):

Вернувшись в *lists/urls.py*, можно их подрезать, чтобы включать только последнюю часть трех URL-адресов и ничего другого, относящегося к родительскому *urls.py*:

lists/urls.py (ch07l046)

```
from django.conf.urls import url
from lists import views

urlpatterns = [
  url(r'^new$', views.new_list, name='new_list'),
  url(r'^(\d+)/$', views.view_list, name='view_list'),
  url(r'^(\d+)/add_item$', views.add_item, name='add_item'),
]
```

Выполните модульные тесты повторно, чтобы проверить, что все работает.

Сделав это, я просто не мог поверить, что у меня все получлось правильно с первой попытки. Всегда стоит скептически относиться к своим способностям, поэтому я намеренно немного изменил один из URL-адресов, только чтобы убедиться, что это нарушит тест. И действительно. Мы подстрахованы.

Не стесняйтесь пробовать сами! Не забудьте вернуть все назад, проверить, что тесты снова проходят, и затем выполнить заключительную фиксацию:

```
$ git status
$ git add lists/urls.py
$ git add superlists/urls.py
$ git diff --staged
$ git commit
```

Уф. Эта глава – все равно что марафонский забег. Но мы затронули много важных тем, начиная с изоляции тестов и некоторых взглядов на структуру кода. Охватили некоторые эмпирические правила, такие как YAGNI и «если клюнуло трижды – рефакторизуй». Но самое главное – мы увидели, как адаптировать существующий сайт шаг за шагом, переходя из одного рабочего состояния в другое, чтобы итеративно прийти к новой структуре кода.

Я бы сказал, мы близки к тому, чтобы быть в состоянии представить этот сайт заказчику, как самую первую бетаверсию веб-сайта суперсписков, который покроит мир. Возможно, потребуется придать ему немного привлекательный вид… Давайте посмотрим в нескольких следующих главах, что нужно сделать, чтобы его развернуть.

Еще немного философии TDD

Из одного рабочего состояния в другое, или Билли-тестировщик против Тома-рефакторщика

Наш естественный порыв – брать и исправлять все за один присест… Но если мы не будем осторожны, то закончим как Том-рефакторщик, то есть с грудой изменений в программном коде, который к тому же не работает. Билли-тестировщик рекомендует делать по одному шагу за один раз и переходить из одного рабочего состояния в другое.

Разбить работу на маленькие достижимые задачи

Иногда это означает, что нужно начинать со скучной работы, вместо того чтобы стразу приступать к забавной части. Но вам придется поверить, что в параллельной вселенной, где вы нарушили все, что можно, вы, вероятно, переживали бы тяжелые времена, изо всех сил пытаясь заставить приложение заработать снова.

YAGNI

Вам это не понадобится! Избегайте искушения писать программный код, который, по вашему мнению, может быть полезным, просто потому, что в тот момент он напрашивается сам. Велики шансы, что вы им не воспользуетесь либо вы неправильно оценили свои будущие потребности. Смотрите главу 22 о методологии, которая помогает избежать этой ловушки.

Часть II

Непременные условия веб-разработки

Настоящие разработчики [свой продукт] поставляют.
– Джеф Этвуд

Если бы эта книга была простым руководством по методологии TDD в нормальных условиях программирования, мы, возможно, смогли бы поздравить себя с достижением текущего состояния. В конце концов, у нас есть некоторые твердые основы методологии TDD и Django за плечами; у нас есть все для того, чтобы создать веб-сайт.

Но настоящие разработчики свой продукт поставляют, и чтобы его поставить, мы встаем перед необходимостью заняться некоторыми более тонкими, но неизбежными аспектами веб-разработки: статическими файлами, подтверждением допустимости данных формы, ужасающим JavaScript. Но самым пугающим своей сложностью является развертывание на производсвенном сервере.

На каждом из этих этапов методология TDD также поможет нам во всем разобраться.

В этом разделе я по-прежнему постараюсь сохранить кривую обучения относительно мягкой, однако мы встретим несколько новых важных понятий и технологий. Мне удастся лишь слегка коснуться каждой темы – надеюсь продемонстрировать их в достаточной мере, чтобы вы могли приступить к работе над собственным проектом. Но вам также придется заняться расширением своего круга чтения, когда вы начнете применять эти темы на практике.

Например, если вы не были знакомы с Django до того, как приступили к чтению этой книги, можете взять в этой точке небольшую паузу, чтобы пробежаться по официальному учебному руководству Django. Это дополнит знания, которым вы научились до сих пор, сделает вас более уверенными в Django на протяжении следующих нескольких глав и вы сможете сосредоточиться на ключевых понятиях.

О, впереди столько забавного! То ли еще будет!

Глава 8

Придание привлекательного вида: макет, стилевое оформление сайта и что тут тестировать

Мы подумываем о релизе первой версии нашего сайта, но немного смущает то, как уродливо он сейчас выглядит. В этой главе мы рассмотрим некоторые основы оформления, включая интеграцию платформы HTML/CSS под названием Bootstrap. Мы изучим, как статические файлы работают в Django и что нам нужно делать для их тестирования.

Что функционально тестируется в макете и стилевом оформлении

Бесспорно, наш сайт сейчас выглядит непривлекательно (рис. 8.1).

Если вы запустите свой сервер разработки командой `manage.py runserver`, то можете столкнуться с ошибкой базы данных «table lists_item has no column named list_id» (в таблице `lists_item` отсутствует столбец с именем list_id). Нужно обновить локальную базу данных, чтобы отразились изменения, которые мы внесли в *models.py*. Примените команду `manage.py migrate`. Если она будет доставать вас ошибкой `IntegrityErrors`, просто удалите[1] файл базы данных и попробуйте еще раз.

[1] Что? Удалить базу данных? Да вы с ума сошли? Не совсем. Локальная база данных разработки нередко рассинхронизируется со своими миграциями, по мере того как мы возвращаемся назад и движемся вперед в нашем процессе разработки. Она не содержит каких-то важных данных, поэтому ничего страшного не произойдет, если время от времени ее обнулять. Однако мы будем гораздо внимательнее, когда у нас на сервере появится «производственная» база данных. Более подробно об этом можно прочитать в приложении D.

Мы не можем усиливать репутацию Python как уродливого языка[2], поэтому давайте внесем свою лепту в его полировку. Вот пара вещей, которые мы, возможно, захотим сделать:

- Хорошее большое поле ввода для добавления новых и существующих списков.
- Большое, привлекающее внимание поле по центру для вставки списка.

Каким образом применить методологию TDD в этих случаях? Большинство людей скажет, что не следует тестировать эстетику, и они будут правы. Это несколько смахивает на тестирование константы в том смысле, что тесты обычно не добавляют никаких значений.

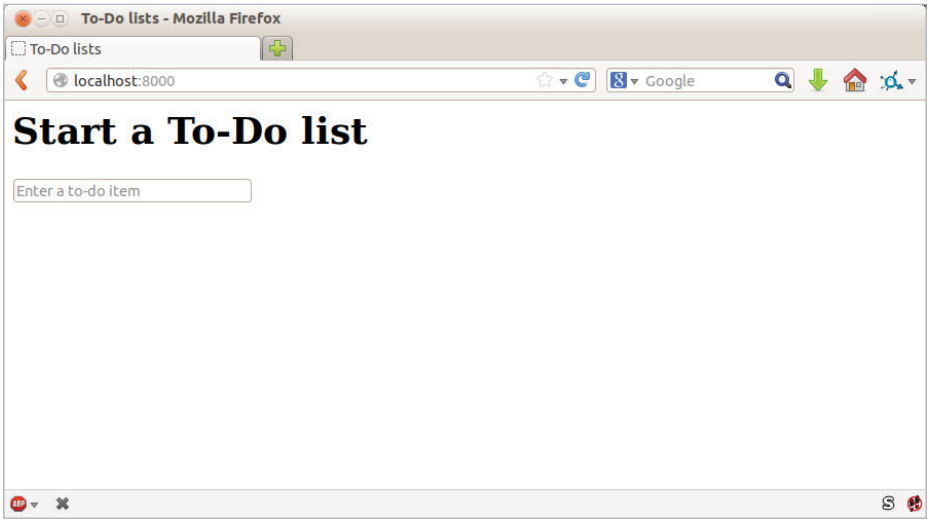


Рис. 8.1. Наша домашняя страница выглядит немного уродливой...

Но мы можем протестировать *реализацию* нашей эстетики – как раз чтобы убедиться, что все работает. Например, для стилевого оформления мы будем применять каскадные таблицы стилей (CSS), они загружаются как статические файлы. Статические файлы бывает сложновато конфигурировать (особенно, как мы увидим позже, когда вы «перезжаете» со своего ПК на размещающий сайт). Поэтому нам нужно что-то вроде простой «проверки на токсичность» (smoke test), подтверждающей, что CSS загрузился. Нам не требуется тестировать шрифты и цвета и каждый пиксель, но мы можем сделать быструю проверку, что на каждой странице основное поле ввода выровнено именно так, как мы хотим, и это даст нам уверенность, что остальная часть стилевого оформления страницы, вероятно, тоже загрузилась.

Начнем с нового метода тестирования в функциональном тесте:

[2] См. http://grokcode.com/746/dear-python-why-are-you-so-ugly/

functional_tests/tests.py (ch08l001)

```
class NewVisitorTest(LiveServerTestCase):
    '''тест нового посетителя'''
    [...]

    def test_layout_and_styling(self):
        '''тест макета и стилевого оформления'''
        # Эдит открывает домашнюю страницу
        self.browser.get(self.live_server_url)
        self.browser.set_window_size(1024, 768)

        # Она замечает, что поле ввода аккуратно центрировано
        inputbox = self.browser.find_element_by_id('id_new_item')
        self.assertAlmostEqual(
            inputbox.location['x'] + inputbox.size['width'] / 2,
            512,
            delta=10
        )
```

Здесь есть несколько новых нюансов. Мы начинаем с установки размера окна в фиксированное значение. Затем отыскиваем элемент `input`, смотрим на его размер и расположение и выполняем небольшие математические вычисления, чтобы проверить, расположен ли он посередине страницы. `assertAlmostEqual` помогает нам справиться с погрешностями округления и случайными странностями из-за полос прокрутки и т. п., позволяя нам указать, что мы хотели бы, чтобы результаты наших арифметических вычислений работали с точностью до +/–10 пикселов.

Если мы выполним функциональные тесты, то получим:

```
$ python manage.py test functional_tests
[...]
.F.
======================================================================
FAIL: test_layout_and_styling (functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 129, in
test_layout_and_styling
    delta=10
AssertionError: 107.0 != 512 within 10 delta

 ----------------------------------------------------------------------
Ran 3 tests in 9.188s

FAILED (failures=1)
```

Это ожидаемая неполадка. Однако этот вид ФТ легко понять неправильно, поэтому давайте применим черновое решение «с обманом», чтобы проверить, что ФТ тоже проходит, когда поле ввода центрировано. Мы удалим этот программный код, как только применим его для проверки ФТ:

lists/templates/home.html (ch08l002)

```
<form method="POST" action="/lists/new">
  <p style="text-align: center;">
    <input name="item_text" id="id_new_item" placeholder="Введите элемент
списка" />
  </p>
  {% csrf_token %}
</form>
```

Этот вариант проходит, а значит, ФТ работает. Давайте расширим его, чтобы удостовериться, что поле ввода для нового списка тоже выровнено по центру:

functional_tests/tests.py (ch08l003)

```
# Она начинает новый список и видит, что поле ввода там тоже
# аккуратно центировано
inputbox.send_keys('testing')
inputbox.send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: testing')
inputbox = self.browser.find_element_by_id('id_new_item')
self.assertAlmostEqual(
    inputbox.location['x'] + inputbox.size['width'] / 2,
    512,
    delta=10
)
```

Это дает нам еще одну неполадку теста:

```
  File "/.../superlists/functional_tests/tests.py", line 141, in
test_layout_and_styling
    delta=10
AssertionError: 107.0 != 512 within 10 delta
```

Давайте зафиксируем только ФТ:

```
$ git add functional_tests/tests.py
$ git commit -m "Первые шаги ФТ для макетирования + стилевого оформления"
```

Теперь, похоже, мы имеем все основания для поиска надлежащего решения, которое отвечает нашей потребности в более хорошем стиле-

вом оформлении сайта. Мы можем отступить от неуклюжего решения с `<p style="text-align: center">`:

```
$ git reset --hard
```

Команда Git `$ git reset --hard` – это, что называется, «взлетай и уничтожай территорию с орбиты», поэтому будьте с ней осторожны – она сдует все ваши незафиксированные изменения. В отличие от почти всего остального, что можно делать при помощи Git, после этой команды нет никакого способа отыграть назад.

Придание привлекательного вида: использование платформы CSS

Дизайн – это сложный творческий процесс, и он сложен вдвойне теперь, когда приходится иметь дело с мобильными телефонами, планшетами и т. д. Вот почему многие программисты, особенно такие ленивые, как я, обращаются к платформам CSS, чтобы те решили за них некоторые проблемы. Существует множество платформ, но одной из самых ранних и популярных является Bootstrap компании Twitter. Давайте ею воспользуемся.

Ее можно найти по http://getbootstrap.com/.

Мы ее скачаем и поместим в новую папку с именем *static* в приложении `lists`[3]:

```
$ wget -O bootstrap.zip https://github.com/twbs/bootstrap/releases/download/\
v3.3.4/bootstrap-3.3.4-dist.zip
$ unzip bootstrap.zip
$ mkdir lists/static
$ mv bootstrap-3.3.4-dist lists/static/bootstrap
$ rm bootstrap.zip
```

Bootstrap реализован в виде простой, неперсонализуемой инсталляции в папке *dist*. На данный момент мы будем использовать именно ее, но никогда не делайте этого в условиях реального сайта – классическая платформа Bootstrap легко опознаваема и является ясным сигналом для любого, кто в курсе, что вы не потрудились в стилевом оформлении сайта. Научитесь использовать LESS и менять шрифт по крайней мере! В документации Bootstrap можно отыскать соответствующую информацию, либо обратитесь к неплохому руководству по ссылке[4].

[3] В Windows у вас может не быть утилит `wget` и `unzip`, но я уверен, что вы сможете скачать Bootstrap, распаковать эту платформу и разместить ее содержимое в подпапку *dist* в папке *lists/static/bootstrap*.

[4] См. http://coding.smashingmagazine.com/2013/03/12/customizing-bootstrap/

Наша папка *lists* в итоге будет выглядеть так:

```
$ tree lists
lists
├── __init__.py
├── __pycache__
│   └── [...]
├── admin.py
├── models.py
├── static
│   └── bootstrap
│       ├── css
│       │   ├── bootstrap.css
│       │   ├── bootstrap.css.map
│       │   ├── bootstrap.min.css
│       │   ├── bootstrap-theme.css
│       │   ├── bootstrap-theme.css.map
│       │   └── bootstrap-theme.min.css
│       ├── fonts
│       │   ├── glyphicons-halflings-regular.eot
│       │   ├── glyphicons-halflings-regular.svg
│       │   ├── glyphicons-halflings-regular.ttf
│       │   ├── glyphicons-halflings-regular.woff
│       │   └── glyphicons-halflings-regular.woff2
│       └── js
│           ├── bootstrap.js
│           ├── bootstrap.min.js
│           └── npm.js
├── templates
│   ├── home.html
│   └── list.html
├── tests.py
├── urls.py
└── views.py
```

Если заглянуть в раздел «Getting Started» («С чего начинать») документации Bootstrap[5], то мы увидим, что она требует от нас, чтобы HTML-шаблон содержал что-то вроде этого:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

[5] См. http://getbootstrap.com/getting-started/%23template

```
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap 101 Template</title>
    <!-- Bootstrap -->
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script src="http://code.jquery.com/jquery.js"></script>
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

У нас уже есть два HTML-шаблона. Мы не хотим добавлять целую груду шаблонного кода в каждый из них, так что, похоже, пришло время применить правило «DRY» («Не повторяйся») и объединить все общие части. К счастью, язык шаблонов Django упрощает эту операцию благодаря наследованию шаблонов.

Наследование шаблонов в Django

Предлагаю вашему вниманию небольшой обзор того, чем различаются *home.html* и *list.html*:

```
$ diff lists/templates/home.html lists/templates/list.html
<     <h1>Start a new To-Do list</h1>
<     <form method="POST" action="/lists/new">
---
>     <h1>Your To-Do list</h1>
>     <form method="POST" action="/lists/{{ list.id }}/add_item">
[...]
>     <table id="id_list_table">
>       {% for item in list.item_set.all %}
>         <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
>       {% endfor %}
>     </table>
```

У них разные тексты заголовочных элементов, и в их формах используются разные URL-адреса. Плюс *list.html* имеет дополнительный элемент `<table>`.

Теперь, когда нам ясно, чем они похожи и чем отличаются, можно сделать так, чтобы два шаблона наследовали у общего шаблона, который будет суперклассом. Для начала сделаем копию *home.html*:

```
$ cp lists/templates/home.html lists/templates/base.html
```

Превратим его в основной шаблон, который просто содержит общее стандартное оформление, и выделим «блоки», то есть места, где дочерние шаблоны могут настраивать его под свои нужды:

lists/templates/base.html

```
<html>
  <head>
    <title>To-Do lists</title>
  </head>
  <body>
    <h1>{% block header_text %}{% endblock %}</h1>
    <form method="POST" action="{% block form_action %}{% endblock %}">
      <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
      {% csrf_token %}
    </form>
    {% block table %}
    {% endblock %}
  </body>
</html>
```

Основной шаблон определяет серию областей, именуемых блоками, к которым другие шаблоны могут подключаться и добавлять свое собственное наполнение. Давайте посмотрим, как это работает на практике, изменив *home.html* так, чтобы он наследовал от *base.html*:

lists/templates/home.html

```
{% extends 'base.html' %}

{% block header_text %}Start a new To-Do list{% endblock %}

{% block form_action %}/lists/new{% endblock %}
```

Вы видите, что много шаблонного HTML исчезает, и мы просто концентрируемся на фрагментах, которые хотим настроить. То же самое делаем для *list.html*:

lists/templates/list.html

```
{% extends 'base.html' %}

{% block header_text %}Your To-Do list{% endblock %}

{% block form_action %}/lists/{{ list.id }}/add_item{% endblock %}

{% block table %}
```

```
  <table id="id_list_table">
    {% for item in list.item_set.all %}
      <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
    {% endfor %}
  </table>
{% endblock %}
```

В этом заключается рефакторизация работы шаблонов. Выполняем функциональные тесты повторно, чтобы удостовериться, что ничего не повредили...

```
AssertionError: 107.0 != 512 within 10 delta
```

Разумеется, они по-прежнему добираются точно до того места, что и раньше. Это заслуживает фиксации:

```
$ git diff -b
# -b означает игнорировать пробельные символы, полезно, поскольку мы немного
изменили расстановку отступов html
$ git status
$ git add lists/templates # пока оставить папку static
$ git commit -m "Рефакторизованы шаблоны для использования базового шаблона"
```

Интеграция платформы Bootstrap

Теперь намного проще интегрировать шаблонный код, который от нас требует Bootstrap. Добавлять JavaScript мы пока не будем, только CSS:

lists/templates/base.html (ch08l006)

```
<!DOCTYPE html>
<html lang="en">

  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>To-Do lists</title>
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
[...]
```

Строки и столбцы

Наконец, давайте на практике воспользуемся волшебством Bootstrap! Вам придется самостоятельно почитать документацию, но вы должны

научиться использовать комбинацию сеточной системы и класса `text-center`, чтобы получить то, что мы хотим:

lists/templates/base.html (ch08l007)

```
<body>
  <div class="container">

    <div class="row">
      <div class="col-md-6 col-md-offset-3">
        <div class="text-center">
          <h1>{% block header_text %}{% endblock %}</h1>
          <form method="POST" action="{% block form_action %}{% endblock %}">
            <input name="item_text" id="id_new_item"
                   placeholder="Enter a to-do item" />
            {% csrf_token %}
          </form>
        </div>
      </div>
    </div>

    <div class="row">
      <div class="col-md-6 col-md-offset-3">
        {% block table %}
        {% endblock %}
      </div>
    </div>

  </div>
</body>
```

Если вы никогда не видели тег HTML, разбитый на несколько строк, то этот элемент `<input>` может вас немного шокировать. Это определенно допустимо, но от вас никто не требует его использовать, если вы находите это оскорбительным. ;)

Неспешно просмотрите документацию Bootstrap, если вы ее никогда не видели раньше. Это целая тележка, до краев наполненная полезными инструментами для применения на сайте.

Это работает?

```
AssertionError: 107.0 != 512 в пределах дельты в 10 единиц
```

Хм. Нет. Почему же наш CSS не загружается?

Статические файлы в Django

Для того, чтобы справляться со статическими файлами, программной инфраструктуре Django (да и любому веб-серверу) нужно знать две вещи:

1. Когда URL-запрос предназначен для статического файла, а не для некоего HTML, который будет роздан через функцию представления.
2. Где найти статический файл, который хочет пользователь.

Другими словами, статические файлы являются результатом преобразования URL-адресов в имена дисковых файлов.

В отношении первого пункта Django позволяет нам определить префикс URL-адреса, говорящий, что любой URL, который начинается с этого префикса, нужно рассматривать как запрос на статические файлы. По умолчанию таким префиксом является `/static/`. Он определен в *settings.py*:

superlists/settings.py

```
[...]

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/dev/howto/static-files/

STATIC_URL = '/static/'
```

Все остальные части настроек, которые мы добавим в этот раздел, касаются второго пункта: нахождение фактических статических файлов на диске.

С учетом того, что мы используем сервер разработки Django (`manage.py runserver`), можно опираться на Django, чтобы та волшебным образом находила статические файлы за нас – Django просто будет заглядывать в каждую подпапку одного из наших приложений с именем *static*.

Теперь вы понимаете, почему мы помещаем все статические файлы платформы Bootstrap в *lists/static*. Итак, почему же они в данный момент не работают? Потому что мы не используем префикс URL-адреса `/static/`. Взгляните еще раз на ссылку на CSS в *base.html*:

```
<link href="css/bootstrap.min.css" rel="stylesheet">
```

Чтобы заставить это работать, ее нужно изменить на:

lists/templates/base.html

```
<link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet">
```

Когда команда `runserver` видит запрос, она знает, что он предназначен для статического файла, потому что запрос начинается с префикса

`/static/`. И тогда команда пытается найти файл с именем `bootstrap/css/bootstrap.min.css`, заглядывая во все папки приложений в поисках подпапок с именем *static*, и она найдет его в *lists/static/bootstrap/css/bootstrap.min.css*.

Если вы посмотрите вручную, то увидите, что все происходит как на рис. 8.2.

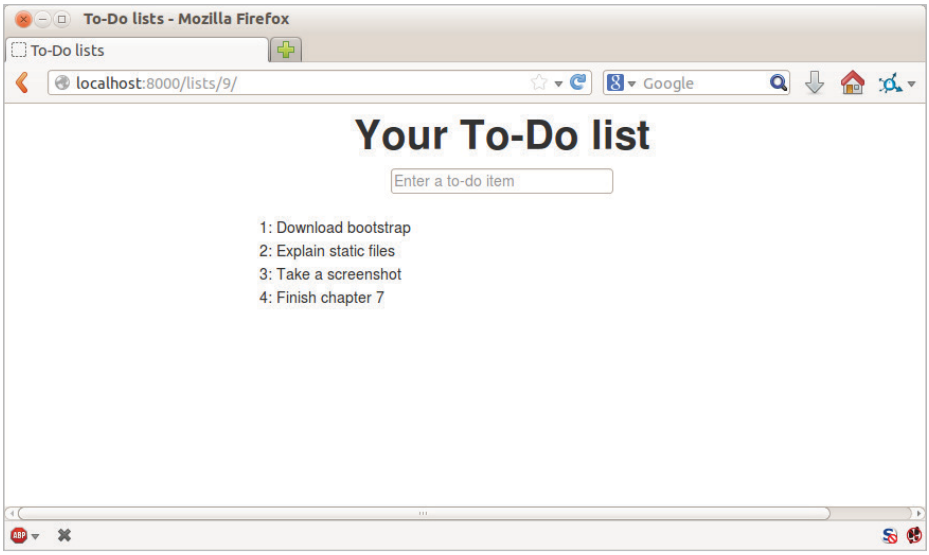


Рис. 8.2. Сайт выглядит немного лучше...

Переход на StaticLiveServerTestCase

Правда, если вы выполните ФТ, то он не сработает:

```
AssertionError: 107.0 != 512 within 10 delta
```

Это потому, что, несмотря на то что `runserver` автоматически находит статические файлы, `LiveServerTestCase` этого не делает. Хотя не стоит волноваться – разработчики Django создали еще более волшебный тестовый класс под названием `StaticLiveServerTestCase` (см. документацию[6]).

Давайте перейдем на него:

functional_tests/tests.py

```
@@ -1,14 +1,14 @@
-from django.test import LiveServerTestCase
+from django.contrib.staticfiles.testing import StaticLiveServerTestCase
 from selenium import webdriver
 from selenium.common.exceptions import WebDriverException
 from selenium.webdriver.common.keys import Keys
```

```
 import time

 MAX_WAIT = 10

-class NewVisitorTest(LiveServerTestCase):
+class NewVisitorTest(StaticLiveServerTestCase):

     def setUp(self):
```

И теперь он найдет новый CSS, что заставит наш тест пройти успешно:

```
$ python manage.py test functional_tests
Creating test database for alias 'default'...
...
 ----------------------------------------------------------------------
Ran 3 tests in 9.764s
```

В этой точке пользователи Windows могут увидеть некоторые (безопасные, но отвлекающие) сообщения об ошибках, в которых говорится `socket.error: [WinError 10054] An existing connection was forcibly closed by the remote host` (Ошибка подключения: существующее соединение было принудительно закрыто удаленным узлом в сети). Чтобы от них избавиться в `tearDown`, прямо перед `self.browser.quit()`, добавьте инструкцию `self.browser.refresh()`. Данный вопрос отслеживается в приведенном по ссылке сообщении о дефекте в трекере Django[7].

Ура!

Использование компонентов Bootstrap для улучшения внешнего вида сайта

Давайте убедимся, что у нас получится сделать еще лучше, если мы воспользуемся некоторыми другими инструментами в арсенале платформы Bootstrap.

Класс jumbotron

Bootstrap имеет класс под названием jumbotron для вещей, предназначенных для визуального выделения элементов страницы. Предлагаю воспользоваться им, чтобы увеличить главный верхний заголовок страницы и входную форму:

[7] См. https://code.djangoproject.com/ticket/21227

lists/templates/base.html (ch08l009)

```
<div class="col-md-6 col-md-offset-3 jumbotron">
  <div class="text-center">
    <h1>{% block header_text %}{% endblock %}</h1>
      <form method="POST" action="{% block form_action %}{% endblock %}">
        [...]
```

Занимаясь дизайном и макетом сайта, лучше всего иметь открытое окно, где можно периодически нажимать значок обновления. Используйте `python manage.py runserver`, чтобы запустить сервер разработки, и затем перейдите на http://localhost:8000, чтобы видеть свою работу по мере продвижения.

Большие поля ввода

Класс `jumbotron` является хорошим началом, но теперь поле ввода имеет крошечный текст по сравнению со всем остальным. К счастью, классы управления формой в Bootstrap позволяют сделать поле ввода большим:

lists/templates/base.html (ch08l010)

```
<input name="item_text" id="id_new_item"
       class="form-control input-lg"
       placeholder="Enter a to-do item" />
```

Стилистическое оформление таблицы

Текст таблиц теперь тоже выглядит слишком мелким по сравнению с остальной частью страницы. Добавление класса Bootstrap `table` улучшает ситуацию:

lists/templates/list.html (ch08l011)

```
<table id="id_list_table" class="table">
```

Использование собственного CSS

Я хочу слегка сместить поле ввода относительно текста заголовка. В Bootstrap нет готовых решений для такой операции, а значит, мы сделаем это самостоятельно. Для этого потребуется определить наш собственный файл CSS:

lists/templates/base.html

```
[...]
  <title>To-Do lists</title>
  <link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet">
```

```
  <link href="/static/base.css" rel="stylesheet">
</head>
```

Создаем файл в *lists/static/base.css* с новым правилом CSS. Мы будем использовать `id` элемента input, `id_new_item`, чтобы найти его на странице и придать ему некоторую стилизацию:

lists/static/base.css

```
#id_new_item {
    margin-top: 2ex;
}
```

Мне потребовалось несколько попыток, но результатом я волне доволен (рис. 8-3).

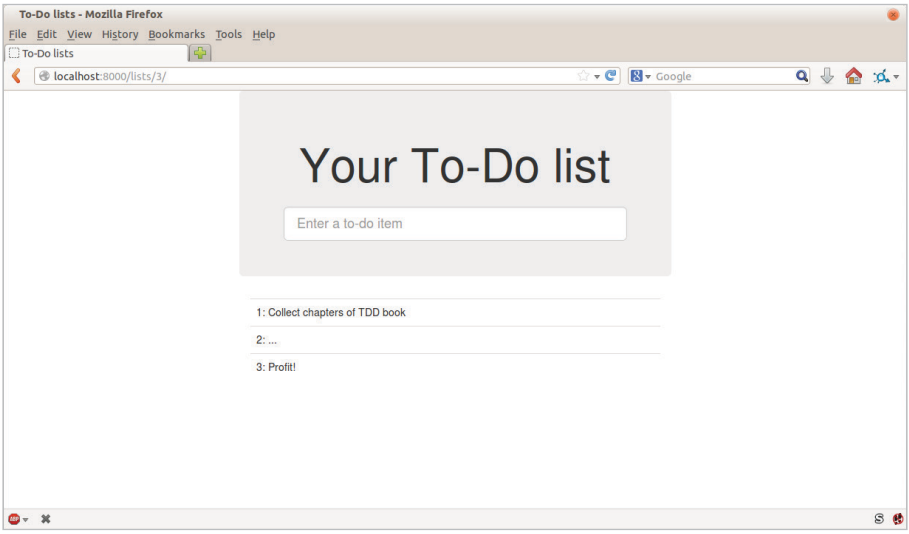


Рис. 8-3. Страница приложения `lists` с большими блочными элементами...

Если вы захотите пойти дальше с настройкой Bootstrap под свои нужды, вам нужно влезть в компиляцию LESS. Я определенно рекомендую не торопиться и заняться этим потом. LESS и другие подобные псевдо-CSS вещи, например SASS, являются большим усовершенствованием простого CSS и полезным инструментом, даже если вы не используете Bootstrap. В этой книге я не буду их рассматривать, но вы можете найти соответствующие ресурсы в Интернете. Один из них указан в сноске[8].

Последний прогон функциональных тестов, чтобы убедиться, что все по-прежнему работает как надо:

```
$ python manage.py test functional_tests
[...]
```

[8] См. http://coding.smashingmagazine.com/2013/03/12/customizing-bootstrap/

```
...
 ----------------------------------------------------------------------
Ran 3 tests in 10.084s

OK
```

Точка! Определенно самое время для фиксации:

```
$ git status # изменяет tests.py, base.html, list.html + не отслеживаемые
lists/static
$ git add .
$ git status # теперь покажет все добавления за счет bootstrap
$ git commit -m "Использован Bootstrap для улучшения макета"
```

О чем мы умолчали: collectstatic и другие статические каталоги

Ранее мы видели, что сервер разработки Django волшебным образом находит все ваши статические файлы в папках приложения и раздает их вам. Это прекрасно во время разработки, но когда вы работаете на реальном веб-сервере, вам не хотелось бы, чтобы Django раздавал ваш статический контент – использование Python для раздачи необработанных файлов приводит к медленной и неэффективной работе, и такой веб-сервер, как Apache либо Nginx, могут сделать это все за вас. Вы можете даже закачать все свои статические файлы на CDN, вместо того чтобы размещать их самостоятельно.

По этим причинам нужно иметь возможность собирать все статические файлы из всех папок приложения и копировать в одно готовое к развертыванию место. Для этого предназначена команда `collectstatic`.

Место назначения, то есть куда направляются собранные статические файлы, определяется в *settings.py* как `STATIC_ROOT`. В следующей главе мы чуть подробнее займемся развертыванием, а сейчас просто поэкспериментируем. Мы поменяем его значение на папку недалеко от нашего репозитория – это будет папка сразу рядом с главной исходной папкой:

```
workspace
|   ├── superlists
|   |   ├── lists
|   |   |   ├── models.py
|   |   |
|   |   ├── manage.py
|   |   ├── superlists
|   |
|   ├── static
|   |   ├── base.css
|   |   ├── etc...
```

Логика в том, что папка со статическими файлами не должна быть частью вашего репозитория – мы не хотим ее ставить под контроль системы управления версиями, потому что это копия всех файлов внутри *lists/static*.

Вот аккуратный способ указать расположение этой папки относительно основного каталога проекта:

superlists/settings.py (ch08l018)

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.11/howto/static-files/

STATIC_URL = '/static/'
STATIC_ROOT = os.path.abspath(os.path.join(BASE_DIR, '../static'))
```

Взгляните на верхнюю часть файла настроек, и вы увидите, как переменная `BASE_DIR` с готовностью определяется для нас с использованием переменной `__file__` (которая сама по себе является действительно полезным встроенным инструментом Python).

Попробуем выполнить `collectstatic`:

```
$ python manage.py collectstatic
[...]
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap-theme.css'
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap.min.css'

76 static files copied to '/.../static'.
```

И если мы посмотрим в *.../static*, то найдем все наши файлы CSS:

```
$ tree ../static/
../static/
├── admin
│   ├── css
│   │   ├── base.css

[...]

│               └── xregexp.min.js
├── base.css
└── bootstrap
├── css
│   ├── bootstrap.css
│   ├── bootstrap.css.map
│   ├── bootstrap.min.css
│   ├── bootstrap-theme.css
│   ├── bootstrap-theme.css.map
```

```
│   └── bootstrap-theme.min.css
├── fonts
│   ├── glyphicons-halflings-regular.eot
│   ├── glyphicons-halflings-regular.svg
│   ├── glyphicons-halflings-regular.ttf
│   ├── glyphicons-halflings-regular.woff
│   └── glyphicons-halflings-regular.woff2
└── js
├── bootstrap.js
├── bootstrap.min.js
└── npm.js

14 directories, 76 files
```

Утилита `collectstatic` также подхватила все CSS для сайта администратора. Сайт администратора – один из мощных функциональных механизмов Django, и мы узнаем о нем все, но только не сегодня, так как мы еще не готовы его использовать. Поэтому пока отключим его:

superlists/settings.py

```
INSTALLED_APPS = [
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
]
```

И пробуем еще раз:

```
$ rm -rf ../static/
$ python manage.py collectstatic --noinput
Copying '/.../superlists/lists/static/base.css'
[...]
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap-theme.css'
Copying '/.../superlists/lists/static/bootstrap/css/bootstrap.min.css'

15 static files copied to '/.../static'.
```

Намного лучше.

Теперь мы знаем, как собрать все статические файлы в одну папку, где веб-сервер их легко найдет. В следующей главе мы узнаем об этом все, в том числе как это тестировать!

А пока давайте сохраним изменения, внесенные в *settings.py*:

```
$ git diff # должна показать изменения в settings.py*
$ git commit -am "Установлен STATIC_ROOT в settings и деактивирован admin"
```

Несколько вещей, которые не удались

Разумеется, это был всего лишь ураганный тур по стилистическому оформлению и CSS, так что некоторые темы мне не удалось раскрыть. Вот несколько кандидатов для дальнейшего исследования:

- Настройка под свои нужды платформы Bootstrap при помощи LESS или SASS.
- Шаблонный тег `{% static %}` для более «сухих» (DRY от «Don't Repeat Yourself» («Не повторяйся»)) и менее жестко кодированных URL-адресов.
- Инструменты пакетирования на стороне клиента, такие как `npm` и `bower`.

Резюме: о тестировании дизайна и макета

Если кратко, то вам не следует писать тесты для дизайна и макета сайта как таковые. Это слишком смахивает на тестирование константы, и тесты, которые вы напишете, нередко будут получаться хрупкими.

С учетом сказанного реализация дизайна и макета сопряжена с чем-то довольно неоднозначным: CSS и статическими файлами. В результате большую значимость имеет минимальная проверка на токсичность, которая проверяет, что ваши статические файлы и CSS работают. Как мы увидим в следующей главе, это помогает засечь проблемы при развертывании программного кода в производственной среде.

Аналогичным образом: если отдельная часть стилевого оформления требует много клиентского программного кода JavaScript (у меня ушло немало времени на динамическое изменение размеров), то вам наверняка захочется провести для этого немного тестов.

Старайтесь писать минимальные тесты, которые подтвердят, что дизайн и макет сайта работают как надо, без тестирования того, что они собой представляют фактически. Стремитесь оставаться в позиции, где вы можете свободно вносить изменения в дизайн и макет без необходимости все время возвращаться и корректировать тесты.

Глава 9

Тестирование развертывания с использованием промежуточного сайта

Все забава да игра, пока не приходит время вводить в эксплуатацию.
– Devops Borat[1]

Пора развернуть первую версию нашего сайта и сделать его публичным. Говорят, если ждать, пока не почувствуешь себя готовым поставить продукт, значит, ждать слишком долго.

Годится ли наш сайт для использования? Действительно ли он лучше, чем ничего? Сможем ли мы составлять на нем списки? Да, да и да.

Нет, вы еще не можете регистрироваться в системе. Нет, вы не можете отмечать задачи как завершенные. Но разве нам все это нужно? Не совсем – никогда не знаешь, что именно пользователи собираются делать с сайтом, пока он не попадется им в руки. Мы полагаем, что пользователи хотят составлять на сайте списки неотложных дел, но, возможно, они на самом деле хотят использовать его для создания списков «10 лучших рыбных мест для ловли на муху», для которых совсем не нужна функция «пометить как завершенное». Мы не узнаем, пока не разместим его.

В этой главе мы пройдем процедуру развертывания сайта на реальном, работающем веб-сервере.

Вы можете испытать желание пропустить эту главу – в ней много чрезвычайно сложного материала, и, возможно, вы посчитаете, что подписывались совсем не на это. Но я *настоятельно* призываю вас дать ей шанс. Это один из разделов книги, которым я доволен больше всего и о котором мне люди часто пишут, говоря, что получили удовольствие от его прочтения.

Если вы никогда прежде не выполняли развертывание сайта на сервере, то этот раздел сорвет для вас покров тайны с целого мира, и нет ничего лучшего, чем видеть, как сайт работает в реальном Интернете. Дайте ему

[1] См. https://twitter.com/DEVOPS_BORAT/status/192271992253190144

модное имя, типа расхожего словечка DevOps, если это убедит вас, что оно того стоит.

Почему бы не сообщить мне, когда ваш сайт заработает, и не отправить мне его URL-адрес? От этого всегда становится тепло и уютно... obeythetestinggoat@gmail.com.

TDD и опасные зоны развертывания

Развертывание сайта на работающем веб-сервере относится к разряду сложных и заковыристых тем. Часто можно услышать отчаянный вопль: *«Но он же работает на моей машине!»*.

Некоторые самые небезопасные зоны процесса развертывания включают в себя:

Статические файлы (CSS, JavaScript, изображения и т. д.)

Веб-серверы обычно требуют наличия специальной конфигурации, чтобы их раздавать.

База данных

Могут быть вопросы, связанные с полномочиями и путями доступа. Нужно осторожно подходить к сохранению данных между развертываниями.

Зависимости

Мы должны удостовериться, что программные пакеты, на которые опирается наше ПО, установлены на сервере и имеют правильные версии.

Однако для каждого из этих вопросов существуют решения:

- *Промежуточный сайт* на точно такой же инфраструктуре, что и производственный, поможет проверить развертывание и сделать все верно, прежде чем мы перейдем на реальный сайт.
- Мы также можем *выполнять функциональные тесты* для *промежуточного сайта*. Так мы убедимся, что на сервере размещен верный программный код и нужные пакеты, и, поскольку теперь у нас имеется проверка макета сайта на токсичность, мы будем знать, что CSS загружается правильно.
- Точно так же, как и на нашем собственном ПК, на сервере удобно использовать *Virtualenv* для управляющих пакетов и зависимостей, когда вы, возможно, будете выполнять более одного приложения Python.

- И наконец, *автоматизация, автоматизация, еще раз автоматизация*. С помощью автоматизированного сценария развертывания новых версий и одинакового сценария развертывания на промежуточном и производственном серверах, мы можем обнадежить себя, что промежуточный сервер во многом аналогичен реальному[2].

На нескольких следующих страницах я собираюсь пройти процедуру развертывания. Она не претендует на *идеал*, поэтому не принимайте ее как лучшую практику применения. Она лишь демонстрирует типы проблем, связанных с развертыванием, и показывает место, куда вписывается тестирование.

Обзор глав о развертывании

В трех следующих главах очень много материала, поэтому я подготовил краткий обзор, чтобы было удобнее ориентироваться в нем.

Данная глава: приведение в рабочее состояние

- Адаптировать ФТ таким образом, чтобы они могли работать с промежуточным сервером.
- Запустить сервер, установить необходимое программное обеспечение и указать на нем промежуточный и реальный домены.
- Закачать код на сервер при помощи Git.
- Попытаться получить черновую версию сайта, работающего на промежуточном домене, используя сервер разработки Django.
- Выполнить ручную настройку виртуальной среды virtualenv на сервере (без обертки `virtualenvwrapper`) и убедиться, что база данных и статические файлы работают.
- По мере продвижения мы продолжим выполнять ФТ, который будет сообщать, что работает, а что нет.

Следующая глава: перемещение в готовую к эксплуатации конфигурацию

- Переместить из черновой версии в готовую к эксплуатации конфигурацию; прекратить использовать сервер разработки Django; настроить приложение, чтобы оно запускалось автоматически во время начальной загрузки; установить DEBUG в `False` и т. д.

Третья глава о развертывании: автоматизация развертывания

1. Когда у нас будет рабочая конфигурация, мы напишем сценарий для автоматизации процесса, который только что прошли вручную, чтобы в будущем сайт можно было развертывать автоматически.
2. Наконец, воспользуемся данным сценарием для развертывания производственной версии сайта на реальном домене.

[2] То, что я называю промежуточным сервером, некоторые называют сервером разработки, другие же предпочитают термин «предпроизводственные серверы». Как бы мы его ни называли, суть в том, чтобы иметь какое-то место, где можно испытать программный код в среде, которая аналогична реальному производственному серверу.

Как всегда, начинайте с теста

Давайте немного адаптируем наши функциональные тесты, чтобы они могли выполняться на промежуточном сайте. Для этого слегка поправим параметр, обычно использующийся для изменения адреса, по которому выполняется временный сервер теста:

functional_tests/tests.py (ch08l001)

```
import os
[...]

class NewVisitorTest(StaticLiveServerTestCase):
    '''тест нового посетителя'''

    def setUp(self):
        '''установка'''
        self.browser = webdriver.Firefox()
        staging_server = os.environ.get('STAGING_SERVER')  ❶
        if staging_server:
            self.live_server_url = 'http://' + staging_server  ❷
```

Если вы помните, я сказал, что `LiveServerTestCase` имеет определенные ограничения. Одно из них: он всегда предполагает, что вы хотите использовать его собственный тестовый сервер, который он делает доступным по `self.live_server_url`. Временами у меня по-прежнему возникает потребность это делать, но я хочу иметь возможность выборочно сообщать ему, чтобы он не беспокоился, и использовать реальный сервер.

❶ Я решил это сделать с помощью переменной окружения под названием `STAGING_SERVER`.

❷ Вот она, наша правка: мы заменяем `self.live_server_url` на адрес реального сервера.

Проверяем, что означенная правка ничего не нарушила, путем выполнения функциональных тестов обычным образом:

```
$ python manage.py test functional_tests
[...]
Ran 3 tests in 8.544s

OK
```

Теперь можно испытать их относительно URL-адреса промежуточного сервера. Я планирую разместить промежуточный сервер по адресу *superlists-staging.ottg.eu*:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests

======================================================================
FAIL: test_can_start_a_list_for_one_user
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 49, in
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
AssertionError: 'To-Do' not found in 'Domain name registration | Domain names
| Web Hosting | 123-reg'
[...]

======================================================================
FAIL: test_multiple_users_can_start_lists_at_different_urls
(functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/tests.py", line 86,
in test_layout_and_styling
    inputbox = self.browser.find_element_by_id('id_new_item')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_new_item"]
[...]
======================================================================
FAIL: test_layout_and_styling (functional_tests.tests.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_new_item"]
[...]

Ran 3 tests in 19.480s:

FAILED (failures=3)
```

Если в Windows вы видите ошибку, которая сообщает что-то вроде: «STAGING_SERVER is not recognized as a command» («STAGING_SERVER не опознан в качестве команды»), вероятно, это связано с тем, что вы не используете Git-Bash. Обратитесь еще раз к разделу «Предпосылки и предположения» в начале книги.

Мы видим, что оба теста не проходят, как и ожидалось, поскольку фактически я еще не настроил домен. По сути, из результатов первой обратной трассировки видно, что тест практически заканчивается на домашней странице регистратора доменных имен.

Похоже, ФТ тестирует правильные вещи, поэтому давайте фиксировать:

```
$ git diff # должна показать изменения в functional_tests.py
$ git commit -am "Настроен исполнитель ФТ для тестирования промежуточного
сервера"
```

Не используйте export для установки переменной окружения *STAGING_SERVER*; в противном случае все ваши последующие прогоны тестов в терминале будут выполняться для промежуточного сервера (и это может стать причиной недоразумений, если вы этого не ожидаете). Лучше всего устанавливать ее в явном виде каждый раз, когда вы выполняете функциональные тесты..

Получение доменного имени

В данной точке нам потребуется пара доменных имен – они могут быть субдоменами одного домена. Я собираюсь использовать *superlists.ottg.eu* и *superlists-staging.ottg.eu*. Если вы еще не владеете доменом, то самое время его зарегистрировать! Опять-таки, это то, что действительно нужно сделать. Если вы никогда прежде не регистрировали домен, то просто выберите любого старого регистратора и приобретите самое дешевое имя – оно обойдеется примерно в $5. Можно даже найти бесплатные. Я обещаю, что вид вашего сайта на реальном веб-сайте вызовет непередаваемые ощущения.

Ручное обеспечение работы сервера для размещения сайта

Мы можем разбить развертывание на две задачи:

- *Обеспечение работы* нового сервера, чтобы на нем можно было разместить программный код.
- *Развертывание* новой версии программного кода на существующем сервере.

Некоторым нравится для каждого развертывания использовать совершенно новый сервер – это именно то, что мы делаем в PythonAnywhere. Правда, это нужно только для более крупных, более сложных сайтов либо

для существенных изменений в имеющемся. Для такого простого сайта, как наш, имеет смысл эти две задачи разделить. И, хотя в конечном счете мы хотим, чтобы обе задачи были полностью автоматизированы, пока мы можем обойтись ручной системой обеспечения.

Читая настоящую главу, вы должны осознавать, что обеспечение работы варьируется от случая к случаю и как результат можно найти совсем немного универсальных методических рекомендаций по развертыванию. Поэтому, вместо того чтобы пытаться запомнить конкретные особенности процесса, постарайтесь понять, что и для чего я делаю, чтобы вы могли применить такой же образ мыслей в конкретных обстоятельствах, с которыми вы столкнетесь в будущем.

Выбор места размещения сайта

В наши дни существует уйма разных решений, но все они в широком смысле распадаются на два лагеря:

- Выполнение вашего собственного (возможно, виртуального) сервера.
- Использование платформы как услуги (PaaS), предлагаемой такими службами, как Heroku, OpenShift или PythonAnywhere.

PaaS дает множество преимуществ, особенно для небольших сайтов, и я настоятельно рекомендую их изучить. Правда, в этой книге мы не будем использовать PaaS, по нескольким причинам. Во-первых, есть конфликт интересов: я считаю PythonAnywhere самой лучшей платформой и должен напомнить, что я там работаю. Во-вторых, все предложения PaaS очень различаются и процедуры развертывания на каждой из платформ сильно варьируются – знание одной не обязательно скажет вам что-то о других. Любая из них может радикально изменить свой процесс либо просто обанкротиться к тому времени, когда вы начнете читать эту книгу.

Вместо этого мы изучим лишь крошечную часть хорошего старомодного администратора сервера, включая SSH и конфигурацию веб-сервера. Они вряд ли когда-либо исчезнут, и вашу небольшую осведомленность о них уважительно оценят седые динозавры.

Я только лишь попытался настроить сервер таким образом, чтобы он очень походил на среду, которую вы получаете от PaaS, и поэтому вы должны суметь применить информацию, которую мы изучаем в разделе о развертывании, независимо от того, какой вариант обеспечения работы вы выберете.

Запуск сервера

Я не собираюсь вам диктовать, как это делать, – подойдет любое решение, будь то Amazon AWS, Rackspace, Digital Ocean, сервер в вашем собст-

венном центре обработки и хранения данных или Raspberry Pi на полке под лестницей, если:

- Ваш сервер работает под ОС Ubuntu 16.04 (также известной под именем Xenial/LTS).
- У вас есть к нему root-доступ.
- Он имеет общий доступ в Интернет.
- Вы можете к нему подключаться по протоколу SSH.

Я рекомендую Ubuntu в качестве дистрибутива, потому что он позволяет легко инсталлировать Python 3.6 и имеет некоторые специфические методы конфигурирования Nginx, которые собираюсь я использовать в дальнейшем. Если вы знаете, что делаете, возможно, вы остановитесь на чем-то еще, – действуйте по своему усмотрению.

Если вы никогда не запускали сервер Linux прежде и вы абсолютно не представляете, с чего начинать, я подготовил очень краткое руководство, которое приведено в сноске[3].

Некоторые читатели, добравшись до этой главы, испытывают желание пропустить фрагмент текста про домен и «получение реального сервера», а просто использовать виртуальную машину (VM) на своем ПК. Не делайте этого. Это не то же самое, и вы испытаете больше трудностей, когда будете следовать инструкциям, которые и так достаточно сложны по своей сути. Если вас волнует стоимость, поищите – и вы наверняка найдете бесплатные опции для обоих случаев. Напишите мне по электронной почте, если нуждаетесь в дополнительных советах – я всегда рад помочь.

Учетные записи пользователей, SSH и полномочия

В приведенных ниже инструкциях я исхожу из того, что у вас создана учетная запись без доступа root, но с полномочиями sudo, поэтому каждый раз, когда мы должны сделать что-то, что требует root-доступа, мы используем sudo. Я показываю это в явном виде в инструкциях ниже.

Мой пользователь называется elspeth, но вы можете использовать имя вашего пользователя, каким бы оно ни было!

Инсталляция Nginx

Нам потребуется веб-сервер. Все крутые ребята нынче используют Nginx, поэтому мы тоже будем его использовать. Потратив много лет на борьбу с

[3] См. https://github.com/hjwp/Book-TDD-Web-Dev-Python/blob/master/server-quickstart.md

Apache, я могу сказать, что это благословенное облегчение с точки зрения удобочитаемости его файлов конфигурации по крайней мере!

Установка Nginx на моем сервере была всего лишь вопросом выполнения команды `apt-get`, затем я увидел стандартный экран Hello World от Nginx:

```
elspeth@server:$ sudo apt-get install nginx
elspeth@server:$ sudo systemctl start nginx
```

(Вам, возможно, сначала придется выполнить `apt-get update` и/или `apt-get upgrade`.)

Обратите внимание на `elspeth@server` в распечатках командных строк этой главы. Они говорят о том, что команды должны выполняться на сервере, в отличие от команд, которые вы выполняете на собственном ПК.

В этой точке вы должны быть в состоянии перейти на IP-адрес сервера и увидеть страницу «Welcome to nginx» как на рис. 9-1.

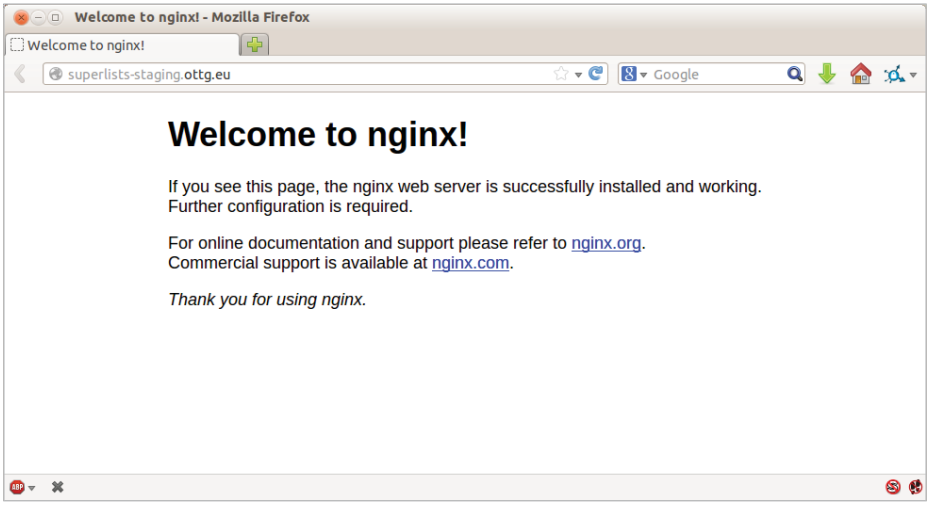


Рис. 9-1. Nginx – он работает!

Если вы ее не видите, это может быть из-за того, что у вашего брандмауэра не открыт порт 80. На AWS, например, может потребоваться сконфигурировать «группу безопасности», чтобы сервер открыл порт 80.

Инсталляция Python 3.6

При написании данной главы Python 3.6 не был доступен в стандартных репозиториях в Ubuntu, но имелся в предоставленном пользователем Deadsnakes, PPA[4]. Вот как мы его устанавливаем.

Хотя у нас есть root-доступ, давайте убедимся, что сервер имеет основные компоненты программного обеспечения, в котором мы нуждаемся на системном уровне: Python, Git, pip и virtualenv.

```
elspeth@server:$ sudo add-apt-repository ppa:fkrull/deadsnakes
elspeth@server:$ sudo apt-get update
elspeth@server:$ sudo apt-get install python3.6 python3.6-venv
```

И раз уж мы тут, убедимся, что git тоже установлен.

```
elspeth@server:$ sudo apt-get install git
```

Конфигурирование доменов для промежуточного и реального серверов

Мы не хотим все время возиться с IP-адресами, поэтому нам нужно указать серверу промежуточный и реальный домены. У моего регистратора доменных имен экран управления именами выглядит как показано на рис. 9-2.

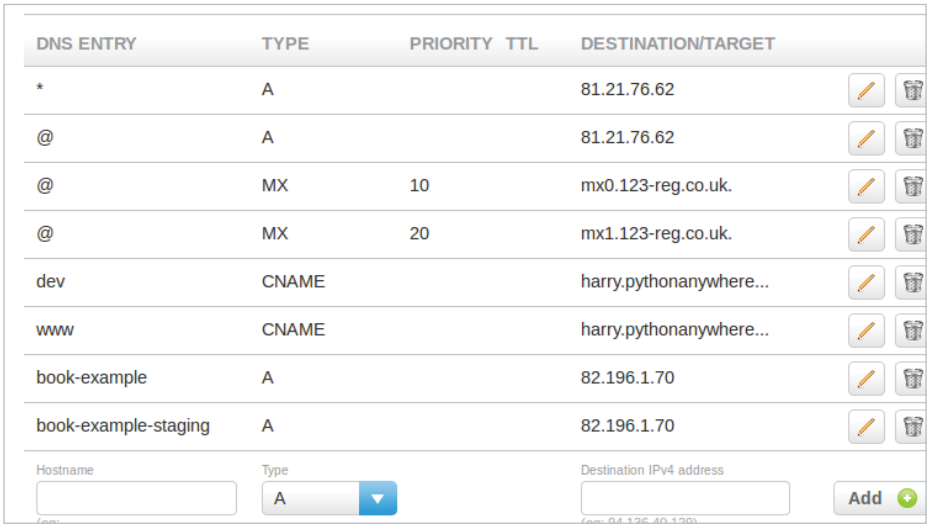


Рис. 9-2. Настройка доменного имени

[4] См. https://launchpad.net/~fkrull/+archive/ubuntu/deadsnake

В системе DNS указание домена на конкретный IP-адрес называется A-записью[5]. Все регистраторы немного различаются, но несколько переходов по страницам приведет вас к нужному экрану.

Использование ФТ для подтверждения, что домен работает и Nginx выполняется

Чтобы убедиться, что все работает, можно повторно выполнить функциональные тесты и увидеть, что их сообщения о неполадках немного изменились – в частности, одно из них теперь должно упомянуть Nginx:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_new_item"]
[...]
AssertionError: 'To-Do' not found in 'Welcome to nginx!'
```

Прогресс! Похлопайте себя по спине и угостите себя чашечкой чая с пончиком[6].

Развертывание исходного кода вручную

На следующем шаге нам нужно привести копию промежуточного сайта в рабочее состояние и проверить, что мы можем заставить Nginx и Django взаимодействовать друг с другом. Таким образом мы переходим от этапа обеспечения к этапу развертывания, поэтому должны подумать, как автоматизировать этот процесс.

Одно из эмпирических правил, которое позволяет отличить этап обеспечения работы от этапа развертывания, гласит, что для первого чаще нужны полномочия root, которые не нужны для второго.

Нам нужен каталог, чтобы разместить исходный код. Мы разместим его где-нибудь в домашней папке нашего пользователя без полномочий root; в моем случае это */home/elspeth* (такая конфигурация, скорее всего, будет в любой системе совместного хостинга, но в любом случае всегда выполняй-

[5] A-запись (адресная запись) – запись DNS, позволяющая сопоставить доменное имя с IP-адресом сервера. – *Прим. перев.*

[6] См. https://en.wikipedia.org/wiki/Digestive_biscuit

те веб-приложения от лица пользователя, без полномочий root). Я собираюсь выполнить настройку своих сайтов следующим образом:

```
/home/elspeth
├── sites
│   ├── www.live.my-website.com
│   │ ├── database
│   │ │   └── db.sqlite3
│   │ ├── source
│   │ │   ├── manage.py
│   │ │   ├── superlists
│   │ │   ├── etc...
│   │ │
│   │ ├── static
│   │ │   ├── base.css
│   │ │   ├── etc...
│   │ │
│   │ └── virtualenv
│   │     ├── lib
│   │     ├── etc...
│   │
│   ├── www.staging.my-website.com
│   │   ├── database
│   │   ├── etc...
```

Каждый сайт (промежуточный, реальный или любой другой) имеет свою папку. Внутри нее есть отдельная папка для исходного кода, базы данных и статических файлов. Логика проста: в то время как исходный код может меняться от одной версии сайта к другой, база данных останется прежней. Статическая папка находится в том же месте относительно вышеупомянутой папки *../static*, которую мы организовали в конце предыдущей главы. Наконец, virtualenv тоже получает собственную подпапку (на сервере нет необходимости использовать `virtualenvwrapper`, мы создадим virtualenv вручную).

Корректировка расположения базы данных

Сначала нужно изменить местоположение базы данных в *settings.py* и удостовериться, что мы можем заставить ее работать на нашем локальном ПК:

superlists/settings.py (ch08l003)

```
# Создать пути внутри проекта, как тут: os.path.join(BASE_DIR, ...)
import os
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

```
[...]

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, '../database/db.sqlite3'),
    }
}
```

Проверьте, как определен `BASE_DIR` в *settings.py*. Обратите внимание, что сначала выполняется `abspath` (то есть вложенная функция). Всегда следуйте этой схеме, когда работаете с файловыми путями, иначе начнут происходить странности в зависимости от того, как импортирован файл. Спасибо Green Nathan[7] за эту подсказку!

Теперь попробуем работу локально:

```
$ mkdir ../database
$ python manage.py migrate --noinput
Operations to perform:
Apply all migrations: auth, contenttypes, lists, sessions
Running migrations:
[...]
$ ls ../database/
db.sqlite3
```

Кажется, работает. Давайте это зафиксируем:

```
$ git diff # должна показать изменения в settings.py
$ git commit -am "Перемещена база данных sqlite за пределы главного дерева
исходного кода"
```

Чтобы переместить исходный код на сервер, мы будем использовать Git и один из сайтов обмена исходным кодом. Если вы еще это не сделали, поместите свой код на GitHub, BitBucket или аналогичный. Все они имеют превосходные инструкции для новичков о том, как это делается.

Вот некоторые команды оболочки bash, которые все это организуют. Если вы с ней не знакомы, обратите внимание на команду `export`, которая позволяет мне задать локальную переменную в bash:

```
elspeth@server:$ export SITENAME=superlists-staging.ottg.eu
elspeth@server:$ mkdir -p ~/sites/$SITENAME/database
```

[7] См. https://github.com/CleanCut/green

```
elspeth@server:$ mkdir -p ~/sites/$SITENAME/static
elspeth@server:$ mkdir -p ~/sites/$SITENAME/virtualenv
# вам следует заменить URL в следующей строке на URL вашего репозитория
elspeth@server:$ git clone https://github.com/hjwp/book-example.git \
~/sites/$SITENAME/source
Resolving deltas: 100% [...]
```

Жизнь переменной bash, определенной при помощи `export`, ограничена консольным сеансом. Если вы выйдете из сервера и снова зарегистрируетесь, то вам потребуется ее переопределить. Такая ситуация чревата, потому что Bash не получит ошибку, а просто заменит переменную пустой строкой, что приведет к странным результатам... Если сомневаетесь, выполните быструю команду **`echo $SITENAME`**.

Итак, наш сайт установлен, давайте просто попытаемся выполнить сервер разработки – это проверка на токсичность, чтобы убедиться, что все подвижные части соединены:

```
elspeth@server:$ $ cd ~/sites/$SITENAME/source
$ python manage.py runserver
Traceback (most recent call last):
File "manage.py", line 8, in <module>
from django.core.management import execute_from_command_line
ImportError: No module named django.core.management
```

Ага. Django не установлен на сервере.

Создание Virtualenv вручную и использование requirements.txt

Чтобы сохранить список пакетов, которые нам понадобятся в виртуальной среде virtualenv, и чтобы суметь его воссоздать на сервере, создаем файл *requirements.txt*:

```
$ echo "django==1.11" > requirements.txt
$ git add requirements.txt
$ git commit -m "Добавлен requirements.txt для virtualenv"
```

Возможно, вас заинтересовало, почему мы не добавили в список необходимых пакетов нашу другую зависимость – Selenium. Дело в том, что Selenium является зависимостью только для тестов, но не для прикладного кода. Некоторым также нравится создавать файл под названием *test-requirements.txt*.

Теперь мы выполняем `git push`, чтобы отправить обновления с локального ПК на сайт обмена исходным кодом:

```
$ git push
```

И мы можем получить оттуда эти изменения на сервер:

```
elspeth@server:$ git pull # может запросить сначала выполнить git config
```

Создание virtualenv вручную (то есть без `virtualenvwraper`) сопряжено с использованием модуля venv стандартной библиотеки и определением пути, в который virtualenv будет входить:

```
elspeth@server:$ pwd
/home/espeth/sites/staging.superlists.com/source
elspeth@server:$ python3.6 -m venv ../virtualenv
elspeth@server:$ ls ../virtualenv/bin
activate      activate.fish  easy_install-3.6  pip3    python
activate.csh  easy_install   pip               pip3.6  python3
```

Если бы мы хотели активировать virtualenv, это можно было бы сделать с помощью `source ../virtualenv/bin/activate`. Но нам это не нужно. В действительности мы можем сделать все, что хотим, путем вызова версий Python, менеджера пакетов pip и других исполняемых файлов в каталоге *bin* виртуальной среды virtualenv, как мы увидим далее.

Чтобы установить пакеты из файла *requirements.txt* в virtualenv, используем pip в среде virtualenv:

```
elspeth@server:$ ../virtualenv/bin/pip install -r requirements.txt
Downloading/unpacking Django==1.11 (from -r requirements.txt (line 1))
[...]
Successfully installed Django
```

И чтобы выполнять Python в virtualenv, используем бинарный файл `python` в virtualenv:

```
elspeth@server:$ ../virtualenv/bin/python manage.py runserver
Validating models...
0 errors found
[...]
```

В зависимости от конфигурации брандмауэра в этой точке вы можете посетить сайт вручную. Для этого нужно выполнить `runserver 0.0.0.0:8000`, чтобы начать прослушивать на общедоступном и частном IP-адресе, и затем перейти по http://your.domain.com:8000.

Похоже, он успешно работает, и пока можно прервать (**Ctrl+C**) его работу.

Значительный прогресс! У нас есть система для передачи изменений исходного кода на сервер и с сервера (`git push` и `git pull`), установлена виртуальная среда virtualenv, которая соответствует нашей локальной, и есть один файл *requirements.txt* с информацией об их синхронизации.

Далее мы сконфигурируем веб-сервер Nginx, который будет общаться с Django и поднимет наш сайт на стандартном порту 80.

Простое конфигурирование Nginx

Мы создаем файл конфигурации веб-сервера Nginx, чтобы поручить ему отправку запросов для нашего промежуточного сайта в Django. Минимальная конфигурация выглядит так:

server: /etc/nginx/sites-available/superlists-staging.ottg.eu

```
server {
    listen 80;
    server_name superlists-staging.ottg.eu;

    location / {
        proxy_pass http://localhost:8000;
    }
}
```

Данная конфигурация говорит о том, что веб-сервер будет прослушивать только наш промежуточный домен и будет «проксировать» все запросы на локальный порт 8000, где должен находиться Django, отклика которого он будет ожидать.

Я сохранил эту конфигурацию в файл *superlists-staging.ottg.eu* внутри папки */etc/nginx/sites-available*.

Не знаете, как отредактировать файл на сервере? Для этого есть редактор `vi`, с которым я вам рекомендую немного познакомиться, но, возможно, на сегодня уже достаточно новых вещей. Вместо него можете попробовать дружественный для новичка редактор `nano`[8]. Обратите внимание: вам придется также воспользоваться `sudo`, потому что этот файл находится в системной папке.

Затем мы добавляем его к сайтам с включенной поддержкой сервера путем создания символьной ссылки на него:

[8] См. http://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/

```
elspeth@server:$ echo $SITENAME # check this still has our site in
superlists-staging.ottg.eu
elspeth@server:$ sudo ln -s ../sites-available/$SITENAME /etc/nginx/sites-
enabled/$SITENAME
elspeth@server:$ ls -l /etc/nginx/sites-enabled # проверьте, что символьная
ссылка на месте
```

Это предпочтительный в Debian/Ubuntu способ сохранения конфигурации Nginx – реальный конфигурационный файл лежит в *sites-available*, а символьная ссылка – в *sites-enabled*. Фишка в том, что это упрощает включение и выключение поддержки.

Мы также можем удалить используемую по умолчанию конфигурацию Welcome to Nginx, чтобы избежать путаницы:

```
elspeth@server:$ sudo rm /etc/nginx/sites-enabled/default
```

И теперь ее протестируем:

```
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ ../virtualenv/bin/python manage.py runserver
```

Я также должен был отредактировать */etc/nginx/nginx.conf* и раскомментировать `server_names_hash_bucket_size 64;`, чтобы мое длинное доменное имя заработало. У вас этой проблемы может и не быть; Nginx выдаст предупреждение, когда вы выполните `reload`, в случае возникновения у него каких-то проблем со своими конфигурационными файлами.

Быстрый визуальный осмотр подтверждает – сайт поднят (рис. 9.3)!

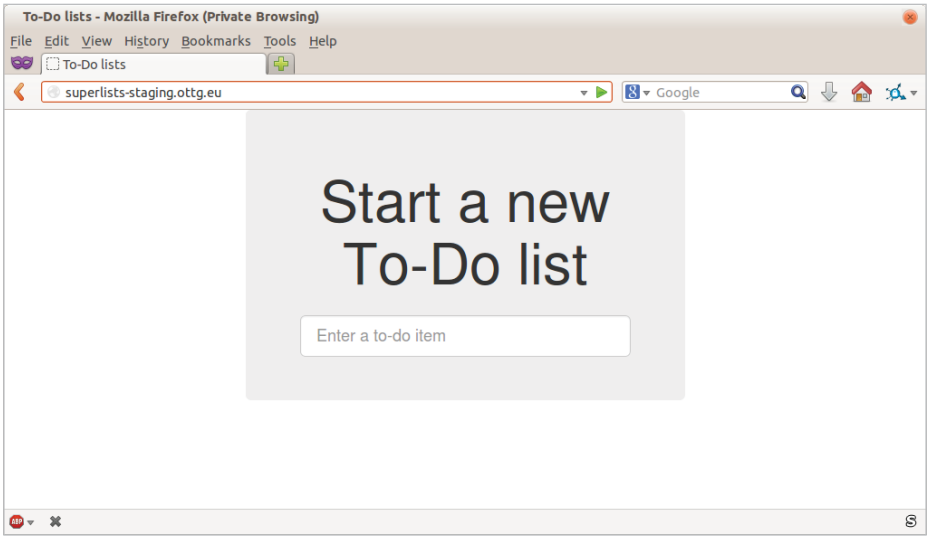


Рис. 9.3. Промежуточный сайт поднят!

Если вы когда-нибудь обнаружите, что Nginx ведет себя не так, как ожидалось, попробуйте команду sudo nginx-t, которая выполнит проверку конфигурации и предупредит вас обо всех проблемах в конфигурационных файлах.

Давайте посмотрим, что скажут наши функциональные тесты:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
[...]
AssertionError: 0.0 != 512 within 3 delta
```

Тесты не проходят, как только они пытаются передать новый элемент, потому что мы не установили базу данных. Вероятно, вы заметили желтую страницу отладки Django (рис. 9.4), которая сообщает нам результаты прохождения тестов либо о том, что попытка выполнена вручную.

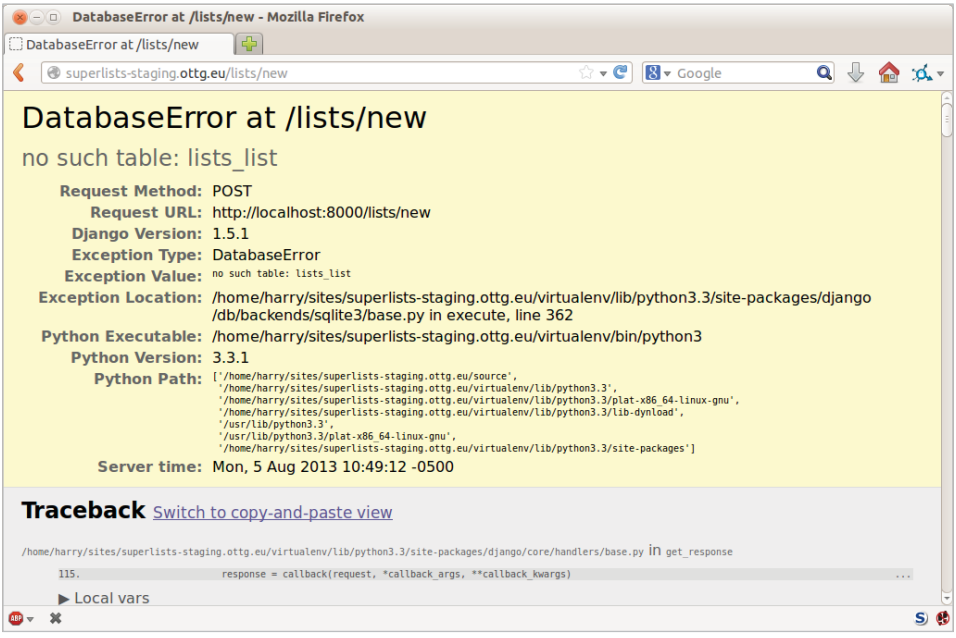


Рис. 9.4. Но база данных не подключена

Тесты спасли нас от потенциальных сложностей. Сайт *выглядел хорошо*, когда мы загрузили его первую страницу. Если бы мы немного поспешили, возможно, мы решили бы, что завершили работу, и только первые пользователи обнаружили бы противную отладочную страницу Django. Ну хорошо, немного преувеличиваю для эффекта! Возможно, мы бы выполнили проверку, но что бывает, когда сайт становится все сложнее и сложнее? Ведь вы не сможете проверить все! А вот тесты могут.

Создание базы данных при помощи команды migrate

Мы выполняем `migrate`, используя аргумент `--noinput` для подавления двух небольших сообщений «Вы уверены?», требующих подтверждения:

```
elspeth@server:$ ../virtualenv/bin/python manage.py migrate --noinput
Creating tables ...
[...]
elspeth@server:$ ls ../database/
db.sqlite3
elspeth@server:$ ../virtualenv/bin/python manage.py runserver
```

Давайте снова попробуем функциональные тесты:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
...
 ----------------------------------------------------------------------
Ran 3 tests in 10.718s

OK
```

Приятно наблюдать, что сайт находится в рабочем состоянии! В данный момент можно наградить себя заслуженным перерывом на чай, прежде чем мы двинемся к следующему разделу…

Если вы видите ошибку «502 – Bad Gateway» («502 – Недопустимый шлюз»), вероятно, вы забыли перезапустить сервер разработки командой `manage.py runserver` после команды `migrate`.

Еще несколько советов по отладке даны ниже во вставке.

Советы по отладке сервера

Развертывания полны каверз! Если дело не идет точно как ожидалось, предлагаю вам несколько советов и нюансов, к которым следует приглядеться:

- Наверняка вы уже перепроверили, что каждый файл находится точно там, где он должен быть, и имеет правильное содержимое – один случайный символ может все изменить.
- Журнал регистрации ошибок веб-сервера Nginx находится в */var/log/nginx/error.log*.
- Можно поручить Nginx проверить свою конфигурацию при помощи флага `-t`: `nginx -t`.
- Удостоверьтесь, что ваш браузер не кеширует устаревший отклик. Используйте комбинацию клавиш **Ctrl+Refresh** или запустите новое приватное окно браузера.
- Это может выглядеть как хватание за соломинку, но я иногда замечал необъяснимое поведение на сервере, которое удавалось урегулировать, только когда я полностью его перезапускал с помощью `sudo reboot`.

Если вы когда-нибудь окажетесь в полном тупике, всегда есть возможность снести ваш сервер и начать с нуля! Во второй раз все должно пойти быстрее.

Победа! Наше хакерское развертывание работает

Уф. Если учесть, что вам удалось привести в рабочее состояние все, что требовалось, то теперь по крайней мере можно надеяться, что основная магистраль работает, однако в действительности у нас не получится использовать сервер разработки Django в производственной среде. Мы также не можем опираться на его ручной запуск командой `runserver`. В следующей главе результат нашего неуклюжего развертывания станет более готовым к эксплуатации.

Тест-драйв конфигурации сервера и развертывания

Тесты устраняют часть неопределенности, которая есть в процессе развертывания

Администрирование сервера всегда дарит разработчикам приятные сюрпризы. Я имею в виду, что данный процесс полон неопределенности и неожиданностей. При написании этой главы я хотел показать, что комплект функциональных тестов может устранить часть неопределенности из данного процесса.

Типичные болевые точки: база данных, статические файлы, зависимости, пользовательские настройки

Во время любого развертывания нужно бдительно следить за конфигурацией базы данных, статическими файлами, зависимостями программного обеспечения и пользовательскими настройками, которые могут сильно различаться в зависимости от целей разработки или эксплуатации. Каждую из них нужно тщательно продумать для ваших собственных развертываний.

Тесты позволяют нам экспериментировать

Всегда, когда мы вносим изменение в конфигурацию сервера, можно выполнить комплект тестов повторно и удостовериться, что все работает так же, как прежде. Это позволит экспериментировать с нашими настройками с меньшим опасением (как мы увидим в следующей главе).

Глава 10

Переход к развертыванию, готовому к эксплуатации

В этой главе мы внесем некоторые коррективы в сайт, чтобы получить конфигурацию, более подготовленную к эксплуатации. По мере внесения каждого изменения будем использовать тесты для подтверждения, что все работает по-прежнему, как надо.

Что плохого в нашем неуклюжем развертывании? Дело в том, что мы не можем использовать сервер разработки Django в производственной среде, он не предназначен для реальных нагрузок. Вместо него для выполнения программного кода Django мы воспользуемся HTTP-сервером под названием Gunicorn и заставим Nginx раздавать статические файлы.

Наш файл *settings.py* в настоящее время содержит `DEBUG=True`. Настоятельно советую не использовать эту строку в производственной среде (вы же не хотите, к примеру, чтобы пользователи глазели на отладочную обратную трассировку программного кода, когда ваш сайт получает ошибки). Также нам нужно будет установить `ALLOWED_HOSTS` для безопасности.

Мы хотим, чтобы сайт запускался автоматически каждый раз, когда сервер перезагружается. Для этого мы напишем конфигурационный файл Systemd.

Наконец, жестко кодированный порт 8000 не позволит выполнять на этом сервере многочисленные сайты, поэтому мы перейдем на использование сокетов домена Unix для обеспечения связи между Nginx и Django.

Переход на Gunicorn

Вы знаете, почему талисманом Django является пони? История такова: в Django полно полезных вещей: ORM, разнообразное промежуточное программное обеспечение, сайт администратора… «Ну что еще тебе нужно, пони?» Так вот, Gunicorn означает «Green Unicorn» (Зеленый единорог). Если у вас уже есть пони, полагаю, единорог будет следующим, кого вы захотите…

```
elspeth@server:$ ../virtualenv/bin/pip install gunicorn
```

Gunicorn должен знать путь к серверу WSGI – обычно это функция под названием `application`. Django ее предоставляет в *superlists/wsgi.py*:

```
elspeth@server:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
2013-05-27 16:22:01 [10592] [INFO] Starting gunicorn 0.19.6
2013-05-27 16:22:01 [10592] [INFO] Listening at: http://127.0.0.1:8000 (10592)
[...]
```

Если теперь вы взглянете на сайт, то обнаружите, что CSS неработоспособен (как на рис. 10.1).

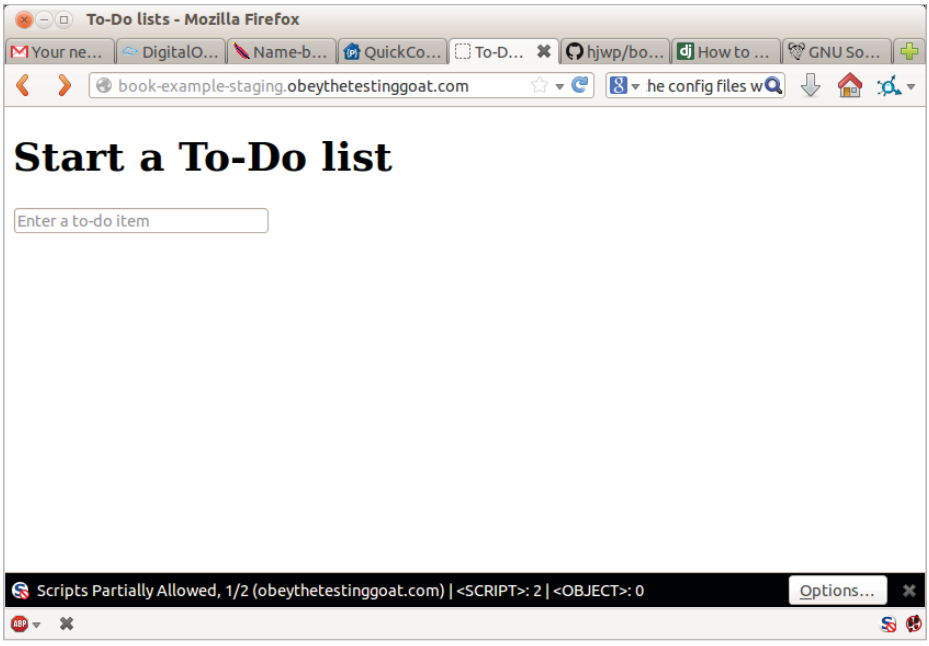


Рис. 10.1. Поврежденный CSS

И если выполнить функциональные тесты, они подтвердят, что что-то не так. Тест на добавление элементов списка проходит на ура, а тест на макет и стилистическое оформление – нет. Благодарим за работу, тесты!

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
AssertionError: 125.0 != 512 within 3 delta
FAILED (failures=1)
```

Причина повреждения CSS в том, что, в отличие от сервера разработки Django, который волшебным образом раздает вам статические фай-

лы, Gunicorn этого не делает. Самое время, чтобы вместо него это делал Nginx.

Получается, один шаг вперед и один шаг назад, но по крайней мере тесты всегда под рукой и нас выручат. Идем дальше!

Настройка Nginx для раздачи статических файлов

Сначала мы выполняем `collectstatic`, чтобы скопировать все статические файлы в папку, где Nginx может их найти:

```
elspeth@server:$ ../virtualenv/bin/python manage.py collectstatic --noinput
elspeth@server:$ ls ../static/
base.css bootstrap
```

Теперь говорим Nginx приступить к раздаче этих статических файлов:

```
server {
    listen 80;
    server_name superlists-staging.ottg.eu;

    location /static {
        alias /home/elspeth/sites/superlists-staging.ottg.eu/static;
    }

    location / {
        proxy_pass http://localhost:8000;
    }
}
```

Перезагрузите Nginx и перезапустите Gunicorn…

```
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ ../virtualenv/bin/gunicorn superlists.wsgi:application
```

Если еще раз взглянуть на сайт, мы увидим, что все выглядят куда лучше. Можно выполнить ФТ повторно:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]

...
 ----------------------------------------------------------------------
```

```
Ran 3 tests in 10.718s

OK
```

Уф.

Переход на использование сокетов Unix

Когда мы хотим раздавать в условиях промежуточного и реального вариантов, оба сервера не смогут использовать один и тот же порт 8000. Можно выделить разные порты, но это несколько необоснованно и до опасного легко перепутать и запустить промежуточный сервер на реальном порту или наоборот.

Более удачное решение – использовать сокеты домена Unix – они представляют собой что-то вроде файлов на диске, но могут использоваться в Nginx и Gunicorn, чтобы обмениваться сообщениями между собой. Мы поместим наши сокеты в */tmp*. Давайте изменим настройки прокси в Nginx:

server: /etc/nginx/sites-available/superlists-staging.ottg.eu

```
[...]
    location / {
        proxy_set_header Host $host;
        proxy_pass http://unix:/tmp/superlists-staging.ottg.eu.socket;
    }
}
```

`proxy_set_header` используется для подтверждения, что Gunicorn и Django знают, на каком домене они работают. Это требуется для свойства безопасности `ALLOWED_HOSTS`, которое мы собираемся включить.

Теперь перезапускаем Gunicorn, но на этот раз говорим ему прослушивать на сокете, а не на стандартном порту:

```
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ ../virtualenv/bin/gunicorn --bind \
unix:/tmp/superlists-staging.ottg.eu.socket superlists.wsgi:application
```

И вновь повторно выполняем функциональный тест, чтобы удостовериться, что все по-прежнему проходит:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
OK
```

Еще пара шагов!

Присвоение DEBUG значения False и настройка ALLOWED_HOSTS

Режим отладки DEBUG в Django очень хорош для того, чтобы копаться на своем собственном сервере, но оставлять эти страницы, полные сообщений об обратной трассировке, небезопасно[1].

Параметр настройки DEBUG находится вверху *settings.py*. Присваивая ему значение `False`, мы также должны установить другой параметр с именем `ALLOWED_HOSTS`. Он был добавлен как свойство безопасности в Django 1.5. К сожалению, в стандартном *settings.py* у него нет полезного комментария, но мы можем добавить его сами. Сделаем это на сервере:

server: superlists/settings.py

```
# ПРЕДУПРЕЖДЕНИЕ СИСТЕМЫ БЕЗОПАСНОСТИ:
# не запускать при включенном режиме debug в условиях эксплуатации!
DEBUG = False

TEMPLATE_DEBUG = DEBUG

# Необходимо, когда DEBUG=False
ALLOWED_HOSTS = ['superlists-staging.ottg.eu']
[...]
```

Еще раз мы перезапускаем Gunicorn и выполняем ФТ, чтобы проверить, что все по-прежнему работает.

Не фиксируйте эти изменения на сервере. Пока это просто обходной путь, используемый для того, чтобы все работало. Это не те изменения, которые мы хотим сохранить в репозитории. В целом, для упрощения я собираюсь выполнять фиксации Git только с локального ПК, используя `git push` и `git pull`, когда нужно синхронизировать их с сервером.

Сделаем еще один прогон теста, чтобы убедиться, что все по-прежнему работает?

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
OK
```

Хорошо.

[1] См. http://bit.ly/SuvluV

Применение Systemd для проверки, что Gunicorn запускается на начальной загрузке

Наш последний шаг – проверка, что сервер автоматически запускает Gunicorn на начальной загрузке и автоматически перезагружает, если он сбоит. В Ubuntu это делается при помощи Systemd:

server: /etc/systemd/system/gunicorn-superlists-staging.ottg.eu.service

```
[Unit]
Description=Gunicorn server for superlists-staging.ottg.eu

[Service]
Restart=on-failure  ❶
User=Elspeth  ❷
WorkingDirectory=/home/elspeth/sites/superlists-staging.ottg.eu/source  ❸
ExecStart=/home/elspeth/sites/superlists-staging.ottg.eu/virtualenv/bin/gunicorn \
    --bind unix:/tmp/superlists-staging.ottg.eu.socket \
    superlists.wsgi:application  ❹

[Install]
WantedBy=multi-user.target  ❺
```

К счастью, Systemd довольно легко конфигурируется (особенно если вы когда-либо имели сомнительное удовольствие писать сценарий `init.d`) и вполне говорит сам за себя.

❶ `Restart=on-failure` перезапустит процесс автоматически, если он сбоит.

❷ `User=elspeth` заставляет процесс выполняться в качестве пользователя elspeth.

❸ `WorkingDirectory` устанавливает текущий рабочий каталог.

❹ `ExecStart` – это фактически исполняемый процесс. Мы используем символы продолжения строк '\', чтобы расположить полную команду на нескольких строках для удобочитаемости, но она может быть написана и одной строкой.

❺ `WantedBy` в разделе `[Install]` говорит Systemd, что мы хотим, чтобы эта служба запускалась на начальной загрузке.

Сценарии Systemd лежат в */etc/systemd/system*, их имена должны заканчиваться на *.service*.

Теперь поручаем Systemd запускать Gunicorn командой `systemctl`:

```
# эта команда поручает Systemd загрузить новый файл конфигурации
elspeth@server:$ sudo systemctl daemon-reload
```

```
# эта команда поручает Systemd всегда загружать службу на начальной загрузке
elspeth@server:$ sudo systemctl enable gunicorn-superlists-staging.ottg.eu
# эта команда фактически запускает нашу службу
elspeth@server:$ sudo systemctl start gunicorn-superlists-staging.ottg.eu
```

(Кстати, команда `systemctl` откликается на завершение нажатием клавишей **Tab**, включая имя службы.)

Можно снова выполнить функциональные тесты, чтобы убедиться, что все по-прежнему работает. Вы можете даже проверить, что сайт возвращается, если перезагрузить сервер!

Дополнительные советы по отладке

- Проверьте записи в журнале Systemd на использование команды `sudo journalctl -u gunicorn-superlistsstaging.ottg.eu`.
- Вы можете запросить Systemd, чтобы он проверил допустимость конфигурации вашей службы: `systemd-analyze verify /path/to/my.service`.
- Не забывайте перезапускать обе службы всякий раз, когда вы вносите изменения.
- Когда вы вносите изменения в файл конфигурации Systemd, вам нужно выполнить `daemon-reload` перед `systemctl restart`, чтобы увидеть результат своих изменений.

Сохранение изменений: добавление Gunicorn в файл requirements.txt

В локальной *копии* вашего репозитория следует добавить Gunicorn в список пакетов, которые нужны нам в виртуальных средах virtualenv:

```
$ pip install gunicorn
$ pip freeze | grep gunicorn >> requirements.txt
$ git commit -am "Добавлен gunicorn как необходимое условие для virtualenv"
$ git push
```

При написании настоящей главы в Windows Gunicorn устанавливался на ура с помощью `pip install`, но на самом деле он не будет работать, если вы попытаетесь его использовать. К счастью, мы выполняем его только на сервере, так что проблем нет. А поддержка Windows находится в процессе обсуждения...[2]

[2] См. http://stackoverflow.com/questions/11087682/does-gunicorn-run-on-windows

Размышления об автоматизации

Резюмируем наши процедуры обеспечения работы и развертывания сайта.

Обеспечение работы

1. Предположим, у нас есть учетная запись пользователя и домашняя папка.
2. `add-apt-repository ppa:fkrull/deadsnakes`
3. `apt-get install nginx git python3.6 python3.6-venv`
4. Добавить конфигурацию Nginx для виртуального узла.
5. Добавить задание Systemd для Gunicorn.

Развертывание

1. Создать структуру каталогов в *~/sites*.
2. Получить исходный код из удаленного репозитория в папку *source*.
3. Запустить virtualenv в *../virtualenv/*.
4. `pip install -r requirements.txt`
5. `manage.py migrate` для базы данных.
6. `collectstatic` для статических файлов.
7. Присвоить `DEBUG = False` и `ALLOWED_HOSTS` в *settings.py*.
8. Перезапустить задание Gunicorn.
9. Выполнить функциональные тесты, чтобы проверить, что все работает.

Если учесть, что мы не готовы полностью автоматизировать процесс обеспечения работы, каким образом нам следует сохранить полученные на данный момент результаты исследования? Я бы сказал, что файлы конфигурации Nginx и Systemd следует, вероятно, где-то сохранить, чтобы позже их можно было легко использовать повторно. Давайте сохраним их в новой подпапке в репозитории:

Сохранение шаблонов конфигурационных файлов этапа обеспечения работы

Сначала создаем подпапку:

```
$ mkdir deploy_tools
```

Вот обобщенный шаблон для конфигурирования нашего Nginx:

deploy_tools/nginx.template.conf

```
server {
    listen 80;
```

```
    server_name SITENAME;

    location /static {
        alias /home/elspeth/sites/SITENAME/static;
    }

    location / {
        proxy_set_header Host $host;
        proxy_pass http://unix:/tmp/SITENAME.socket;
    }
}
```

А вот – для службы Gunicorn Systemd:

deploy_tools/gunicorn-systemd.template.service

```
[Unit]
Description=Gunicorn server for SITENAME

[Service]
Restart=on-failure
User=elspeth
WorkingDirectory=/home/elspeth/sites/SITENAME/source
ExecStart=/home/elspeth/sites/SITENAME/virtualenv/bin/gunicorn \
    --bind unix:/tmp/SITENAME.socket \
    superlists.wsgi:application

[Install]
WantedBy=multi-user.target
```

Теперь мы легко можем использовать эти два файла для генерации нового сайта, выполнив поиск и замену в `SITENAME`.

Что касается всего прочего, вполне достаточно оставить несколько примечаний. Почему бы их тоже не сохранить в файле в репозитории?

deploy_tools/provisioning_notes.md

```
Обеспечение работы нового сайта
===============================
## Необходимые пакеты:
* nginx
* Python 3.6
* virtualenv + pip
* Git

например, в Ubuntu:

    sudo add-apt-repository ppa:fkrull/deadsnakes
```

```
    sudo apt-get install nginx git python36 python3.6-venv

## Конфигурация виртуального узла Nginx

* см. nginx.template.conf
* заменить SITENAME, например, на staging.my-domain.com

## Служба Systemd

* см. gunicorn-systemd.template.service
* заменить SITENAME, например, на staging.my-domain.com

## Структура папок:
Если допустить, что есть учетная запись пользователя в /home/username

/home/username
└── sites
    └── SITENAME
        ├── database
        ├── source
        ├── static
        └── virtualenv
```

Мы можем это зафиксировать:

```
$ git add deploy_tools
$ git status # см. три новых файла
$ git commit -m "Файлы заметок и конфигурации шаблона для обеспечения
работы"
```

Дерево исходного кода теперь выглядит примерно так:

```
.
├── deploy_tools
│   ├── gunicorn-systemd.template.service
│   ├── nginx.template.conf
│   └── provisioning_notes.md
├── functional_tests
│   ├── [...]
├── lists
│   ├── __init__.py
│   ├── models.py
│   ├── [...]
│   ├── static
│   │   ├── base.css
│   │   └── bootstrap
```

```
|   |       ├── [...]
|   ├── templates
|   |   ├── base.html
|   |   ├── [...]
|   ├── tests.py
|   ├── urls.py
|   └── views.py
├── manage.py
├── requirements.txt
└── superlists
    ├── [...]
```

Сохранение хода выполнения

Успешное выполнение наших ФТ для промежуточного сервера может очень хорошо обнадеживать. Но в большинстве случаев от вас не требуется выполнять ФТ для реального сервера. Чтобы сохранить нашу работу и обнадежить себя, что производственный сервер будет функционировать точно так же, как реальный сервер, нужно сделать процесс развертывания повторимым.

Ответ лежит в области автоматизации, и она является темой следующей главы.

Готовность к эксплуатации при развертывании серверов

Вот несколько моментов, которые следует обдумать при попытке создать готовую к эксплуатации серверную среду.

Не используйте сервер разработки Django в производственной среде

Инструменты, подобные Gunicorn или Uwsgi, являются самыми оптимальными для выполнения Django – к примеру, они позволят вам выполнять многочисленные рабочие процессы.

Не используйте Django для раздачи статических файлов

Нет никакого смысла в использовании процесса Python для выполнения простого задания по раздаче статических файлов. С этим может справиться Nginx и другие веб-серверы, например Apache или uWSGI.

Проверьте файл settings.py на наличие настроек, предназначенных только для разработки

Мы рассмотрели две такие настройки: `DEBUG=True` и `ALLOWED_HOSTS`, но у вас, вероятно, будут и другие (мы увидим больше, когда начнем посылать электронные письма с сервера).

Безопасность

Серьезное обсуждение безопасности сервера выходит за рамки этой книги, и я настоятельно рекомендую не запускать свой собственный сервер без достаточных знаний по этой теме. (Одна из причин, почему люди используют PaaS для размещения своего кода заключается в том, что это дает несколько меньше проблем с безопасностью, о которой придется беспокоиться.) Если вам нужно место для старта, то в сноске ниже описывается хороший вариант (Смотрите статью «Мои первые 5 минут на сервере».) Я определенно могу рекомендовать бесценный опыт установки fail2ban и понаблюдать за его журнальными файлами, чтобы увидеть, как быстро он улавливает случайные хакерские атаки с попытками заполучить ваш логин по SSH. Интернет – это опасное место!

[3] См. https://plusbryan.com/my-first-5-minutes-on-a-server-or-essential-security-for-linux-servers

Глава 11

Автоматизация развертывания с помощью Fabric

Автоматизируйте, автоматизируйте и еще раз автоматизируйте.
– Риф Хорстман

Автоматизация развертывания имеет крайне важное значение для того, чтобы наши тесты на промежуточном сайте хоть чего-то стоили. Удостоверившись, что процедура развертывания обладает повторяемостью, мы обеспечиваем себе гарантию, что все пойдет хорошо, когда мы развернем систему в производственной среде.

Fabric – это инструмент, который позволяет автоматизировать команды, которые вы хотите выполнять на серверах. «fabric3» – дочерняя копия для Python 3:

```
$ pip install fabric3
```

Ничего страшного, если вы проигнорируете любые ошибки с сообщением «failed building wheel» («не удалось создать wheel-файл») во время установки fabric3, раз уж в конце говорится «Successfully installed...» («Успешно установлен...»).

В обычной конфигурации должен быть файл с именем *fabfile.py* с одной или более функциями, к которым позже можно обратиться из инструмента командной строки под названием `fab`, как показано ниже:

```
fab function_name:host=SERVER_ADDRESS
```

Эта команда вызовет `function_name`, передавая соединение с сервером по адресу `SERVER_ADDRESS`. Для указания имен пользователей и паролей есть много других опций, о которых можно узнать, применив команду `fab --help`.

Описание частей сценария Fabric для развертывания

Лучший способ увидеть, как он работает, – использовать пример. В сноске приведена ссылка на один из примеров, который я сделал ранее[1]. В нем показаны все шаги развертывания, которые мы прошли. Главная функция называется `deploy`; как раз ее мы вызываем из командной строки. Далее она вызывает несколько вспомогательных функций, которые мы одну за другой создадим вместе, давая пояснения по ходу.

deploy_tools/fabfile.py (ch09l001)

```
from fabric.contrib.files import append, exists, sed
from fabric.api import env, local, run
import random

REPO_URL = 'https://github.com/hjwp/book-example.git'  ❶

def deploy():
    '''развернуть'''
    site_folder = f'/home/{env.user}/sites/{env.host}'  ❷ ❸
    source_folder = site_folder + '/source'
    _create_directory_structure_if_necessary(site_folder)
    _get_latest_source(source_folder)
    _update_settings(source_folder, env.host)  ❷
    _update_virtualenv(source_folder)
    _update_static_files(source_folder)
    _update_database(source_folder)
```

❶ Может потребоваться обновить переменную `REPO_URL`, используя URL-адрес вашего репозитория Git на его сайте обмена исходным кодом.

❷ `env.host` будет содержать адрес сервера, который мы указали в командной строке, например *superlists.ottg.eu*.

❸ `env.user` будет содержать имя пользователя, которое вы используете для входа на сервер.

Радует, что каждая из этих вспомогательных функций имеет довольно информативные имена. Поскольку любая функция в fabfile теоретически может вызываться из командной строки, я использовал форму записи с начальным символом подчеркивания, который указывает, что они не предназначены быть частью общедоступного API в fabfile. Рассмотрим каждую из них в хронологическом порядке.

[1] См. http://www.bbc.co.uk/cult/classic/bluepeter/valpetejohn/trivia.shtml

Создание структуры каталогов

Вот как создать структуру каталогов, чтобы она не посыпалась, если уже существует:

deploy_tools/fabfile.py (ch09l002)

```
def _create_directory_structure_if_necessary(site_folder):
    '''создать структуру каталога, если нужно'''
    for subfolder in ('database', 'static', 'virtualenv', 'source'):
        run(f'mkdir -p {site_folder}/{subfolder}')  ❶ ❷
```

❶ `run` – наиболее распространенная команда Fabric. Она означает «выполнить эту команду оболочки на сервере». Команда `run` в этой главе будет повторять многие команды, которые мы выполняли в двух предыдущих главах вручную.

❶ Команда `mkdir -p` – полезная разновидность `mkdir`, которая лучше в двух ипостасях: она может создавать каталоги нескольких уровней вложенности и создает их только в случае необходимости. Так, `mkdir -p /tmp/foo/bar` создаст каталог *bar* и его родительский каталог *foo*, если ей потребуется. Она также не будет жаловаться, если *bar* уже существует[2].

Получение исходного кода из репозитория командой git

Далее мы хотим скачать последнюю версию исходного кода на сервер подобно тому, как мы делали это командой `git pull` в предыдущих главах:

deploy_tools/fabfile.py (ch09l003)

```
def _get_latest_source(source_folder):
    '''получить самый свежий исходный код'''
    if exists(source_folder + '/.git'):  ❶
        run(f'cd {source_folder} && git fetch')  ❷ ❸
    else:
        run(f'git clone {REPO_URL} {source_folder}')  ❹
    current_commit = local("git log -n 1 --format=%H", capture=True)  ❺
    run(f'cd {source_folder} && git reset --hard {current_commit}')  ❻
```

❶ `exists` проверяет существование каталога или файла на сервере. Мы ищем скрытую папку *.git*, чтобы проверить, был ли репозиторий уже клонирован в нее.

[2] Возможно вам интересно, почему мы формируем пути вручную при помощи f-строк вместо команды `os.path.join`, которую встречали ранее? Объяснение простое: потому что, если вы выполняете сценарии из Windows, то `path.join` будет использовать обратные косые, но на сервере нам определенно требуются прямые косые. Это типичная ловушка!

❷ Многие команды начинаются с `cd`, чтобы перейти в текущий рабочий каталог. Fabric не имеет состояний, поэтому не помнит, в каком каталоге вы находитесь от одной команды `run` до другой[3].

❸ `git fetch` внутри существующего репозитория получает все последние фиксации из веб (она аналогична `git pull`, но без немедленного обновления живого дерева исходного кода).

❹ Как вариант, можно использовать `git clone` с URL-адресом репозитория, чтобы принести свежее дерево исходного кода.

❺ Команда Fabric `local` выполняет команду на вашей локальной машине – в сущности, это просто обертка для `subprocess.Popen`, но она довольно удобна. Здесь мы захватываем выход из этого вызова `git log`, чтобы получить ID текущей фиксации, которая находится на вашем локальном ПК. Это означает, что сервер в итоге будет с любым исходным кодом, чей снимок взят из хранилища и помещен в настоящее время на вашу машину (при условии, что до этого вы отправили его на сервер).

❻ Выполняем полный сброс `reset --hard` в эту фиксацию, которая сотрет любые текущие изменения в каталоге исходного кода на сервере.

В результате мы выполняем либо `git clone` (если это свежее развертывание), либо `git fetch + git reset --hard` (если предыдущая версия исходного кода уже там); это эквивалент `git pull`, которую мы использовали, когда выполняли это вручную, но с `reset --hard`, чтобы перезаписать любые локальные изменения.

Для работы данного сценария необходимо предварительно выполнить `git push` вашей текущей локальной фиксации, чтобы сервер мог ее извлечь и выполнить сброс `reset` под нее. Если вы видите ошибку с сообщением `Could not parse object` (Невозможно проанализировать объект), попытайтесь выполнить `git push`.

Обновление файла settings.py

Далее мы обновляем файл настроек, чтобы установить переменные `ALLOWED_HOSTS` и `DEBUG` и создать новый `SECRET_KEY`:

deploy_tools/fabfile.py (ch09l004)

```
def _update_settings(source_folder, site_name):
    '''обновить настройки'''
```

[3] В Fabric имеется команда «cd», но я подумал, что ознакомление с новыми командами – уж слишком много для одной главы.

```
settings_path = source_folder + '/superlists/settings.py'
sed(settings_path, "DEBUG = True", "DEBUG = False")  ❶
sed(settings_path,
    'ALLOWED_HOSTS =.+$',
    f'ALLOWED_HOSTS = ["{site_name}"]'  ❷
)
secret_key_file = source_folder + '/superlists/secret_key.py'
if not exists(secret_key_file):  ❸
    chars = 'abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(-_=+)'
    key = ''.join(random.SystemRandom().choice(chars) for _ in range(50))
    append(secret_key_file, f'SECRET_KEY = "{key}"')
append(settings_path, '\nfrom .secret_key import SECRET_KEY')  ❹ ❺
```

❶ Команда Fabric `sed` выполняет строковую замену в файле; здесь она меняет `DEBUG` с `True` на `False`.

❷ А здесь она корректирует `ALLOWED_HOSTS` при помощи регулярного выражения, путем поиска совпадений с правильной строкой в файле.

❸ Django использует `SECRET_KEY` для некоторых своих крипто-операций – таких как данные cookie и защита от межсайтовой подделки запросов (CSRF). Считается хорошей практикой сделать так, чтобы секретный ключ на сервере отличался от секретного ключа в вашем репозитории исходного кода, потому что он может быть заметен для незнакомцев. Этот раздел генерирует новый ключ для импорта в настройки, если его там нет (как только у вас есть секретный ключ, он должен оставаться одинаковым между развертываниями). Более подробную информацию вы можете почерпнуть в документации Django[4].

❹ `append` просто добавляет строку в конец файла. (Эта команда достаточна умна, чтобы не беспокоиться, если строка уже там, но недостаточно умна, когда нужно автоматически добавить символ новой строки, если файл им не заканчивается. Отсюда и `\n`.)

❺ Я использую *относительный импорт* (`from .secret_key` вместо `from secret_key`), чтобы быть абсолютно уверенным, что мы импортируем локальный модуль, а не откуда-то в другом месте в `sys.path`. Об относительном импорте поговорим подробнее в следующей главе.

Вот так вот, копаясь в файле настроек, мы получаем возможность изменить конфигурацию на сервере. Еще одна общепринятая схема заключается в использовании переменных окружения. Мы рассмотрим ее в главе 21. Решайте сами, какая вам больше нравится.

[4] См. https://docs.djangoproject.com/en/1.11/topics/signing/

Обновление virtualenv

Далее мы создаем или обновляем виртуальную среду virtualenv:

deploy_tools/fabfile.py (ch09l005)

```
def _update_virtualenv(source_folder):
    '''обновить виртуальную среду'''
    virtualenv_folder = source_folder + '/../virtualenv'
    if not exists(virtualenv_folder + '/bin/pip'):  ❶
        run(f'python3.6 -m venv {virtualenv_folder}')
    run(f'{virtualenv_folder}/bin/pip install -r {source_folder}/
requirements.txt')  ❷
```

❶ Мы смотрим внутрь папки virtualenv в поисках исполняемого файла `pip`, чтобы проверить его существование.

❷ Затем используем `pip install -r`, как мы делали ранее.

Обновление статических файлов – это одиночная команда:

deploy_tools/fabfile.py (ch09l006)

```
def _update_static_files(source_folder):
    '''обновить статические файлы'''
    run(
        f'cd {source_folder}'  ❶
        ' && ../virtualenv/bin/python manage.py collectstatic --noinput'  ❷
```

❶ В Python вы можете разделять длинные значения на несколько строк, как тут, они конкатенируются в одну строку. Это очень частый источник дефектов, когда вам на самом деле требовался список строк, но вы забыли поставить запятую!

❷ Мы используем папку с двоичными файлами virtualenv всякий раз, когда нужно выполнить команду Django *manage.py*, чтобы убедиться, что мы получаем virtualenv в версии Django, а не системную версию.

Миграция базы данных при необходимости

Наконец, мы обновляем базу данных командой `manage.py migrate`:

deploy_tools/fabfile.py (ch09l007)

```
def _update_database(source_folder):
    '''обновить базу данных'''
    run(
        f'cd {source_folder}'
        ' && ../virtualenv/bin/python manage.py migrate --noinput'
    )
```

Опция `--noinput` удаляет любые интерактивные подтверждения в формате да/нет, с которыми Fabric не справляется.

И мы закончили! Могу представить, сколько нового вам пришлось усвоить, но я надеюсь, вы способны увидеть, как автоматизация повторяет всю ту работу, которую мы делали ранее вручную, с капелькой логики, чтобы все работало как на совершенно новых развертываниях, так и на существующих, которые нуждаются лишь в обновлении. Если вам нравятся слова с латинскими корнями, можно описать автоматизацию как *идемпотентную*, то есть имеющую одинаковый эффект независимо от того, выполняете ли вы ее один раз или многократно.

Испытание автоматизации

Давайте испытаем автоматизацию на существующем промежуточном сайте и посмотрим, как она работает для обновления развертывания, которое уже существует:

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu

[superlists-staging.ottg.eu] Executing task 'deploy'
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists-stagin
[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[localhost] local: git log -n 1 --format=%H
[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out: HEAD is now at 85a6c87 Add a fabfile for autom
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: sed -i.bak -r -e 's/DEBUG = True/DEBUG = False
[superlists-staging.ottg.eu] run: echo 'ALLOWED_HOSTS = ["superlists-staging.ott
[superlists-staging.ottg.eu] run: echo 'SECRET_KEY = '\\''4p2u8fi6)bltep(6nd_3tt
[superlists-staging.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >> "

[superlists-staging.ottg.eu] run: /home/elspeth/sites/superlists-staging.ottg.eu
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade t
[superlists-staging.ottg.eu] out: Requirement already satisfied (use --upgrade t
[superlists-staging.ottg.eu] out: Cleaning up...
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out:
```

```
[superlists-staging.ottg.eu] out: 0 static files copied, 11 unmodified.
[superlists-staging.ottg.eu] out:

[superlists-staging.ottg.eu] run: cd /home/elspeth/sites/superlists-staging.ottg
[superlists-staging.ottg.eu] out: Creating tables ...
[superlists-staging.ottg.eu] out: Installing custom SQL ...
[superlists-staging.ottg.eu] out: Installing indexes ...
[superlists-staging.ottg.eu] out: Installed 0 object(s) from 0 fixture(s)
[superlists-staging.ottg.eu] out:
Done.
Disconnecting from superlists-staging.ottg.eu... done.
```

Потрясающе! Обожаю заставлять компьютеры выдавать страницы за страницами выходных данных, как тут (на самом деле просто не могу отделаться от желания слышать, как они издают тихие бурчащие звуки компьютеров из 70-х подобно компьютеру космического корабля «Мать» из кинофильма «Чужой»). Если присмотреться, мы увидим, что он выполняет все наши указания: команды `mkdir-p` проходят на ура, даже притом что каталоги уже существуют. Далее `git pull` получает пару фиксаций, которые мы только что сделали. Команды `sed` и `echo >>` модифицируют наш файл *settings.py*. Затем `pip install -r requirements.txt` завершается без проблем, отмечая, что существующий virtualenv уже имеет все нужные пакеты. `collectstatic` также отмечает, что все статические файлы уже имеются, и наконец `migrate` завершается без помех.

Конфигурирование Fabric

Если для входа в систему вы используете ключ SSH, храните его в месте, заданном по умолчанию, и на сервере используете то же самое имя пользователя, что и локально, то Fabric должен просто заработать. Если же это не так, нужно внести несколько изменений, чтобы заставить команду `fab` выполнить ваши указания. Они касаются имени пользователя, месте хранения используемого ключа SSH или пароля.

Их можно передать в Fabric в командной строке. Сверьтесь:

```
$ fab --help
```

Либо обратитесь к документации[5] Fabric за более подробной информацией.

Развертывание на работающем сайте

Итак, давайте попытаемся применить Fabric на работающем (живом) сайте!

[5] См. http://docs.fabfile.org/

```
$ fab deploy:host=elspeth@superlists.ottg.eu

$ fab deploy --host=superlists.ottg.eu
[superlists.ottg.eu] Executing task 'deploy'
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/databa
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/static
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/virtua
[superlists.ottg.eu] run: mkdir -p /home/elspeth/sites/superlists.ottg.eu/source
[superlists.ottg.eu] run: git clone https://github.com/hjwp/book-example.git /ho
[superlists.ottg.eu] out: Cloning into '/home/elspeth/sites/superlists.ottg.eu/s
[superlists.ottg.eu] out: remote: Counting objects: 3128, done.
[superlists.ottg.eu] out: Receiving objects: 0% (1/3128)
[...]
[superlists.ottg.eu] out: Receiving objects: 100% (3128/3128), 2.60 MiB | 829 Ki
[superlists.ottg.eu] out: Resolving deltas: 100% (1545/1545), done.
[superlists.ottg.eu] out:

[localhost] local: git log -n 1 --format=%H
[superlists.ottg.eu] run: cd /home/elspeth/sites/superlists.ottg.eu/source && gi
[superlists.ottg.eu] out: HEAD is now at 6c8615b use a secret key file
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: sed -i.bak -r -e 's/DEBUG = True/DEBUG = False/g' "$(e
[superlists.ottg.eu] run: echo 'ALLOWED_HOSTS = ["superlists.ottg.eu"]' >> "$(ec
[superlists.ottg.eu] run: echo 'SECRET_KEY = '\\''mqu(ffwid5vleol%ke^jil*x1mkj-4
[superlists.ottg.eu] run: echo 'from .secret_key import SECRET_KEY' >> "$(echo /
[superlists.ottg.eu] run: python3.6 -m venv /home/elspeth/sites/superl
[superlists.ottg.eu] out: Using interpreter /usr/bin/python3.6
[superlists.ottg.eu] out: Using base prefix '/usr'
[superlists.ottg.eu] out: New python executable in /home/elspeth/sites/superlist
[superlists.ottg.eu] out: Also creating executable in /home/elspeth/sites/superl
[superlists.ottg.eu] out: Installing Setuptools.....................done.
[superlists.ottg.eu] out: Installing Pip...........................done.
[superlists.ottg.eu] out:

[superlists.ottg.eu] run: /home/elspeth/sites/superlists.ottg.eu/source/../virtu
[superlists.ottg.eu] out: Downloading/unpacking Django==1.11.5 (from -r /home/el
[superlists.ottg.eu] out: Downloading Django-1.11.5.tar.gz (8.0MB):
[...]
[superlists.ottg.eu] out: Downloading Django-1.11.5.tar.gz (8.0MB): 100% 8.0M
[superlists.ottg.eu] out: Running setup.py egg_info for package Django
[superlists.ottg.eu] out:
[superlists.ottg.eu] out: warning: no previously-included files matching '__
[superlists.ottg.eu] out: warning: no previously-included files matching '*.
```

```
[superlists.ottg.eu] out: Downloading/unpacking gunicorn==17.5 (from -r /home/el
[superlists.ottg.eu] out: Downloading gunicorn-17.5.tar.gz (367kB): 100% 367k
[...]
[superlists.ottg.eu] out: Downloading gunicorn-17.5.tar.gz (367kB): 367kB down
[superlists.ottg.eu] out: Running setup.py egg_info for package gunicorn
[superlists.ottg.eu] out:
[superlists.ottg.eu] out: Installing collected packages: Django, gunicorn
[superlists.ottg.eu] out: Running setup.py install for Django
[superlists.ottg.eu] out: changing mode of build/scripts-3.3/django-admin.py
[superlists.ottg.eu] out:
[superlists.ottg.eu] out: warning: no previously-included files matching '__
[superlists.ottg.eu] out: warning: no previously-included files matching '*.
[superlists.ottg.eu] out: changing mode of /home/elspeth/sites/superlists.ot
[superlists.ottg.eu] out: Running setup.py install for gunicorn
[superlists.ottg.eu] out:
[superlists.ottg.eu] out: Installing gunicorn_paster script to /home/elspeth
[superlists.ottg.eu] out: Installing gunicorn script to /home/elspeth/sites/
[superlists.ottg.eu] out: Installing gunicorn_django script to /home/elspeth
[superlists.ottg.eu] out: Successfully installed Django gunicorn
[superlists.ottg.eu] out: Cleaning up...
[superlists.ottg.eu] out:
[superlists.ottg.eu] run: cd /home/elspeth/sites/superlists.ottg.eu/source && ..
[superlists.ottg.eu] out: Copying '/home/elspeth/sites/superlists.ottg.eu/source
[superlists.ottg.eu] out: Copying '/home/elspeth/sites/superlists.ottg.eu/source
[...]
[superlists.ottg.eu] out: Copying '/home/elspeth/sites/superlists.ottg.eu/source
[superlists.ottg.eu] out:
[superlists.ottg.eu] out: 11 static files copied.
[superlists.ottg.eu] out:
[superlists.ottg.eu] run: cd /home/elspeth/sites/superlists.ottg.eu/source && ..
[superlists.ottg.eu] out: Creating tables ...
[superlists.ottg.eu] out: Creating table auth_permission
[...]
[superlists.ottg.eu] out: Creating table lists_item
[superlists.ottg.eu] out: Installing custom SQL ...
[superlists.ottg.eu] out: Installing indexes ...
[superlists.ottg.eu] out: Installed 0 object(s) from 0 fixture(s)
[superlists.ottg.eu] out:
Done.
Disconnecting from superlists.ottg.eu... done.
```

Бр-р-р. Вы видите, что этот сценарий следует по несколько иному пути, выполняя `git clone`, чтобы принести совершенно новый репозиторий вместо `git pull`. Он также требует настройки новой виртуальной среды virtualenv с нуля, включая свежую установку pip и Django. Коман-

да `collectstatic` на этот раз фактически создает новые файлы, и команда `migrate`, похоже, тоже сработала.

Конфигурирование Nginx и Gunicorn при помощи sed

Что еще мы должны сделать, чтобы вывести работающий сайт в производственную среду? Мы обращаемся к нашим запискам на тему обеспечения работы сайта, в которых говорится об использовании шаблонных файлов для создания виртуального узла Nginx и службы Systemd. А как насчет небольшого волшебства командной строки Unix?

```
elspeth@server:$ sed "s/SITENAME/superlists.ottg.eu/g" \
    source/deploy_tools/nginx.template.conf \
    | sudo tee /etc/nginx/sites-available/superlists.ottg.eu
```

`sed` (от «stream editor» – потоковый редактор) принимает поток текста и выполняет с ним операции правки. Совсем не случайно, что в Fabric строковая команда замены имеет то же название. В этом случае мы просим, чтобы она заменила строку *SITENAME* с адресом нашего сайта на синтаксическую конструкцию `s/заменить-меня/на-это/g`[6]. Ее результат мы передаем по конвейеру (`|`) в процесс root-пользователя (sudo), который использует `tee` для записи в файл того, что ему передано по конвейеру, в данном случае – доступный для Nginx-сайтов файл конфигурации virtualhost.

Затем мы активируем этот файл символьной ссылкой:

```
elspeth@server:$ sudo ln -s ../sites-available/superlists.ottg.eu \
    /etc/nginx/sites-enabled/superlists.ottg.eu
```

И мы пишем службу Systemd при помощи еще одной команды `sed`:

```
elspeth@server: sed "s/SITENAME/superlists.ottg.eu/g" \
    source/deploy_tools/gunicorn-systemd.template.service \
    | sudo tee /etc/systemd/system/gunicorn-superlists.ottg.eu.service
```

Наконец запускаем обе службы:

```
elspeth@server:$ sudo systemctl daemon-reload
elspeth@server:$ sudo systemctl reload nginx
elspeth@server:$ sudo systemctl enable gunicorn-superlists.ottg.eu
elspeth@server:$ sudo systemctl start gunicorn-superlists.ottg.eu
```

И бросаем взгляд на наш сайт: рис. 11.1. Все работает, ура!

Он – молодец! Хороший fabfile, возьми булочку. Ты заработал право быть добавленным в репозиторий:

[6] Возможно, вам встречались фанаты, которые используют в Интернет эту странную нотацию s/изменить-это/на-это/. Теперь вы знаете почему!

```
$ git add deploy_tools/fabfile.py
$ git commit -m "Добавлен fabfile для автоматизированных размещений"
```

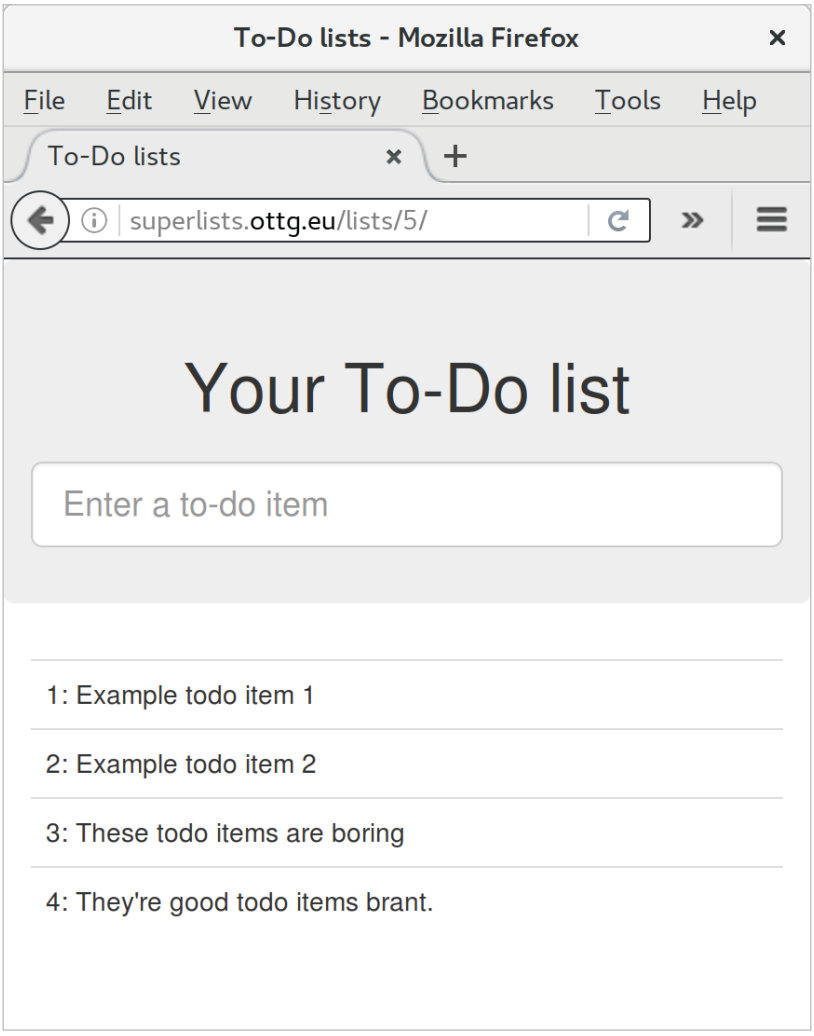


Рис. 11.1. Бр-р-р ... сработало!

Маркировка релиза командой git tag

Заключительная часть администраторской работы. Чтобы хранить историю изменений, мы будем использовать теги Git для маркировки состояния кодовой базы, которое отражает то, что в настоящее время работает на сервере:

```
$ git tag LIVE
$ export TAG=$(date +DEPLOYED-%F/%H%M) # генерирует метку времени
```

```
$ echo $TAG # должна показать "DEPLOYED-" и затем метку времени
$ git tag $TAG
$ git push origin LIVE $TAG # запихивает теги в репо
```

Теперь можно легко, в любое время проверить, какова разница между текущей кодовой базой и той, которая работает на серверах. Это пригодится в нескольких главах, когда мы обратимся к миграциям базы данных. Взгляните на тег в истории изменений:

```
$ git log --graph --oneline --decorate
[...]
```

Так или иначе теперь у вас есть действующий веб-сайт! Расскажите всем своим друзьям! Расскажите своей маме, если никому больше это не интересно! В следующей главе мы вернемся к программированию.

Дополнительные материалы для чтения

В процессе развертывания нет того, что называется одним единственным истинным путем, и я не седой эксперт в любом случае. Я попытался направить вас по вполне нормальному пути, но многое вы могли бы сделать иначе и узнать гораздо больше. Вот некоторые ресурсы, которые я использовал для вдохновения:

- Hynek Schlawack, «*Основательные развертывания на Python для всех*» (Solid Python Deployments for Everybody), http://hynek.me/talks/python-deployments;
- Dan Bravender, «*Потрясающие развертывания на основе Git и Fabric*» (Git-based fabric deployments are awesome), http://bit.ly/U6tUo5;
- Глава о развертывании в «*Двух ложечках Django*» (Two Scoops of Django), Dan Greenfeld и Audrey Roy;
- Heroku team, «*12-факторное приложение*» (The 12-factor App), http://12factor.net/.

Что касается некоторых идей о том, что вам делать с автоматизацией этапа обеспечения работы тестовой среды и альтернативы для Fabric под названием Ansible, обратитесь к приложению С.

Автоматизированные развертывания

Fabric

Инструмент Fabric позволяет выполнять команды на серверах изнутри сценариев Python. Это великолепный инструмент для автоматизации задач администрирования серверов.

Идемпотентность

Если ваш сценарий развертывания внедряется на существующие серверы, его нужно конструировать так, чтобы он инсталлировал относительно свежие версии ПО и работал относительно сервера, который уже сконфигурирован.

Держите файлы конфигурации под управлением версиями исходного кода

Проверьте, чтобы ваша единственная копия файла конфигурации не находилась на сервере! Они имеют критически важное значение для вашего приложения и должны находиться под управлением версиями, как все остальное.

Автоматизация этапа обеспечения работы тестовой среды

В конечном счете должно быть автоматизировано абсолютно все, включая запуск совершенно новых серверов и наличие на них всего нужного программного обеспечения. Это касается и взаимодействия с API вашего поставщика услуг по размещению сайтов (услуг хостинга).

Инструменты управления конфигурацией

Fabric – очень гибкий инструмент, но его логика по-прежнему основывается на сценариях. Более усовершенствованные инструменты проявляют более декларативный подход и могут сделать вашу жизнь еще проще. Ansible и Vagrant – два инструмента, которыми стоит заняться (см. приложение C), но есть еще и множество других (Chef, Puppet, Salt, Juju…).

Глава 12

Разделение тестов на многочисленные файлы и обобщенный помощник ожидания

Следующий функциональный элемент, реализацией которого мы займемся, – это небольшая проверка вводимых значений. Однако начав писать новые тесты, мы заметим, что в одном-единственном файле *functional_tests.py* и *tests.py* становится трудно ориентироваться, поэтому мы реорганизуем их в многочисленные файлы – небольшая рефакторизация наших тестов, если вы не против.

Мы также создадим обобщенный помощник с явным ожиданием.

Начало с ФТ валидации данных: предотвращение пустых элементов

По мере того, как наши первые несколько пользователей начинают работать с сайтом, мы замечаем, что иногда они допускают ошибки, которые портят их списки: например, случайная отправка пустых элементов списка или случайный ввод двух идентичных элементов. Компьютеры предназначены для того, чтобы не давать нам совершать глупые ошибки, поэтому давайте убедимся, что сайт сможет в этом помочь.

Вот краткое описание ФТ:

functional_tests/tests.py (ch11l001)

```
def test_cannot_add_empty_list_items(self):
    '''тест: нельзя добавлять пустые элементы списка'''
    # Эдит открывает домашнюю страницу и случайно пытается отправить
```

```
    # пустой элемент списка. Она нажимает Enter на пустом поле ввода

    # Домашняя страница обновляется, и появляется сообщение об ошибке,
    # которое говорит, что элементы списка не должны быть пустыми

    # Она пробует снова, теперь с неким текстом для элемента, и теперь
    # это срабатывает

    # Как ни странно, Эдит решает отправить второй пустой элемент списка

    # Она получает аналогичное предупреждение на странице списка

    # И она может его исправить, заполнив поле неким текстом
    self.fail('напиши меня!')
```

Это все очень хорошо, но наш файл с функциональными тестами начинает переполняться. Давайте разделим его на несколько файлов, в которых будет один метод тестирования.

Напомним, что функциональные тесты неразрывно связаны с «историями пользователя». Если бы вы использовали какой-то инструмент управления проектами наподобие системы отслеживания вопросов, то могли бы сделать так, чтобы каждый файл соответствовал одному вопросу или карточке, а имя файла содержало идентификатор карточки. Или же, если вы мыслите категориями «компонентов», где один компонент может иметь несколько историй пользователя, то у вас может быть один файл и класс для компонента и несколько методов для каждой истории пользователя.

Также у нас будет один базовый класс тестирования, от которого они все могут наследовать. Вот как этого можно добиться в пошаговом режиме.

Пропуск теста

Всегда приятно при рефакторизации иметь комплект полностью проходящих тестов. Мы только что написали тест с преднамеренной неполадкой. На время выключим его с помощью декоратора пропуска фрагментов кода под названием «skip» из модуля `unittest`:

functional_tests/tests.py (ch11l001-1)

```
from unittest import skip
[...]

    @skip
    def test_cannot_add_empty_list_items(self):
        '''тест: нельзя добавлять пустые элементы списка'''
```

Этот фрагмент говорит исполнителю тестов проигнорировать этот тест. Вы можете увидеть, как это работает – если выполнить тесты повторно, декоратор скажет, что они проходят:

```
$ python manage.py test functional_tests
[...]
Ran 4 tests in 11.577s
OK
```

Декораторы пропуска небезопасны – вам придется помнить об их удалении перед выполнением фиксаций в репозитории. Вот почему неплохо выполнять построчную проверку различий между версиями!

Не забывайте про рефакторизацию в правиле «красный, зеленый, рефакторизуй»

Критические замечания, иногда выдвигаемые в адрес методологии TDD, заключаются в том, что она приводит к плохо структурированному программному коду. Разработчик концентрируется лишь на том, чтобы тесты успешно проходили, не задумываясь о том, как должна разрабатываться вся система в целом. Считаю такие замечания несколько пристрастными.

TDD – это не чудодейственное средство. Нам по-прежнему приходится тратить время на размышления о хорошей структуре кода. Просто часто случается, что люди забывают о рефакторизации в правиле «Красный, зеленый, рефакторизуй». Данная методология позволяет сводить вместе любой старый программный код таким образом, чтобы ваши тесты проходили успешно, но она также просит вас потратить некоторое время на его рефакторизацию с целью улучшения структуры. В противном случае слишком легко допустить рост «технической задолженности»[1].

Однако зачастую лучшие мысли о том, как перестроить свой код, приходят не сразу. Они могут появиться спустя дни, недели и даже месяцы после написания фрагмента программного кода, когда вы уже работаете над чем-то совершенно другим. И вы оказываетесь перед старым программным кодом, глядя на него новыми глазами. Но если вы занимаетесь какой-то другой задачей, следует ли вам останавливаться, чтобы выполнить рефакторизацию старого кода?

Ответ зависит от обстоятельств. В данном случае мы в начале главы и даже не начинали писать новый программный код. Мы знаем, что у нас все работает, поэтому можем решительно поставить декоратор пропуска на нашем новом ФТ (чтобы вернуться к полностью проходным тестам) и немедленно выполнить небольшую рефакторизацию.

[1] См. https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

> Позже в этой главе мы идентифицируем другие фрагменты кода, которые хотим изменить. В этих случаях, вместо того чтобы рисковать с рефакторизацией приложения, которое пока еще не работает, отметим фрагмент, который мы хотим изменить, в нашем блокноте и дождемся возвращения к комплекту из полностью проходящих тестов, прежде чем заняться рефакторизацией.

Разбиение функциональных тестов на несколько файлов

Сначала помещаем каждый тест в свой собственный класс, по-прежнему в том же файле:

functional_tests/tests.py (ch11l002)

```
class FunctionalTest(StaticLiveServerTestCase):
    '''функциональный тест'''

    def setUp(self):
        '''установка'''
        [...]
    def tearDown(self):
        '''демонтаж'''
        [...]
    def wait_for_row_in_list_table(self, row_text):
        '''ожидать строки в таблице списка'''
        [...]

class NewVisitorTest(FunctionalTest):
    '''тест нового посетителя'''

    def test_can_start_a_list_for_one_user(self):
        '''тест: можно начать список для одного пользователя'''
        [...]
    def test_multiple_users_can_start_lists_at_different_urls(self):
        '''тест: многочисленные пользователи могут начать списки
           по разным url'''
        [...]

class LayoutAndStylingTest(FunctionalTest):
    '''тест макета и стилевого оформления'''

    def test_layout_and_styling(self):
        '''тест макета и стилевого оформления'''
        [...]

class ItemValidationTest(FunctionalTest):
```

```
    '''тест валидации элемента списка'''

    @skip
    def test_cannot_add_empty_list_items(self):
        '''тест: нельзя добавлять пустые элементы списка'''
        [...]
```

В этой точке мы можем выполнить функциональные тесты повторно и убедиться, что все они по-прежнему работают:

```
Ran 4 tests in 11.577s

OK
```

Здесь мы немного перегнули палку. Пожалуй, можно было обойтись меньшим количеством шагов, но, как я продолжаю неустанно повторять, практическое применение пошагового метода на простых случаях делает работу намного проще, когда мы сталкиваемся со сложным случаем.

Перейдем от использования одиночного файла с тестами к использованию одного файла для каждого класса и одного основного файла с базовым классом, от которого все тесты наследуют. Мы сделаем четыре копии *tests.py*, назвав их соответствующим образом, и из каждого удалим фрагменты, которые нам не нужны:

```
$ git mv functional_tests/tests.py functional_tests/base.py
$ cp functional_tests/base.py functional_tests/test_simple_list_creation.py
$ cp functional_tests/base.py functional_tests/test_layout_and_styling.py
$ cp functional_tests/base.py functional_tests/test_list_item_validation.py
```

Файл *base.py* можно свести только к классу `FunctionalTest`. Мы оставляем вспомогательный метод в базовом классе, так как у нас есть подозрения, что мы будем использовать его повторно в новом ФТ:

functional_tests/base.py (ch11l003)

```
import os
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium import webdriver

class FunctionalTest(StaticLiveServerTestCase):
    '''функциональный тест'''

    def setUp(self):
        '''установка'''
        [...]
    def tearDown(self):
```

```
        '''демонтаж'''
        [...]
    def wait_for_row_in_list_table(self, row_text):
        '''ожидать строки в таблице списка'''
        [...]
```

Хранение вспомогательных методов в базовом классе `FunctionalTest` – один полезных способов предотвратить дублирование во всех ФТ. В главе 25 мы воспользуемся «Страничным шаблоном проектирования», который связан с этим способом, но в нем предпочтение отдается композиции, а не наследованию, что всегда неплохо.

Наш первый ФТ находится теперь в своем собственном файле и должен представлять собой всего один класс и один метод тестирования:

functional_tests/test_simple_list_creation.py (ch11l004)

```
from .base import FunctionalTest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class NewVisitorTest(FunctionalTest):
    '''тест нового посетителя'''

    def test_can_start_a_list_for_one_user(self):
        '''тест: можно начать список для одного пользователя'''
        [...]
    def test_multiple_users_can_start_lists_at_different_urls(self):
        '''тест: многочисленные пользователи могут начать списки
           по разным url'''
        [...]
```

Я использовал относительный импорт (`from .base`). Некоторым нравится применять его повсюду в исходном коде Django (например, представления могут импортировать модели, используя `from .models import List`, вместо `from list.models`). В конечном счете это вопрос личных предпочтений. Я использую относительный импорт, только когда абсолютно уверен, что относительное положение импортируемого не изменится. В данном случае это применимо, потому что я знаю наверняка, что все тесты будут находиться рядом с *base.py*, от которого они наследуют.

Макет и стилевое оформление ФТ теперь должны быть одним файлом и одним классом:

functional_tests/test_layout_and_styling.py (ch11l005)

```
from selenium.webdriver.common.keys import Keys
from .base import FunctionalTest

class LayoutAndStylingTest(FunctionalTest):
    '''тест макета и стилевого оформления'''
    [...]
```

Наконец, наша новая проверка допустимости вводимых данных тоже находится в собственном файле:

functional_tests/test_list_item_validation.py (ch11l006)

```
from selenium.webdriver.common.keys import Keys
from unittest import skip
from .base import FunctionalTest

class ItemValidationTest(FunctionalTest):
    '''тест валидации элемента списка'''
    @skip
    def test_cannot_add_empty_list_items(self):
        '''тест: нельзя добавлять пустые элементы списка'''
        [...]
```

И мы можем протестировать, что все работает, повторно выполнив `manage.py test functional_tests`, и еще раз убедиться, что все четыре теста выполняются:

```
Ran 4 tests in 11.577s

OK
```

Теперь можно удалить декоратор пропуска:

functional_tests/test_list_item_validation.py (ch11l007)

```
class ItemValidationTest(FunctionalTest):
    '''тест валидации элемента списка'''

    def test_cannot_add_empty_list_items(self):
        '''тест: нельзя добавлять пустые элементы списка'''
        [...]
```

Выполнение только одного файла с тестами

В качестве бонуса теперь мы в состоянии выполнить отдельный файл с тестами, как здесь:

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
AssertionError: write me!
```

Блестяще! Не нужно сидеть без дела, ожидая всех ФТ, когда нас интересует всего один. Правда, важно не забывать время от времени выполнять их все вместе с целью проверки на регрессию. Позже в этой книге мы рассмотрим, как передавать эту задачу в автоматизированный цикл непрерывной интеграции. А пока давайте зафиксируем!

```
$ git status
$ git add functional_tests
$ git commit -m "Перемещены ФТ в свои собственные отдельные файлы"
```

Великолепно! Мы корректно разбили наши функциональные тесты на разные файлы. Далее мы начнем писать наш ФТ, но в ближайшее время, как вы догадываетесь, мы сделаем нечто подобное и с нашими модульными тестами.

Новый инструмент функционального тестирования: обобщенная вспомогательная функция явного ожидания

Сначала займемся реализацией теста или как минимум его начала:

functional_tests/test_list_item_validation.py (ch11l008)

```
def test_cannot_add_empty_list_items(self):
    '''тест: нельзя добавлять пустые элементы списка'''
    # Эдит открывает домашнюю страницу и случайно пытается отправить
    # пустой элемент списка. Она нажимает Enter на пустом поле ввода
    self.browser.get(self.live_server_url)
    self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)

    # Домашняя страница обновляется, и появляется сообщение об ошибке,
    # которое говорит, что элементы списка не должны быть пустыми
    self.assertEqual(
        self.browser.find_element_by_css_selector('.has-error').text,  ❶
        "You can't have an empty list item"  ❷
    )

    # Она пробует снова теперь с неким текстом для элемента, и теперь
    # это срабатывает
    self.fail('Закончить тест!')
    [...]
```

Вот как мы могли бы написать тест наивно:

❶ Мы указываем, что собираемся использовать класс CSS под названием `.has-error`, чтобы отметить текст с ошибкой. Мы увидим, что платформа Bootstrap имеет несколько подходящих методов стилевого оформления.

❷ И мы можем проверить, что наша ошибка отображает сообщение так, как мы хотим.

Но догадаетесь ли вы, в чем потенциальная проблема теста, который теперь написан?

Подскажу: ответ кроется в заголовке раздела; всякий раз, когда мы делаем что-то, что вызывает обновление страницы, нам требуется явное ожидание, в противном случае Selenium может заняться поиском элемента `.has-error`, прежде чем страница получит шанс загрузиться.

Всякий раз, когда вы отправляете форму с `Keys.ENTER` или нажимаете на что-то, что заставит страницу загрузиться, вы, вероятно, захотите, чтобы выполнялось явное ожидание вашего следующего утверждения `assert`.

Наше первое явное ожидание было встроено во вспомогательный метод. Мы можем решить, что на данном этапе создание конкретного вспомогательного метода – это уже перебор, но было бы неплохо иметь какой-то обобщенный способ уведомлять в тестах: «Жди, пока это утверждение не пропадет». Что-то вроде этого:

functional_tests/test_list_item_validation.py (ch11l009)

```
[...]
    # Домашняя страница обновляется, и появляется сообщение об ошибке,
    # которое говорит, что элементы списка не должны быть пустыми
    self.wait_for(lambda: self.assertEqual(  ❶
        self.browser.find_element_by_css_selector('.has-error').text,
        "You can't have an empty list item"
    ))
```

❶ Вместо того чтобы вызвать утверждение непосредственно, мы заворачиваем его в лямбда-функцию и передаем новому вспомогательному методу, который назовем `wait_for`.

Если вы прежде никогда не встречали лямбда-функции в Python, то смотрите вставку ниже.

Каким же образом этот волшебный метод `wait_for` будет работать? Давайте направимся в *base.py*, сделаем копию нашего существующего метода `wait_for_row_in_list_table` и слегка его адаптируем:

functional_tests/base.py (ch11l010)

```
def wait_for(self, fn):  ❶
    '''ожидать'''
    start_time = time.time()
    while True:
        try:
            table = self.browser.find_element_by_id('id_list_table')  ❷
            rows = table.find_elements_by_tag_name('tr')
            self.assertIn(row_text, [row.text for row in rows])
            return
        except (AssertionError, WebDriverException) as e:
            if time.time() - start_time > MAX_WAIT:
                raise e
            time.sleep(0.5)
```

❶ Мы делаем копию метода, но называем его `wait_for` и изменяем его аргумент. Он ожидает, что ему будет передана функция.

❷ Пока у нас по-прежнему старый программный код, который проверяет строки таблицы. Как преобразовать его во нечто работающее для любой обобщенной функции `fn`, которая была передана?

Это делается вот так:

functional_tests/base.py (ch11l011)

```
def wait_for(self, fn):
    '''ожидать'''
    start_time = time.time()
    while True:
        try:
            return fn()  ❶
        except (AssertionError, WebDriverException) as e:
            if time.time() - start_time > MAX_WAIT:
                raise e
            time.sleep(0.5)
```

❶ Вместо того чтобы быть специфическим фрагментом кода, предназначенным для исследования строк таблицы, тело блока `try/except` становится вызовом функции, которую мы передаем. Мы также применяем `return` с ее возвращаемым значением, чтобы иметь возможность сразу же выйти из цикла, если исключение не было поднято.

Лямбда-функции

`lambda` в Python – это синтаксическая конструкция для создания короткой, временной функции. Она снимает необходимость использовать конструкцию `def..():` и блока с отступом.

```
>>> myfn = lambda x: x+1
>>> myfn(2)
3
>>> myfn(5)
6
>>> adder = lambda x, y: x + y
>>> adder(3, 2)
5
```

В нашем случае мы используем ее для преобразования фрагмента кода, который в противном случае был бы выполнен сразу, в функцию, которую мы можем передавать как аргумент и которая может выполняться позже и многократно:

```
>>> def addthree(x):
...     return x + 3
...
>>> addthree(2)
5
>>> myfn = lambda: addthree(2) # отметим, addthree здесь не вызывается сразу
>>> myfn
<function <lambda> at 0x7f3b140339d8>
>>> myfn()
5
>>> myfn()
5
```

Посмотрим на нашего экстравагантного помощника `wait_for` в действии:

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
======================================================================
ERROR: test_cannot_add_empty_list_items
(functional_tests.test_list_item_validation.ItemValidationTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_list_item_validation.py",
line 15, in test_cannot_add_empty_list_items
    self.wait_for(lambda: self.assertEqual(  ❶
  File "/.../superlists/functional_tests/base.py", line 37, in wait_for
    raise e  ❷
```

```
  File "/.../superlists/functional_tests/base.py", line 34, in wait_for
    return fn()  ❷
  File "/.../superlists/functional_tests/test_list_item_validation.py",
line 16, in <lambda>  ❸
    self.browser.find_element_by_css_selector('.has-error').text,  ❸
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: .has-error
 ----------------------------------------------------------------------
Ran 1 test in 10.575s

FAILED (errors=1)
```

Порядок следования обратной трассировки немного сбивает с толку, но мы можем более или менее отследить, что произошло:

- ❶ В строке 15 нашего ФТ мы входим во вспомогательную функцию `self.wait_for`, передавая ей лямбдафицированную версию `assertEqual`.
- ❷ Мы входим в `self.wait_for` в *base.py*, где видим, что мы вызывали лямбда-функцию достаточное количество раз, чтобы выйти к `raise e`, потому что наш тайм-аут истек.
- ❸ Чтобы объяснить, откуда фактически пришло исключение, обратная трассировка возвращает нас в *test_list_item_validation.py* и внутрь тела лямбда-функции и сообщает, что та пыталась найти элемент `.has-error`, который выдал ошибку.

Передавая функции в качестве аргументов другим функциям, мы входим в пределы функционального программирования, которое может показаться немного заумным. Что касается меня, то мне потребовалось немало времени, чтобы к нему привыкнуть! Внимательно просмотрите этот фрагмент кода и программный код в ФТ пару раз, чтобы все усвоилось. Если вы по-прежнему смущены, не слишком переживайте на сей счет, пусть ваша уверенность растет по ходу работы с ним. Мы еще не раз будем использовать его в этой книге и делать его еще более функционально забавным, вот увидите.

Завершение ФТ

Мы завершим ФТ следующим образом:

functional_tests/test_list_item_validation.py (ch11l012)

```
# Домашняя страница обновляется, и появляется сообщение об ошибке, которое
# говорит, что элементы списка не должны быть пустыми
```

```
self.wait_for(lambda: self.assertEqual(
self.browser.find_element_by_css_selector('.has-error').text,
"You can't have an empty list item"
))

# Эдит пробует снова теперь с неким текстом для элемента, и это
# теперь срабатывает
self.browser.find_element_by_id('id_new_item').send_keys('Buy milk')
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Buy milk')

# Как ни странно, она решает отправить второй пустой элемент списка
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)

# Эдит получает аналогичное предупреждение на странице списка
self.wait_for(lambda: self.assertEqual(
self.browser.find_element_by_css_selector('.has-error').text,
"You can't have an empty list item"
))

# И она может его исправить, заполнив поле неким текстом
self.browser.find_element_by_id('id_new_item').send_keys('Make tea')
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)
self.wait_for_row_in_list_table('1: Buy milk')
self.wait_for_row_in_list_table('2: Make tea'),
```

Вспомогательные методы в функциональных тестах

Теперь у нас есть два вспомогательных метода: обобщенный помощник `self.wait_for` и `wait_for_row_in_list_table`. Первый – это общая утилита, ведь любому ФТ может понадобиться выполнить ожидание.

Второй также помогает предотвратить дублирование по всему программному коду, связанному с функциональным тестированием. В тот день, когда мы решим изменить реализацию характера работы нашей таблицы со списком, мы захотим точно знать, что нам придется изменить программный код ФТ только в одном месте, а не в десятках мест среди массы ФТ…

Смотрите также главу 25 и приложение Е, где дается более подробная информация о структурировании программного кода ФТ.

Разрешаю вам сделать свою собственную первую рабочую фиксацию ФТ.

Рефакторизация модульных тестов на несколько файлов

Когда мы (наконец-то!) начнем программировать наше решение, нам понадобится еще один тест для моделей в *models.py*. Прежде чем это делать, самое время подчистить модульные тесты так же, как и функциональные.

Разница будет в том, что, поскольку приложение `lists` содержит программный код реального приложения вместе с тестами, мы выделим тесты в отдельную папку:

```
$ mkdir lists/tests
$ touch lists/tests/__init__.py
$ git mv lists/tests.py lists/tests/test_all.py
$ git status
$ git add lists/tests
$ python manage.py test lists
[...]
Ran 9 tests in 0.034s

OK
$ git commit -m "Перемещены модульные тесты в папку с одним файлом"
```

Если вы получили сообщение Ran 0 tests (Выполнено 0 тестов), скорее всего, вы забыли добавить файл init с двумя подчеркиваниями[2] – этот файл должен быть там, иначе папка *tests* не является допустимым пакетом Python…

Теперь превратим *test_all.py* в два файла: один – с именем *test_views.py*, который будет содержать только тесты представлений, и другой – с именем *test_models.py*. Начнем с создания двух копий:

```
$ git mv lists/tests/test_all.py lists/tests/test_views.py
$ cp lists/tests/test_views.py lists/tests/test_models.py
```

И разберем *test_models.py* до всего одного теста – ему потребуется гораздо меньше импорта:

lists/tests/test_models.py (ch11l016)

```
from django.test import TestCase
from lists.models import Item, List

class ListAndItemModelsTest(TestCase):
    '''тест моделей списка и элемента списка'''
    [...]
```

[2] Файлы, подобные __init__.py, в Python называются dunder, сокращенно от англ. *double-underscore*, то есть двойное подчеркивание – *Прим. перев.*

В то время как *test_views.py* теряет всего один класс:

lists/tests/test_views.py (ch11l017)

```
--- a/lists/tests/test_views.py
+++ b/lists/tests/test_views.py
@@ -103,34 +104,3 @@ class ListViewTest(TestCase):
         self.assertNotContains(response, 'other list item 1')
         self.assertNotContains(response, 'other list item 2')

-
-
-class ListAndItemModelsTest(TestCase):
-
-    def test_saving_and_retrieving_items(self):
[...]
```

Выполняем тесты повторно, чтобы проверить, что все по-прежнему на месте:

```
$ python manage.py test lists
[...]
Ran 9 tests in 0.040s

OK
```

Великолепно! Вот еще один маленький, рабочий шажок:

```
$ git add lists/tests
$ git commit -m "Разделены модульные тесты на два файла"
```

Некоторым нравится помещать модульные тесты в папку tests сразу же после начала проекта. Это совершенно отличная идея; я просто посчитал, что подожду, пока это не станет необходимостью, чтобы избежать слишком большого объема служебных операций в первой главе!

Ну что ж, наши ФТ и модульный тест безупречно реорганизованы. В следующей главе мы непосредственно займемся валидацией вводимых данных.

Советы по организации тестов и рефакторизации

Используйте отдельную папку для тестов

Аналогично тому, как вы используете многочисленные файлы для хранения программного кода приложения, вам следует разбить тесты на многочисленные файлы.

- Функциональные тесты сгруппируйте по определенному признаку или истории пользователя.
- Для модульных тестов используйте папку с именем *tests* и с файлом *__init__.py* внутри.
- Вероятно, вам нужен отдельный файл с тестами для каждого тестируемого файла исходного кода. В Django это обычно *test_models.py*, *test_views.py* и *test_forms.py*.
- Имейте как минимум тест-заготовку (местозаполнитель) для каждой функции и класса.

Не забывайте про рефакторизацию в правиле «Красный, зеленый, рефакторизуй».

Вся суть наличия тестов заключается в том, чтобы позволить вам выполнять перестройку программного кода! Используйте рефакторизацию и делайте свой код (включая тесты) как можно более чистыми.

Не выполняйте рефакторизацию относительно неработающих тестов

- Вообще!
- Но ФТ, с которым вы в настоящее время работаете, не считается.
- На тест, который тестирует что-то, что вы еще не написали, иногда можно поставить декоратор пропуска.
- Но чаще всего возьмите на заметку рефакторизацию, которую хотите сделать, затем закончите то, с чем вы работаете, и выполните рефакторизацию чуть позже, когда код вернется в рабочее состояние.
- Не забывайте удалять все декораторы пропуска перед фиксацией кода! Необходимо всегда построчно просматривать разницу между версиями для отлавливания подобных нюансов.

Попробуйте обобщенный вспомогательный метод wait_for

Замечательно, когда есть конкретные вспомогательные методы, которые позволяют выполнять явные ожидания. Благодаря этому тесты становятся легкочитаемыми. Но нередко также возникает необходимость в оперативном однострочном утверждении `assert` или взаимодействии с Selenium, в которое вам потребуется добавить ожидание. По-моему, `self.wait_for` справляется с работой вполне успешно, но вы можете найти несколько иную схему, которая будет удобна для вас.

Глава 13

Валидация на уровне базы данных

На протяжении нескольких следующих глав мы будем говорить о тестировании и реализации проверки допустимости вводимых пользователем данных.

С точки зрения наполнения здесь будет довольно много материала о специфических особенностях Django и меньше – о философии методологии TDD. Это не означает, что вы ничего не узнаете о тестировании – здесь много лакомых кусочков тестирования, но они больше касаются реального вхождения в ритм, маятника методологии TDD и того, как мы выполняем работу.

Когда мы пройдем эти три краткие главы, в конце второй части я припрятал несколько забавных вещей, связанных с JavaScript (!). Затем мы перейдем к третьей части, где мы вернемся – я обещаю – к некоторым аспектам реального, практически важного обсуждения методологии TDD – модульным тестам в сравнении с интегрированными тестами, использованию имитаций, или mock-объектов, и многому другому. Не переключайтесь!

Но пока немного валидации. Сначала напомним себе, о чем говорит нам ФТ:

```
$ python3 manage.py test functional_tests.test_list_item_validation
[...]
======================================================================
ERROR: test_cannot_add_empty_list_items
(functional_tests.test_list_item_validation.ItemValidationTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_list_item_validation.py",
line 15, in test_cannot_add_empty_list_items
    self.wait_for(lambda: self.assertEqual(
[...]
  File "/.../superlists/functional_tests/test_list_item_validation.py",
line 16, in <lambda>
```

```
    self.browser.find_element_by_css_selector('.has-error').text,
[...]
selenium.common.exceptions.NoSuchElementException: Message: Невозможно
локализовать элемент: .has-error
```

Он ожидает увидеть сообщение об ошибке, если пользователь попытается ввести пустой элемент.

Валидация на уровне модели

В веб-приложении есть два места, где выполняется валидация вводимых данных: на стороне клиента (используя Javascript или свойств HTML5, как мы увидим позже) и на стороне сервера. Серверная сторона более безопасна, потому что клиентскую сторону всегда можно обойти – злоумышленно либо из-за какого-то дефекта.

Так же как на серверной стороне, в Django существует два уровня, на которых можно выполнять валидацию. Один – это уровень модели, другой – выше, на уровне форм. Мне нравится использовать более низкий уровень, когда это возможно. Отчасти потому, что я слишком привязан к базам данных и правилам их целостности, в том числе потому, что, опять-таки, это безопаснее – иногда можно забыть, какую форму вы используете для валидации вводимых данных, однако вы всегда будете использовать одну и ту же базу данных.

Контекстный менеджер self.assertRaises

Давайте спустимся вниз и напишем модульный тест на уровне моделей. Добавьте новый метод тестирования в `ListAndItemModelsTest`, который пытается создать пустой элемент списка. Этот тест интересен, потому что он проверяет, что тестируемый программный код должен поднять исключение:

lists/tests/test_models.py (ch11l018)

```
from django.core.exceptions import ValidationError
[...]

class ListAndItemModelsTest(TestCase):
    '''тест моделей списка и элемента списка'''
    [...]

    def test_cannot_save_empty_list_items(self):
        '''тест: нельзя добавлять пустые элементы списка'''
        list_ = List.objects.create()
        item = Item(list=list_, text='')
        with self.assertRaises(ValidationError):
            item.save()
```

Если вы в Python новичок, вам, возможно, никогда не встречался оператор `with`. Он используется с так называемыми менеджерами контекста, которые обертывают блок кода обычно фрагментом кода, связанного с настройкой, очисткой или обработкой ошибок. В информации о версии Python 2.5 имеется хорошая рецензия на эту тему (http://docs.python.org/release/2.5/whatsnew/pep-343.html).

Это новый метод модульного тестирования: когда нам нужно проверить, что выполнение чего-то вызывает ошибку, можно использовать менеджер контекста `self.assertRaises`. В качестве альтернативы подойдет нечто вроде этого:

```
try:
    item.save()
    self.fail('Метод save должен поднять исключение')
except ValidationError:
    pass
```

Но формулировка на основе `with` выглядит более опрятно. Теперь мы можем попытаться выполнить тест и увидим его ожидаемую неполадку:

```
    item.save()
AssertionError: ValidationError not raised
```

Причуда Django: сохранение модели не выполняет валидацию

Теперь мы раскроем одну из небольших причуд Django. *Этот тест должен уже пройти успешно*. Если вы взглянете на документацию по полям модели Django[1], то увидите, что `TextField` по умолчанию установлен в `blank=False`, то есть он *должен* запрещать пустые значения.

Тогда почему тест не показывает неполадку? Дело в том, что по слегка нелогичным историческим причинам[2] в моделях Django полная валидация в момент сохранения не выполняется. Как мы увидим позже, любые ограничения, которые фактически реализованы в базе данных, будут поднимать ошибку в момент сохранения, но SQLite не поддерживает наложение ограничений на пустые значения в текстовых столбцах, поэтому наш метод `save` молча пропускает это недопустимое значение.

Есть способ проверить, произойдет ли ограничение на уровне базы данных или нет: если это происходит на уровне базы данных, потребуется миграция, чтобы применить ограничение.

[1] См. http://bit.ly/SuxPJO

[2] См. https://groups.google.com/forum/%23!topic/django-developers/uIhzSwWHj4c

Но Django знает, что SQLite не поддерживает этот тип ограничения, поэтому если мы попытаемся выполнить `makemigrations`, то Django сообщит, что там нечего делать:

```
$ python manage.py makemigrations
No changes detected
```

Тем не менее в Django есть метод ручного выполнения полной валидации – `full_clean` (более подробная информация в документации[3]). Давайте залезем в него и убедимся, что он работает:

lists/tests/test_models.py

```
with self.assertRaises(ValidationError):
    item.save()
    item.full_clean()
```

Это приводит к тому, что тест проходит:

```
OK
```

Хорошо. Это был небольшой урок валидации в Django, и теперь есть тест, который предупредит нас, если вдруг мы забудем о необходимом условии и установим `blank=True` на текстовом (`text`) поле (попробуйте сами!).

Выведение на поверхность ошибок валидации модели в представлении

Давайте попытаемся обеспечить валидацию модели на уровне представлений и перевести ее в наши шаблоны, чтобы пользователь мог их видеть. Вот так в ряде случаев можно отобразить ошибку в нашем HTML: мы проверяем, была ли передана шаблону переменная ошибки, и если да, то отображаем ее рядом с формой:

lists/templates/base.html (ch11l020)

```
<form method="POST" action="{% block form_action %}{% endblock %}">
  <input name="item_text" id="id_new_item"
         class="form-control input-lg"
         placeholder="Enter a to-do item" />
  {% csrf_token %}
  {% if error %}
    <div class="form-group has-error">
      <span class="help-block">{{ error }}</span>
    </div>
```

[3] См. https://docs.djangoproject.com/en/1.11/ref/models/instances/%23django.db.models.Model._full_clean

```
  {% endif %}
</form>
```

Посмотрите документацию Bootstrap[4], чтобы получить дополнительную информацию об элементах управления формой.

Передача этой ошибки в шаблон является задачей функции представления. Давайте посмотрим на модульные тесты в классе `NewListTest`. Я собираюсь использовать здесь два немного отличающихся шаблона обработки ошибок.

В первом случае URL-адрес и представление для новых списков будут факультативно визуализироваться тем же шаблоном, что и для домашней страницы, но с добавлением сообщения об ошибке. Вот модульный тест для этого:

lists/tests/test_views.py (ch11l021)

```
class NewListTest(TestCase):
    '''тест нового списка'''
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        '''тест: ошибки валидации отсылаются назад в шаблон
           домашней страницы'''
        response = self.client.post('/lists/new', data={'item_text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')
        expected_error = "You can't have an empty list item"
        self.assertContains(response, expected_error)
```

При написании этого теста мы можем быть несколько раздражены URL-адресом */lists/new*, который мы вручную вводим как строковое значение. В наших тестах, представлениях и шаблонах очень много жестко кодированных URL-адресов, что нарушает принцип DRY. Я не против небольшого количества дублирования в тестах, но мы определенно должны быть начеку, когда дело касается жестко кодированных URL-адресов в наших представлениях и шаблонах. Следует взять себе на заметку исключить их в результате рефакторизации. Но мы не будем это делать сходу, потому что прямо сейчас наше приложение находится в разобранном состоянии. Сначала вернем его к работе.

Что касается нашего теста, он не срабатывает, потому что в настоящее время представление возвращает переадресацию с кодом состояния 302, а не нормальный отклик с кодом 200:

```
AssertionError: 302 != 200
```

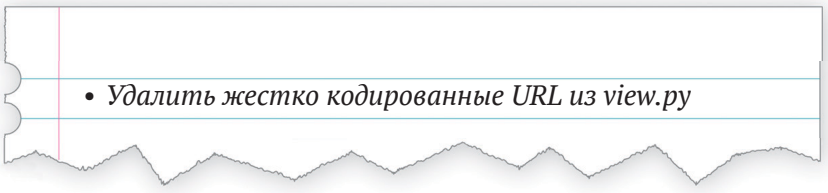
[4] См. http://getbootstrap.com/css/%23forms

Давайте попытаемся вызвать `full_clean()` в представлении:

lists/views.py

```
def new_list(request):
    '''новый список'''
    list_ = List.objects.create()
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    item.full_clean()
    return redirect(f'/lists/{list_.id}/')
```

Глядя на исходный код представления, мы находим хорошего кандидата с жестко кодированным URL-адресом, от которого избавимся. Давайте добавим пункт в наш блокнот:



Теперь валидация модели поднимает исключение, которое проходит через наше представление:

```
[...]
  File "/.../superlists/lists/views.py", line 11, in new_list
    item.full_clean()
[...]
django.core.exceptions.ValidationError: {'text': ['This field cannot be
blank.']}
```

Итак, мы испытываем первый подход: использование блока `try/except` с целью обнаружения ошибок. Повинуясь Билли-тестировщику, мы начинаем только с `try/except` и никак иначе. Тесты должны подсказать, что именно программировать дальше...

lists/views.py (ch11l025)

```
from django.core.exceptions import ValidationError
[...]

    def new_list(request):
        '''новый список'''
        list_ = List.objects.create()
        item = Item.objects.create(text=request.POST['item_text'], list=list_)
```

```
        try:
            item.full_clean()
        except ValidationError:
            pass
        return redirect(f'/lists/{list_.id}/')
```

Это возвращает нас к 302 != 200:

```
AssertionError: 302 != 200
```

Давайте тогда вернемся к шаблону, обернутому в функцию `render`, который также должен позаботиться о проверке шаблона:

lists/views.py (ch11l026)

```
    except ValidationError:
        return render(request, 'home.html')
```

И тесты теперь говорят нам поместить сообщение об ошибке в шаблон:

```
AssertionError: Ложь не является истиной: Невозможно найти строку 'Вы не
можете иметь пустой элемент списка' в отклике
```

Так и поступим, передав в шаблон новую переменную:

lists/views.py (ch11l027)

```
    except ValidationError:
        error = "You can't have an empty list item"
        return render(request, 'home.html', {"error": error})
```

Хм, похоже, это не вполне работает:

```
AssertionError: Ложь не является истиной: Невозможно найти строку 'Вы не
можете иметь пустой элемент списка' в отклике
```

Небольшая отладка на основе распечатки…

lists/tests/test_views.py

```
expected_error = "You can't have an empty list item"
print(response.content.decode())
self.assertContains(response, expected_error)
```

…покажет найти причину: Django экранировал символ апострофа для HTML[5]:

```
[...]
<span class="help-block">You can&#39;t have an empty list
item</span>
```

[5] См. https://docs.djangoproject.com/en/1.11/ref/templates/builtins/%23autoescape

Мы могли бы вставить в наш тест что-то вроде этого:

```
    expected_error = "You can&#39;t have an empty list item"
```

Но использование вспомогательной функции Django, вероятно, будет идеей получше:

lists/tests/test_views.py (ch11l029)

```
from django.utils.html import escape
[...]

        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

Это проходит!

```
Ran 11 tests in 0.047s

OK
```

Проверка, чтобы недопустимые входные данные не сохранялись в базе данных

Прежде чем мы пойдем дальше, задам один вопрос: вы заметили, что мы позволили вкрасться небольшой логической ошибке в нашу реализацию? В настоящее время мы создаем объект, даже если валидация не срабатывает:

lists/views.py

```
    item = Item.objects.create(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
    except ValidationError:
        [...]
```

Давайте добавим новый модульный тест, чтобы удостовериться, что пустые элементы списка не будут сохраняться:

lists/tests/test_views.py (ch11l030-1)

```
class NewListTest(TestCase):
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        '''тест: ошибки валидации отсылаются назад в шаблон
           домашней страницы'''
```

```
        [...]

    def test_invalid_list_items_arent_saved(self):
        '''тест: сохраняются недопустимые элементы списка'''
        self.client.post('/lists/new', data={'item_text': ''})
        self.assertEqual(List.objects.count(), 0)
        self.assertEqual(Item.objects.count(), 0)
```

Это дает:

```
[...]
Traceback (most recent call last):
  File "/.../superlists/lists/tests/test_views.py", line 40, in
test_invalid_list_items_arent_saved
    self.assertEqual(List.objects.count(), 0)
AssertionError: 1 != 0
```

Мы исправляем это вот так:

lists/views.py (ch11l030-2)

```
def new_list(request):
    '''новый список'''
    list_ = List.objects.create()
    item = Item(text=request.POST['item_text'], list=list_)
    try:
        item.full_clean()
        item.save()
    except ValidationError:
        list_.delete()
        error = "You can't have an empty list item"
        return render(request, 'home.html', {"error": error})
    return redirect(f'/lists/{list_.id}/')
```

Функциональные тесты проходят?

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
File "/.../superlists/functional_tests/test_list_item_validation.py",
line 29, in test_cannot_add_empty_list_items
    self.wait_for(lambda: self.assertEqual(
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: .has-error
```

Не совсем, но они действительно продвинулись чуть дальше. Проверяя строку 29, видим, что мы миновали первую часть теста и теперь перехо-

дим ко второй проверке того что отправка второго пустого элемента тоже показывает ошибку.

Тем не менее у нас есть немного рабочего кода, поэтому давайте зафиксируем:

```
$ git commit -am "Скорректировано новое представление списка с учетом
валидации модели"
```

Схема Django: обработка POST-запросов в том же представлении, которое генерирует форму

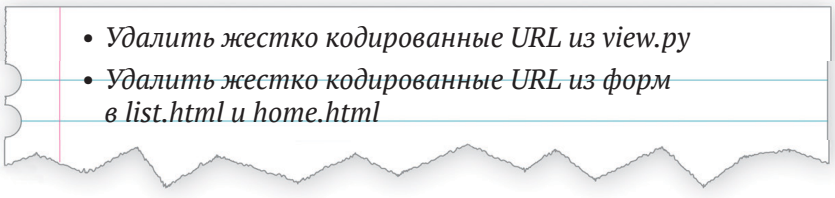
На этот раз мы будем использовать немного иной подход. Фактически он является очень распространенной схемой в Django. То есть будем использовать то же самое представление для обработки POST-запросов, что и для генерации формы, из которой они приходят. Хотя эта схема тоже не совсем укладывается в RESTful-ориентированную URL-модель, у нее есть важное преимущество: одинаковый URL-адрес может отображать форму и любые ошибки, возникающие во время обработки входных данных пользователя.

На данный момент у нас есть одно представление и URL-адрес для отображения списка, а также одно представление и URL-адрес для обработки дополнений к этому списку. Мы собираемся объединить их в одно. Поэтому в *list.html* наша форма будет иметь другую цель:

lists/templates/list.html (ch11l030)

```
{% block form_action %}/lists/{{ list.id }}/{% endblock %}
```

К слову, это еще один жестко кодированный URL-адрес. Давайте добавим его в наш оперативный список неотложных дел, и пока мы об этом размышляем, еще один есть в *home.html*:



Это сразу же нарушит наш первоначальный функциональный тест, потому что страница `view_list` пока не знает, как обрабатывать POST-запросы:

```
$ python manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
```

```
locate element: .has-error
[...]
AssertionError: '2: Сделать мушку из павлиньих перьев' not found in ['1:
Купить павлиньи перья']
```

В этом разделе мы выполняем рефакторизацию на прикладном уровне. Для этого мы изменяем или добавляем модульные тесты и затем корректируем исходный код. Мы используем функциональные тесты, которые сообщат, когда рефакторизация будет завершена и все стало работать, как прежде. Взгляните еще раз на схему в конце главы 4, если необходимо сориентироваться.

Рефакторизация: передача функциональности new_item в view_list

Давайте возьмем из `NewItemTest` все старые тесты, которые касаются сохранения POST-запросов в существующие списки, и переместим их в `ListViewTest`. При этом мы сделаем так, чтобы они указывали на базовый URL-адрес списка вместо *../add_item*:

lists/tests/test_views.py (ch11l031)

```
class ListViewTest(TestCase):
    '''тест представления списка'''

    def test_uses_list_template(self):
        '''тест: используется шаблон списка'''
        [...]

    def test_passes_correct_list_to_template(self):
        '''тест: передается правильный список в шаблон'''
        [...]

    def test_displays_only_items_for_that_list(self):
        '''тест: отображаются элементы только для этого списка'''
        [...]

    def test_can_save_a_POST_request_to_an_existing_list(self):
        '''тест: можно сохранить post-запрос в существующий список'''
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            f'/lists/{correct_list.id}/',
```

```
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new item for an existing list')
        self.assertEqual(new_item.list, correct_list)

    def test_POST_redirects_to_list_view(self):
        '''тест: post-запрос переадресуется в представление списка'''
        other_list = List.objects.create()
        correct_list = List.objects.create()

        response = self.client.post(
            f'/lists/{correct_list.id}/',
            data={'item_text': 'A new item for an existing list'}
        )
        self.assertRedirects(response, f'/lists/{correct_list.id}/')
```

Обратите внимание, что класс `NewItemTest` полностью исчезает. Я также поменял имя теста переадресации, чтобы четко обозначить, что этот тест применим только к POST-запросам.

Это изменение дает:

```
FAIL: test_POST_redirects_to_list_view (lists.tests.test_views.ListViewTest)
AssertionError: Отклик не переадресован, как ожидалось: получен код
отклика 200 (ожидался код 302) и 0 != 1
[...]
FAIL: test_can_save_a_POST_request_to_an_existing_list
(lists.tests.test_views.ListViewTest)
AssertionError: 0 != 1
```

Изменяем функцию `view_list`, чтобы она обрабатывала два типа запросов:

lists/views.py (ch11l032-1)

```
def view_list(request, list_id):
    '''представление списка'''
    list_ = List.objects.get(id=list_id)
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'], list=list_)
        return redirect(f'/lists/{list_.id}/')
    return render(request, 'list.html', {'list': list_})
```

И это дает нам успешно работающие тесты:

```
Ran 12 tests in 0.047s

OK
```

Теперь мы можем удалить представление `add_item`, поскольку оно больше не нужно… Ой, неожиданная неполадка:

```
[...]
AttributeError: модуль 'lists.views' не имеет атрибута 'add_item'
```

Это из-за того, что мы удалили представление, которое по-прежнему упоминается в *urls.py*. Удаляем его оттуда:

lists/urls.py (ch11l033)

```
urlpatterns = [
    url(r'^new$', views.new_list, name='new_list'),
    url(r'^(\d+)/$', views.view_list, name='view_list'),
]
```

И это дает нам `OK`. Давайте попробуем выполинть полный прогон ФТ:

```
$ python manage.py test
[...]
ERROR: test_cannot_add_empty_list_items
[...]

Ran 16 tests in 15.276s
FAILED (errors=1)
```

Мы вернулись к одной неполадке в нашем новом функциональном тесте. Рефакторизация функциональности `add_item` завершена. Здесь мы должны выполнить фиксацию:

```
$ git commit -am "Рефакторизовано представление списка для обработки POST-
запросов с новыми элементами"
```

Нарушил ли я правило никогда не делать рефакторизацию при наличии неработающих тестов? В данном случае это допустимо, потому что рефакторизация требуется для того, чтобы заставить новую функциональность работать. Вы определенно никогда не должны делать рефакторизацию при наличии неработающих модульных тестов. Но в моей книге это нормально, если ФТ для текущей истории пользователя, с которой вы работаете, не проходит[6].

[6] Если вы действительно хотите подчистить прогон теста, то в текущий ФТ нужно добавить декоратор пропуска или же ранний возврат, но тогда вам придется позаботиться о том, чтобы случайно не забыть об этом.

Обеспечение валидации модели в view_list

Мы по-прежнему хотим, чтобы добавление элементов в существующие списки подчинялось нашим правилам валидации модели. Давайте напишем для этого новый модульный тест; он очень похож на тест домашней страницы, только с парой корректив:

lists/tests/test_views.py (ch11l034)

```
class ListViewTest(TestCase):
    '''тест представления списка'''
    [...]

    def test_validation_errors_end_up_on_lists_page(self):
        '''тест: ошибки валидации оканчиваются на странице списков'''
        list_ = List.objects.create()
        response = self.client.post(
            f'/lists/{list_.id}/',
            data={'item_text': ''}
        )
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'list.html')
        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

Он не сработает, потому что в настоящее время наше представление не делает никакой валидации, и просто переадресует все POST-запросы:

```
    self.assertEqual(response.status_code, 200)
AssertionError: 302 != 200
```

Вот реализация:

lists/views.py (ch11l035)

```
def view_list(request, list_id):
    '''представление списка'''
    list_ = List.objects.get(id=list_id)
    error = None

    if request.method == 'POST':
        try:
            item = Item(text=request.POST['item_text'], list=list_)
            item.full_clean()
            item.save()
            return redirect(f'/lists/{list_.id}/')
        except ValidationError:
```

```
            error = "You can't have an empty list item"

    return render(request, 'list.html', {'list': list_, 'error': error})
```

Эта реализация не вызывает глубокого удовлетворения, правда? Здесь явно есть некоторое дублирование кода, блок `try/except` *в views.py* появляется дважды, и в целом ощущается какая-то неуклюжесть.

```
Ran 13 tests in 0.047s

OK
```

Хотя давайте немного подождем, прежде чем предпринять дополнительную рефакторизацию, потому что мы собираемся запрограммировать валидацию дубликатов элементов несколько иначе. Пока мы просто добавим это в наш блокнот:

- *Удалить жестко кодированные URL из view.py*
- *Удалить жестко кодированные URL из форм в list.html и home.html*
- *Удалить дублирование в логике валидации в представлениях*

Одна из причин, почему существует правило «если клюнуло трижды – рефакторизуй», заключается в том, что, если вы ждете, пока у вас не появятся три прецедента (причем каждый может немного отличаться от предыдущих), это дает вам более оптимальное представление об общей функциональности. Если же вы делаете рефакторизацию слишком рано, то можете обнаружить, что третий прецедент не совсем укладывается в перестроенный исходный код…

По крайней мере наши функциональные тесты вернулись к состоянию, когда они проходят:

```
$ python manage.py test functional_tests
[...]
OK
```

Мы вернулись в рабочее состояние, поэтому можем взглянуть на некоторые пункты в нашем блокноте. Самое время для фиксации и, возможно, для перерыва на чай.

```
$ git commit -am "Обеспечена валидация модели в представлении списка"
```

Рефакторизация: удаление жестко кодированных URL-адресов

Помните параметры `name=` в *urls.py*? Мы просто копировали их из стандартного примера, который Django предоставил нам по умолчанию, и я давал им довольно говорящие имена. Теперь мы узнаем, для чего они предназначены.

lists/urls.py

```
url(r'^new$', views.new_list, name='new_list'),
url(r'^(\d+)/$', views.view_list, name='view_list'),
```

Тег {% url %} шаблона

Мы можем заменить жестко кодированные URL-адреса в *home.html* тегом шаблона Django, в котором делается ссылка на «имя» URL-адреса:

lists/templates/home.html (ch11l036-1)

```
{% block form_action %}{% url 'new_list' %}{% endblock %}
```

Проверяем, что это не повредит модульные тесты:

```
$ python manage.py test lists
OK
```

Давайте сделаем другой шаблон. Он более интересен, потому что мы передаем ему параметр:

lists/templates/list.html (ch11l036-2)

```
{% block form_action %}{% url 'view_list' list.id %}{% endblock %}
```

Обратитесь к документации Django, чтобы получить более подробную информацию об обратном разрешении URL-адресов[7].

Мы снова выполняем тесты и убеждаемся, что все они проходят:

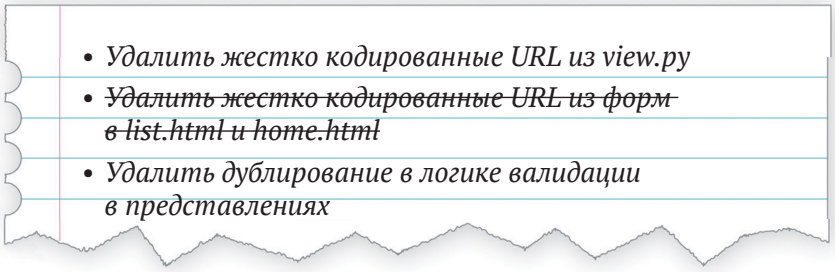
```
$ python manage.py test lists
OK
```

[7] См. https://docs.djangoproject.com/en/1.11/topics/http/urls/%23reverse-resolution-of-urls

```
$ python manage.py test functional_tests
OK
```

Превосходно!

```
$ git commit -am "Рефакторизованы/удалены жестко кодированные URL из шаблонов"
```



Использование get_absolute_url для переадресаций

Теперь давайте займемся представлениями *views.py*. Один из способов решения – сделать так же, как в шаблоне, передав имя URL и позиционный аргумент:

lists/views.py (ch11l036-3)

```
def new_list(request):
    '''новый список'''
    [...]
    return redirect('view_list', list_.id)
```

Это привело бы к тому, что модульный и функциональный тесты прошли успешно, но функция `redirect` может добиться еще большего волшебства! Поскольку модельные объекты часто связаны с определенным URL-адресом, в Django можно определить специальную функцию под названием `get_absolute_url`, которая говорит, какая страница отображает элемент. В данном случае это целесообразно, но она также полезна в администраторе Django (который я не рассматриваю в этой книге, но вскоре вы обнаружите его сами): она позволяет вам перепрыгивать от рассмотрения объекта в представлении администратора к рассмотрению объекта на работающем сайте. Я рекомендую определять `get_absolute_url` для модели всякий раз, когда существует модель, которая имеет смысл. Она вообще не требует времени, а только наличия суперпростого модульного теста в *test_models.py*:

lists/tests/test_models.py (ch11l036-4)

```
def test_get_absolute_url(self):
    '''тест: получен абсолютный url'''
    list_ = List.objects.create()
    self.assertEqual(list_.get_absolute_url(), f'/lists/{list_.id}/')
```

который дает:

```
AttributeError: У объекта'List' нет атрибута 'get_absolute_url'
```

Реализация состоит в использовании функции Django `reverse`, которая осуществляет обратное тому, что Django обычно делает с *urls.py* (см. документацию[8]):

lists/models.py (ch11l036-5)

```
from django.core.urlresolvers import reverse

    class List(models.Model):
        '''список'''

        def get_absolute_url(self):
            '''получить абсолютный url'''
            return reverse('view_list', args=[self.id])
```

Теперь можно применить ее в представлении – функция `redirect` просто берет объект, к которому мы хотим переадресовать, и за кадром автоматически применяет `get_absolute_url`!

lists/views.py (ch11l036-6)

```
def new_list(request):
    '''новый список'''
    [...]
    return redirect(list_)
```

Более подробную информацию вы можете узнать в документации Django[9]. Быстро проверяем, что модульные тесты по-прежнему проходят:

```
OK
```

Далее делаем то же самое с `view_list`:

[8] См. https://docs.djangoproject.com/en/1.11/topics/http/urls/%23reverse-resolution-of-urls

[9] См. https://docs.djangoproject.com/en/1.11/topics/http/shortcuts/%23redirect

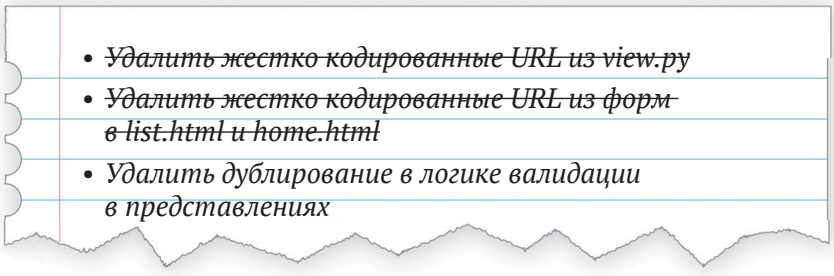
lists/views.py (ch11l036-7)

```
def view_list(request, list_id):
    '''представление списка'''
    [...]

            item.save()
            return redirect(list_)
        except ValidationError:
            error = "You can't have an empty list item"
```

И полный прогон модульного и функционального теста, чтобы убедиться, что все по-прежнему работает:

```
$ python manage.py test lists
OK
$ python manage.py test functional_tests
OK
```

Вычеркиваем из оперативного списка неотложных дел…



И фиксируем изменения…

```
$ git commit -am "Применена get_absolute_url к модели списка для удаления
дублирования url в представлениях (правило DRY)"
```

Наконец-то мы закончили с этим фрагментом! У нас есть работающая валидация на уровне модели, и по пути мы воспользовались возможностью сделать несколько рефакторизаций.

Оставшийся пункт в блокноте станет предметом следующей главы…

О валидации на уровне базы данных

Я всегда предпочитаю размещать программную логику валидации вводимых данных как можно ниже.

Валидация на уровне базы данных – это высшая гарантия целостности данных

Она может гарантировать, что независимо от того, насколько сложным становится исходный код на уровнях выше, на самом низком уровне у вас будут гарантии, что ваши данные допустимы и непротиворечивы.

Но это достигается за счет гибкости

Это преимущество не дается бесплатно! Теперь невозможно, даже временно, иметь противоречивые данные. Иногда у вас могут быть серьезные основания временно хранить данные, которые нарушают правила, вместо того чтобы не хранить вообще ничего. К примеру, вы импортируете данные из внешнего источника за несколько этапов.

И она не предназначена для удобства пользователя

Попытка хранить недопустимые данные вызовет противную ошибку целостности `IntegrityError`, которая поступит из вашей базы данных, и, возможно, пользователь увидит смущающую его страницу с ошибкой 500. Как мы узнаем в последующих главах, валидация вводимых данных на уровне форм создана с учетом пользователя, с заранее предусмотренными полезными сообщениями об ошибках, которые мы хотим ему отправлять.

Глава 14

Простая форма

В конце предыдущей главы мы остались с мыслью, что во фрагментах кода, связанного с валидацией вводимых данных в наших представлениях, слишком много повторяющегося программного кода. Для выполнения работы по валидации вводимых пользователем данных и выбору вариантов выводимых сообщений об ошибках Django рекомендует использовать классы формы. Давайте посмотрим, как это работает.

По мере прохождения данной главы мы также уделим немного времени на очистку наших модульных тестов и обеспечение того, чтобы каждый из них тестировал всего по одному компоненту за раз.

Перемещение программной логики валидации из формы

В Django сложное представление является кодом с душком. Можно ли часть этой логики вытолкнуть из формы? Либо в какие-то пользовательские методы в классе модели? Либо, возможно, даже в не-Django-вский модуль, который представляет вашу бизнес-логику?

Формы в Django имеют несколько супервозможностей:

- Они могут обрабатывать вводимые пользователем данные и проверять их на ошибки.
- Они могут использоваться в шаблонах для визуализации HTML-элементов ввода, а также сообщений об ошибках.
- И, как мы увидим позже, некоторые из них могут даже сохранять данные в базе данных.

Вам не нужно использовать все три супервозможности в каждой форме. Вы можете захотеть развернуть свою собственную HTML-разметку или со-

хранить данные своими силами. Но они являются превосходным местом, где можно содержать программную логику валидации.

Исследование API форм при помощи модульного теста

Давайте немного поэкспериментируем с формами при помощи модульного теста. Мой план состоит в том, чтобы в итеративном режиме прийти к законченному решению, и, хочется надеяться, постепенно познакомить вас с формами в той мере, чтобы они обрели для вас смысл, если никогда прежде вы их не встречали.

Сначала добавим новый файл для наших модульных тестов форм и начнем с теста, который просто смотрит на HTML-форму:

lists/tests/test_forms.py

```
from django.test import TestCase
from lists.forms import ItemForm

class ItemFormTest(TestCase):
    '''тест формы для элемента списка'''

    def test_form_renders_item_text_input(self):
        '''тест: форма отображает текстовое поле ввода'''
        form = ItemForm()
        self.fail(form.as_p())
```

`form.as_p()` выводит форму в качестве HTML-разметки. Этот модульный тест использует `self.fail` для разведочного программирования. С таким же успехом можно использовать сеанс `manage.py shell`, правда, тогда придется перезагружать программный код в случае каждого изменения.

Давайте создадим минимальную форму. Она наследует от базового класса `Form` и имеет единственное поле с именем `item_text`:

lists/forms.py

```
from django import forms

class ItemForm(forms.Form):
    '''форма для элемента списка'''
    item_text = forms.CharField()
```

Теперь мы видим сообщение о неполадке, которое говорит, как будет выглядеть автоматически сгенерированная HTML-форма:

```
    self.fail(form.as_p())
AssertionError: <p><label for="id_item_text">Item text:</label> <input
type="text" name="item_text" required id="id_item_text" /></p>
```

Это уже достаточно близко к тому, что мы имеем в *base.html*. У нас не хватает атрибута `placeholder` для замещающего текста и CSS-классов платформы Bootstrap. Давайте превратим наш модульный тест в тест, который это проверяет:

lists/tests/test_forms.py

```
class ItemFormTest(TestCase):
    '''тест формы для элемента списка'''

    def test_form_item_input_has_placeholder_and_css_classes(self):
        '''тест: поле ввода имеет атрибут placeholder и css-классы'''
        form = ItemForm()
        self.assertIn('placeholder="Enter a to-do item"', form.as_p())
        self.assertIn('class="form-control input-lg"', form.as_p())
```

Это дает ошибку, которая обосновывает необходимость реального программирования. Каким образом выполнить настройку ввода данных для поля формы? Для этого предназначен виджет. Вот он, только здесь он с заполнителем:

lists/forms.py

```
class ItemForm(forms.Form):
    '''форма для элемента'''

    item_text = forms.CharField(
        widget=forms.fields.TextInput(attrs={
            'placeholder': 'Enter a to-do item',
        }),
    )
```

Это дает:

```
AssertionError: 'class="form-control input-lg"' not found in '<p><label
for="id_item_text">Item text:</label> <input type="text" name="item_text"
placeholder="Enter a to-do item" required id="id_item_text" /></p>'
```

И затем:

lists/forms.py

```
widget=forms.fields.TextInput(attrs={
    'placeholder': 'Enter a to-do item',
    'class': 'form-control input-lg',
}),
```

Выполнение подобного рода настройки виджета станет утомительным в случае более крупной и более сложной формы. Обратитесь к django-crispy-forms и django-floppyforms за помощью. (См. соответственно https://django-crispy-forms.readthedocs.org/ и http://bit.ly/1rR5eyD)

Тесты на основе разработки: использование модульных тестов для разведочного программирования

Похоже, здесь разработка управляет тестами, не так ли? Все в порядке, время от времени это случается.

Когда вы выполняете разведку нового программного интерфейса (API), вы имеете абсолютное право некоторое время в нем покопаться, пока не будете готовы вернуться к строгой методологии TDD. Вы можете использовать интерактивную консоль или написать какой-то разведочный программный код (но пообещайте Билли-тестировщику, что вы его выбросите и позже перепишете его как надо).

Здесь мы фактически используем модульный тест как инструмент для экспериментиров с API форм. На самом деле это довольно хороший способ изучить, как он работает.

Переход на Django ModelForm

Что же дальше? Мы хотим, чтобы форма повторно использовала программный код валидации, который мы уже определили на нашей модели. Django предоставляет специальный класс под названием `ModelForm`, который может автоматически генерировать форму для модели. Как вы видите, он сконфигурирован с использованием специального атрибута с именем `Meta`:

lists/forms.py

```
from django import forms
from lists.models import Item

class ItemForm(forms.models.ModelForm):
    '''форма для элемента списка'''

    class Meta:
        model = Item
        fields = ('text',)
```

В `Meta` мы указываем, для какой модели предназначена форма и какие поля мы хотим, чтобы она использовала.

`ModelForm` делает разнообразные умные штуки, например присвоение различным типам полей осмысленных типов данных, вводимых в HTML-формы, и применение принятой по умолчанию валидации данных. Обратитесь к документации[1] за подробностями.

В результате получаем некую HTML-форму, которая выглядит по-другому:

```
AssertionError: 'placeholder="Enter a to-do item"' not found in '<p><label
for="id_text">Text:</label> <textarea name="text" cols="40" rows="10" required
id="id_text">\n</textarea></p>'
```

Она потеряла наш атрибут заполнителя `placeholder` и CSS-класс. Но вы также можете заметить, что вместо `name="item_text"` она использует `name="text"`. Пожалуй, с этим можно было бы смириться, но там используется текстовая область вместо нормального поля ввода, и это не тот пользовательский интерфейс, который подходит для нашего приложения. К счастью, вы можете переопределять виджеты относительно полей `ModelForm`, аналогично тому, как мы это делали с нормальной формой:

lists/forms.py

```
class ItemForm(forms.models.ModelForm):
    '''форма для элемента списка'''

    class Meta:
        model = Item
        fields = ('text',)
        widgets = {
            'text': forms.fields.TextInput(attrs={
                'placeholder': 'Enter a to-do item',
                'class': 'form-control input-lg',
            }),
        }
```

Это приводит к тому, что тест проходит.

Тестирование и индивидуальная настройка валидации формы

Теперь давайте убедимся, что `ModelForm` подхватила те же правила валидации, которые мы определили на модели. Мы также узнаем, как передавать данные в форму, как будто она поступила от пользователя:

[1] См. https://docs.djangoproject.com/en/1.11/topics/forms/modelforms/

lists/tests/test_forms.py (ch11l008)

```
def test_form_validation_for_blank_items(self):
    '''тест валидации формы для пустых элементов'''
    form = ItemForm(data={'text': ''})
    form.save()
```

Это дает нам:

```
ValueError: Невозможно создать Item, потому что данные не прошли валидацию.
```

Хорошо, форма не позволит сохранять данные, если вы предоставите ей пустой текст элемента.

Давайте убедимся, что она сможет использовать конкретное сообщение об ошибке, которое мы хотим. API для проверки валидации формы *до* того, как мы попытаемся сохранить любые данные, предоставляет функцию под названием `is_valid`:

lists/tests/test_forms.py (ch11l009)

```
def test_form_validation_for_blank_items(self):
    '''тест валидации формы для пустых элементов'''
    form = ItemForm(data={'text': ''})
    self.assertFalse(form.is_valid())
    self.assertEqual(
        form.errors['text'],
        ["You can't have an empty list item"]
    )
```

Вызов `form.is_valid()` возвращает `True` или `False`, но она также имеет побочный эффект, заключающийся в валидации входных данных и заполнения атрибута `errors`. Это словарь, который отображает имена полей на списки ошибок для этих полей (поле может иметь более одной ошибки).

Это дает нам:

```
AssertionError: ['This field is required.'] != ["You can't have an empty
list item"]
```

В Django уже есть стандартное сообщение об ошибке, которое мы можем представить пользователю – вы можете его использовать, если торопитесь создать свое веб-приложение. Мы же достаточно неравнодушны, чтобы сделать наше сообщение особенным. Настройка этого сообщения означает изменение `error_messages`, еще одной переменной `Meta`:

lists/forms.py (ch11l010)

```
class Meta:
    model = Item
```

```
    fields = ('text',)
    widgets = {
        'text': forms.fields.TextInput(attrs={
            'placeholder': 'Enter a to-do item',
            'class': 'form-control input-lg',
        }),
    }
    error_messages = {
        'text': {'required': "You can't have an empty list item"}
    }
```

```
OK
```

Что может быть лучше, чем париться со всеми этими строковыми значениями ошибок? Наличие константы:

lists/forms.py (ch11l011)

```
EMPTY_ITEM_ERROR = "You can't have an empty list item"
[...]

        error_messages = {
            'text': {'required': EMPTY_ITEM_ERROR}
        }
```

Повторно выполним тесты, чтобы убедиться, что они проходят… Хорошо. Теперь изменим тест:

lists/tests/test_forms.py (ch11l012)

```
from lists.forms import EMPTY_ITEM_ERROR, ItemForm
[...]

    def test_form_validation_for_blank_items(self):
        '''тест валидации формы для пустых элементов'''
        form = ItemForm(data={'text': ''})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors['text'], [EMPTY_ITEM_ERROR])
```

И тесты по-прежнему проходят:

```
OK
```

Великолепно! Подлежит тотальной фиксации:

```
$ git status # должна показать lists/forms.py и tests/test_forms.py
$ git add lists
$ git commit -m "Новая форма для элементов списка"
```

Использование формы в представлениях

Первоначально я планировал эту форму расширить, чтобы охватить проверки на уникальность и проверки на пустой элемент. Но есть своего рода дополнение к скудной методологии «Развертывай как можно раньше», которое гласит: «Объединяй программный код как можно раньше». Другими словами, во время конструирования такого программного кода для форм легко можно прийти к тому, что будешь продолжать это делать целую вечность, постепенно добавляя в форму все больше функциональности (уж я-то знаю, потому что именно этим я занимался при составлении данной главы и в итоге проделал разнообразную работу, создавая все знающий и на все готовый класс формы, прежде чем понял, что он на самом деле не будет работать для нашего элементарного случая).

Поэтому, наоборот, попытайтесь использовать свой новый фрагмент кода как можно скорее. Тогда у вас никогда не будет разбросанных повсюду неиспользованных фрагментов кода и вы как можно скорее начнете проверять свой код относительно реальной ситуации.

Итак, у нас есть класс формы, который может выводить некоторый HTML и выполнять валидацию по крайней мере одного вида ошибок. Давайте начнем его использовать! Мы сможем применить его в нашем шаблоне *base.html* и, соответственно, во всех наших представлениях.

Использование формы в представлении с GET-запросом

Давайте начнем работу в наших модульных, предназначенных тестах для представления домашней страницы. Добавим новый метод, который проверяет, что мы используем правильный вид формы:

lists/tests/test_views.py (ch11l013)

```
from lists.forms import ItemForm

class HomePageTest(TestCase):
    '''тест домашней страницы'''

    def test_uses_home_template(self):
        '''тест: использует домашний шаблон'''
        [...]

    def test_home_page_uses_item_form(self):
        '''тест: домашняя страница использует форму для элемента'''
        response = self.client.get('/')
        self.assertIsInstance(response.context['form'], ItemForm)  ❶
```

❶ `assertIsInstance` проверяет, что наша форма имеет правильный класс.

Это дает нам:

```
KeyError: 'form'
```

Поэтому мы используем форму в нашем представлении домашней страницы:

lists/views.py (ch11l014)

```
[...]
from lists.forms import ItemForm
from lists.models import Item, List

def home_page(request):
    '''домашняя страница'''
    return render(request, 'home.html', {'form': ItemForm()})
```

Хорошо, теперь попытаемся применить ее в шаблоне – меняем старый текст `<input ..>` на `{{ form.text }}`:

lists/templates/base.html (ch11l015)

```
<form method="POST" action="{% block form_action %}{% endblock %}">
    {{ form.text }}
    {% csrf_token %}
    {% if error %}
        <div class="form-group has-error">
```

`{{ form.text }}` выводит только входные данные HTML для поля `text` формы.

Глобальный поиск и замена

Правда, мы сделали одну штуку: изменили форму – она больше не использует те же самые атрибуты `id` и `name`. Вы увидите, что если выполнить функциональные тесты, они не сработают в первый же раз, как только они попытаются найти поле ввода:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_new_item"]
```

Нам нужно это исправить, и это будет сопряжено с глобальным поиском и заменой. Но прежде сделаем фиксацию, чтобы отделить переименование от изменения в логике:

```
$ git diff # просмтаривает изменения в base.html, views.py и its tests
$ git commit -am "Применена новая форма в home_page, упрощены тесты.
NB нарушена логика"
```

Теперь исправим функциональные тесты. Быстрый поиск при помощи `grep` указывает несколько мест, где мы используем `id_new_item`:

```
$ grep id_new_item functional_tests/test*
functional_tests/test_layout_and_styling.py:          inputbox =
self.browser.find_element_by_id('id_new_item')
functional_tests/test_layout_and_styling.py:          inputbox =
self.browser.find_element_by_id('id_new_item')
functional_tests/test_list_item_validation.py:
self.browser.find_element_by_id('id_new_item').send_keys(Keys.ENTER)
[...]
```

Это хороший сигнал для рефакторизации. Давайте создадим новый вспомогательный метод в *base.py*:

functional_tests/base.py (ch11l018)

```
class FunctionalTest(StaticLiveServerTestCase):
    '''функциональный тест'''
    [...]
    def get_item_input_box(self):
        '''получить поле ввода для элемента'''
        return self.browser.find_element_by_id('id_text')
```

И затем воспользуемся им повсюду – мне пришлось внести четыре изменения в *test_simple_list_cre-ation.py*, два в *test_layout_and_styling.py* и четыре в *test_list_item_validation.py*, к примеру:

functional_tests/test_simple_list_creation.py

```
    # Ей сразу же предлагается ввести элемент списка
    inputbox = self.get_item_input_box()
```

Или:

functional_tests/test_list_item_validation.py

```
    # пустой элемент списка. Она нажимает Enter на пустом поле ввода
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys(Keys.ENTER)
```

Я не стану показывать вам все строки до единой – уверен, вы справитесь с этим самостоятельно! Можно откатить `grep`, чтобы проверить, что вы все их отловили.

Мы миновали первый шаг, но теперь необходимо привести в соответствие остальную часть прикладного кода. Нам нужно найти любые вхождения старого `id` (`id_new_item`) и `name` (`item_text`) и заменить их соответственно на `id_text` и `text`:

```
$ grep -r id_new_item lists/
lists/static/base.css:#id_new_item {
```

Это одно изменение, аналогичным образом делаем для `name`:

```
$ grep -Ir item_text lists
[...]
lists/views.py:    item = Item(text=request.POST['item_text'], list=list_)
lists/views.py:            item = Item(text=request.POST['item_text'],
list=list_)
lists/tests/test_views.py:        self.client.post('/lists/new',
data={'item_text': 'A new list item'})
lists/tests/test_views.py:        response = self.client.post('/lists/new',
data={'item_text': 'A new list item'})
[...]
lists/tests/test_views.py:            data={'item_text': ''}
[...]
```

После завершения повторно выполняем модульные тесты, чтобы проверить, что все по-прежнему работает:

```
$ python manage.py test lists
[...]
.................
 ----------------------------------------------------------------------
Ran 17 tests in 0.126s

OK
```

И функциональные тесты:

```
$ python manage.py test functional_tests
[...]
  File "/.../superlists/functional_tests/test_simple_list_creation.py",
line 37, in test_can_start_a_list_for_one_user
    return self.browser.find_element_by_id('id_text')
  File "/.../superlists/functional_tests/base.py", line 51, in
get_item_input_box
    return self.browser.find_element_by_id('id_text')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [id="id_text"]
[...]
FAILED (errors=3)
```

Не вполне! Давайте посмотрим, где это происходит. Если вы проверите номер строки у одной из ошибок, то увидите, что всякий раз после отправки первого элемента поле ввода исчезало со страницы списков.

Проверив *views.py* и представление `new_list`, мы видим, что это происходит из-за того, что, обнаруживая ошибку валидации, мы фактически не передаем форму в шаблон *home.html*:

lists/views.py

```
except ValidationError:
    list_.delete()
    error = "You can't have an empty list item"
    return render(request, 'home.html', {"error": error})
```

В этом представлении нам тоже потребуется применить форму. Хотя, прежде чем вносить дополнительные изменения, сделаем фиксацию:

```
$ git status
$ git commit -am "Переименованы все id и name в элементах input элементов.
По-прежнему не работает"
```

Использование формы в представлении, принимающем POST-запросы

Теперь мы хотим скорректировать модульные тесты для представления `new_list`, особенно тот, который занимается валидацией. Посмотрим на него:

lists/tests/test_views.py

```
class NewListTest(TestCase):
    '''тест нового списка'''
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        '''тест: ошибки валидации отсылаются назад в шаблон
           домашней страницы'''
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')
        expected_error = escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

Адаптация модульных тестов к представлению new_list

Этот тест проверяет слишком много вещей сразу, здесь у нас есть возможность их разъяснить. Мы должны разделить два разных утверждения:

- Если имеется ошибка валидации, то мы должны отобразить шаблон домашней страницы с кодом состояния 200.

- Если имеется ошибка валидации, то отклик должен содержать текст ошибки.

И мы также можем добавить новое утверждение:

- Если имеется ошибка валидации, то мы должны передать объект формы в шаблон.

И пока мы тут, вместо жестко кодированного строкового значения для этого сообщения об ошибке воспользуемся константой:

lists/tests/test_views.py (ch11l023)

```
from lists.forms import ItemForm, EMPTY_ITEM_ERROR
[...]

class NewListTest(TestCase):
    '''тест нового списка'''
    [...]

    def test_for_invalid_input_renders_home_template(self):
        '''тест на недопустимый ввод: отображает домашний шаблон'''
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')

    def test_validation_errors_are_shown_on_home_page(self):
        '''тест: ошибки валидации выводятся на домашней странице'''
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertContains(response, escape(EMPTY_ITEM_ERROR))

    def test_for_invalid_input_passes_form_to_template(self):
        '''тест на недопустимый ввод: форма передается в шаблон'''
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertIsInstance(response.context['form'], ItemForm)
```

Намного лучше. Каждый тест теперь четко тестирует один компонент, и, если повезет, только один не сработает и скажет нам, что сделать:

```
$ python manage.py test lists
[...]
======================================================================
ERROR: test_for_invalid_input_passes_form_to_template
(lists.tests.test_views.NewListTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests/test_views.py", line 49, in
```

```
test_for_invalid_input_passes_form_to_template
    self.assertIsInstance(response.context['form'], ItemForm)
[...]
KeyError: 'form'

 ----------------------------------------------------------------------
Ran 19 tests in 0.041s

FAILED (errors=1)
```

Использование формы в представлении

А вот так мы используем форму в представлении:

lists/views.py

```
def new_list(request):
    '''новый список'''
    form = ItemForm(data=request.POST)  ❶
    if form.is_valid():  ❷
        list_ = List.objects.create()
        Item.objects.create(text=request.POST['text'], list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})  ❸
```

❶ Передаем данные POST-запроса в конструктор формы.

❷ Используем `form.is_valid()`, чтобы определить, является ли это допустимым отправлением или нет.

❸ Если оно недопустимое, передаем форму в шаблон вместо жестко кодированного строкового значения ошибки.

Такое представление выглядит намного лучше! И все наши тесты проходят, кроме одного:

```
    self.assertContains(response, escape(EMPTY_ITEM_ERROR))
[...]
AssertionError: False is not true : Couldn't find 'You can&#39;t have an
empty list item' in response
```

Использование формы для отображения ошибок в шаблоне

Мы получаем неполадку, потому что еще не начали использовать форму для отображения ошибок в шаблоне:

lists/templates/base.html (ch11l026)

```
<form method="POST" action="{% block form_action %}{% endblock %}">
  {{ form.text }}
  {% csrf_token %}
  {% if form.errors %}  ❶
    <div class="form-group has-error">
      <div class="help-block">{{ form.text.errors }}</div>  ❷
    </div>
  {% endif %}
</form>
```

❶ `form.errors` содержит список всех ошибок для формы.

❷ `form.text.errors` – это список ошибок только для текстового поля.

Выясним, что это делает с тестами?

```
FAIL: test_validation_errors_end_up_on_lists_page
(lists.tests.test_views.ListViewTest)
[...]
AssertionError: False is not true : Couldn't find 'You can&#39;t have an
empty list item' in response
```

Неожиданная неполадка – фактически она находится в тестах, предназначенных для заключительного представления, `view_list`. Поскольку мы изменили вид отображения ошибок во *всех* шаблонах, мы больше не показываем ошибку, которую передаем в шаблон вручную.

Это означает, что прежде чем вернуться в рабочее состояние, нам также придется переделать `view_list`.

Использование формы в другом представлении

Это представление обрабатывает как GET-, так и POST-запросы. Для начала проверим, что форма используется в GET-запросах. Для этого у нас может быть новый тест:

lists/tests/test_views.py

```
class ListViewTest(TestCase):
    '''тест представления списка'''
    [...]

    def test_displays_item_form(self):
        '''тест отображения формы для элемента'''
        list_ = List.objects.create()
        response = self.client.get(f'/lists/{list_.id}/')
```

```
        self.assertIsInstance(response.context['form'], ItemForm)
        self.assertContains(response, 'name="text"')
```

Он дает:

```
KeyError: 'form'
```

Вот минимальная реализация:

lists/views.py (ch11l028)

```
def view_list(request, list_id):
    '''представление списка'''
    [...]
    form = ItemForm()
    return render(request, 'list.html', {
        'list': list_, "form": form, "error": error
    })
```

Вспомогательный метод для нескольких коротких тестов

Далее нам надо использовать ошибки формы во втором представлении. Разобьем наш текущий одиночный тест для недопустимого случая (`test_validation_errors_end_up_on_lists_page`) на несколько отдельных:

lists/tests/test_views.py (ch11l030)

```
class ListViewTest(TestCase):
    '''тест представления списка'''
    [...]

    def post_invalid_input(self):
        '''отправляет недопустимый ввод'''
        list_ = List.objects.create()
        return self.client.post(
            f'/lists/{list_.id}/',
            data={'text': ''}
        )

    def test_for_invalid_input_nothing_saved_to_db(self):
        '''тест на недопустимый ввод: ничего не сохраняется в БД'''
        self.post_invalid_input()
        self.assertEqual(Item.objects.count(), 0)

    def test_for_invalid_input_renders_list_template(self):
        '''тест на недопустимый ввод: отображается шаблон списка'''
        response = self.post_invalid_input()
        self.assertEqual(response.status_code, 200)
```

```
        self.assertTemplateUsed(response, 'list.html')

    def test_for_invalid_input_passes_form_to_template(self):
        '''тест на недопустимый ввод: форма передается в шаблон'''
        response = self.post_invalid_input()
        self.assertIsInstance(response.context['form'], ItemForm)

    def test_for_invalid_input_shows_error_on_page(self):
        '''тест на недопустимый ввод: на странице показывается ошибка'''
        response = self.post_invalid_input()
        self.assertContains(response, escape(EMPTY_ITEM_ERROR))
```

Создав небольшую вспомогательную функцию `post_invalid_input`, мы можем сделать четыре отдельных теста без дублирования большого числа строк исходного кода.

Мы уже видели это несколько раз. Зачастую более естественным выглядит написание тестов представления в виде одного монолитного блока утверждений – «представление должно сделать то, то и это, а затем вернуть это с тем». Но разбивка на многочисленные тесты определенно стоит того. Как мы видели в предыдущих главах, она помогает определить точную проблему, которая может появиться, когда мы меняем исходный код и случайным образом вносим дефект. Вспомогательные методы – один из инструментов, которые снижают психологический барьер.

Например, мы видим всего одну неполадку, и она понятна:

```
FAIL: test_for_invalid_input_shows_error_on_page
(lists.tests.test_views.ListViewTest)
AssertionError: False is not true : Couldn't find 'You can&#39;t have an
empty list item' in response
```

Теперь давайте убедимся, что сможем соответствующим образом переписать представление для использования нашей формы:

lists/views.py

```
def view_list(request, list_id):
    '''представление списка'''
    list_ = List.objects.get(id=list_id)
    form = ItemForm()
    if request.method == 'POST':
        form = ItemForm(data=request.POST)
        if form.is_valid():
            Item.objects.create(text=request.POST['text'], list=list_)
            return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

В результате модульные тесты проходят:

```
Ran 23 tests in 0.086s

OK
```

А что насчет функциональных тестов?

```
ERROR: test_cannot_add_empty_list_items
(functional_tests.test_list_item_validation.ItemValidationTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_list_item_validation.py",
line 15, in test_cannot_add_empty_list_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: .has-error
```

Пока ничего.

Неожиданное преимущество: бесплатная валидация на стороне клиента из HTML5

Что же происходит? Давайте добавим обычный оператор `time.sleep` перед ошибкой и посмотрим (или запустим сайт вручную при помощи `manage.py runserver`, если угодно (рис. 14.1)):

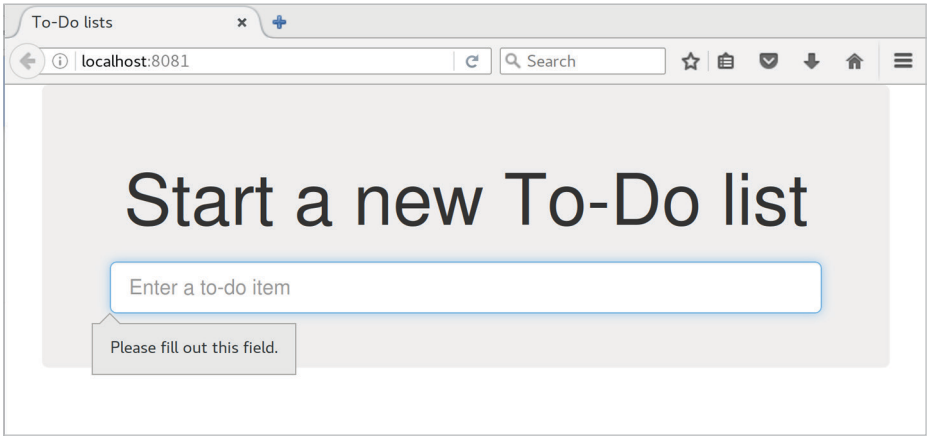


Рис. 14.1. HTML5-валидация говорит «нет»

Похоже, браузер препятствует тому, чтобы пользователь отправлял поле ввода, когда оно пустое.

Причина в том, что Django добавил в HTML-поле ввода атрибут `required`[2] (еще раз гляньте на наши первоначальные распечатки `as_p()`, если не ве-

[2] Этот функционал появился в Django 1.11.

рите). Это новый функционал HTML5[3], и сегодня браузеры выполняют некоторую валидацию на стороне клиента, если замечают это, не давая пользователям отправлять недопустимые входные данные.

Давайте изменим наш ФТ, чтобы это учесть.

functional_tests/test_list_item_validation.py (ch11l032)

```
def test_cannot_add_empty_list_items(self):
    '''тест: нельзя добавлять пустые элементы списка'''
    # Эдит открывает домашнюю страницу и случайно пытается отправить
    # пустой элемент списка. Она нажимает Enter на пустом поле ввода
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys(Keys.ENTER)

    # Браузер перехватывает запрос и не загружает страницу со списком
    self.wait_for(lambda: self.browser.find_elements_by_css_selector(
        '#id_text:invalid'
    ))

    # Эдит начинает набирать текст нового элемента и ошибка исчезает
    self.get_item_input_box().send_keys('Buy milk')
    self.wait_for(lambda: self.browser.find_elements_by_css_selector(
        '#id_text:valid'
    ))

    # И она может отправить его успешно
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table('1: Buy milk')

    # Как ни странно, Эдит решает отправить второй пустой элемент списка
    self.get_item_input_box().send_keys(Keys.ENTER)

    # И снова браузер не подчинится
    self.wait_for_row_in_list_table('1: Buy milk')
    self.wait_for(lambda: self.browser.find_elements_by_css_selector(
        '#id_text:invalid'  ❶
    ))

    # И она может исправиться, заполнив поле текстом
    self.get_item_input_box().send_keys('Make tea')
    self.wait_for(lambda: self.browser.find_elements_by_css_selector(
        '#id_text:valid'  ❷
    ))
```

[3] См. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Input%23attr-required

```
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table('1: Buy milk')
    self.wait_for_row_in_list_table('2: Make tea')
```

❶ Вместо того, чтобы выполнять проверку на наше собственное сообщение об ошибке, мы проверяем, используя псевдоселектор CSS `:invalid`, который браузер применяет к любому полю ввода HTML5 с недопустимыми входными данными.

❷ И обратный ему – в случае допустимых входных данных.

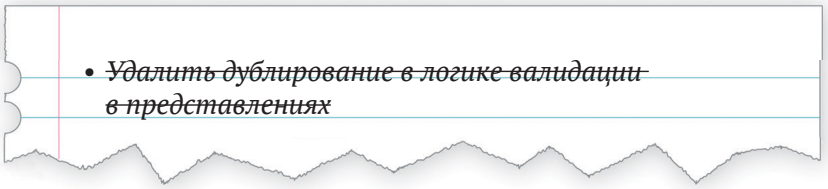
Посмотрите, насколько полезной и гибкой оказалась функция `Self.wait_for`.

Наш ФТ действительно выглядит совершенно не так, как он выглядел в самом начале, не правда ли? Уверен, что прямо сейчас у вас возникла куча вопросов. Не будем пока цепляться к мелочам, обещаю, мы об этом еще поговорим, а сейчас давайте-ка убедимся, что тесты снова проходят:

```
$ python manage.py test functional_tests
[...]
....
 ---------------------------------------------------------------------
Ran 4 tests in 12.154s

OK
```

Одобряющее похлопывание по спине

Сначала давайте похлопаем друг друга от всей души по спине: мы только что внесли существенное изменение в наше небольшое приложение. Это поле ввода со своим именем и идентификатором имеет критически важное значение для того, чтобы все заработало. Мы затронули семь или восемь разных файлов, выполнив рефакторизацию, которая была довольно запутанной..., такого рода вещи без тестов были бы причиной серьезного беспокойства. Пожалуй, я бы даже решил, что не стоит париться с кодом, который работает. Но, ввиду того что у нас есть полный комплект тестов, мы имеем возможность копаться вдоль и поперек и наводить порядок со спокойным осознанием того, что есть тесты, призванные засечь любые ошибки, которые мы допускаем. Этот простой факт делает еще более вероятным то, что вы продолжите выполнять рефакторизацию, наводить порядок, облагораживать, ухаживать за своим программным кодом, поддерживать его аккуратным, опрятным, чистым, гладким, точным, кратким, функциональным и просто хорошим.



И сайчас самое время для фиксации:

```
$ git diff
$ git commit -am "Использована форма во всех представлениях, возврат к
рабочему состоянию"
```

Не потратили ли мы уйму времени впустую?

Но как быть с нашим собственным сообщением об ошибке? Что там со всеми этими усилиями по выводу формы в HTML-шаблоне? Ведь если браузер прерывает запросы, прежде чем пользователь даже их сделает, то мы вовсе не передаем пользователю эти ошибки из Django. И ФТ больше ничего не тестирует!

Вы совершенно правы. Но есть пара-тройка причин, почему все это время не было потрачено напрасно. Прежде всего, валидации на стороне клиента недостаточно, чтобы гарантировать защиту от плохих входных данных. Поэтому, если вы действительно озабочены целостностью данных, то вам всегда будет нужна и серверная сторона тоже. Использование формы является хорошим способом инкапсуляции этой логики.

Кроме того, не все браузеры (*Кхе-кхе – Safari – кхе-кхе*) полностью реализуют HTML5, поэтому некоторые пользователи по-прежнему будут видеть наше собственное сообщение об ошибке. И если или когда мы придем к тому, что разрешим пользователям получать доступ к данным через API (см. приложение F), то наши валидационные сообщения вернутся, чтобы их использовать.

Вдобавок ко всему, мы будем в состоянии снова использовать весь наш программный код, связанный с валидацией, формами и клиентскими классами `.has-error` в следующей главе, когда займемся более продвинутой валидацией данных, которая не может быть выполнена за счет волшебства HTML5.

И даже если все это неправда – вы же не побьете себя за то, что изредка попадаете в тупик, когда программируете. Никто не способен видеть будущее, и мы должны быть сосредоточены на отыскании правильного решения, а не на подсчете времени, «потраченного впустую» на неверное решение.

Использование собственного для формы метода save

Есть еще пара вещей, которые мы можем сделать, чтобы еще больше упростить наши представления. Я уже упоминал, что формы должны уметь сохранять данные в базе данных за нас. Наш случай не будет полностью работать из коробки, потому что элемент должен знать, в каком списке сохранять, но это нетрудно наладить.

Как всегда, начнем с теста. Только чтобы проиллюстрировать проблему, давайте посмотрим, что происходит, если мы просто попытаемся вызвать `form.save()`:

lists/tests/test_forms.py (ch11l033)

```
    def test_form_save_handles_saving_to_a_list(self):
        '''тест: метод save формы обрабатывает сохранение в список'''
        form = ItemForm(data={'text': 'do me'})
        new_item = form.save()
```

Django не в восторге, потому что элемент должен принадлежать списку:

```
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

Наше решение – сказать методу `save` формы, в какой список он должен сохранить:

lists/tests/test_forms.py

```
from lists.models import Item, List
[...]

    def test_form_save_handles_saving_to_a_list(self):
        '''тест: метод save формы обрабатывает сохранение в список'''
        list_ = List.objects.create()
        form = ItemForm(data={'text': 'do me'})
        new_item = form.save(for_list=list_)
        self.assertEqual(new_item, Item.objects.first())
        self.assertEqual(new_item.text, 'do me')
        self.assertEqual(new_item.list, list_)
```

Затем мы удостоверяемся, что элемент правильно сохранен в базе данных с правильными атрибутами:

```
TypeError: save() got an unexpected keyword argument 'for_list'
```

И вот как можно реализовать наш собственный метод `save`:

lists/forms.py (ch11l035)

```
def save(self, for_list):
    self.instance.list = for_list
    return super().save()
```

Атрибут `.instance` на форме представляет объект базы данных, который модифицируется или создается. И я только что узнал об этом, когда писал эту главу! Есть другие способы заставить это работать, включая создание объекта вручную либо использование аргумента `commit=False` в функции `save`, но этот, по-моему, самый крутой. В следующей главе мы рассмотрим иной способ заставить форму «знать», для чего этот список предназначен:

```
Ran 24 tests in 0.086s

OK
```

Наконец мы можем выполнить рефакторизацию наших представлений. Сначала `new_list`:

lists/views.py

```
def new_list(request):
    '''новый список'''
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

Выполняем тест повторно, чтобы проверить, что все по-прежнему проходит:

```
Ran 24 tests in 0.086s

OK
```

И теперь `view_list`:

lists/views.py

```
def view_list(request, list_id):
    '''представление списка'''
    list_ = List.objects.get(id=list_id)
    form = ItemForm()
    if request.method == 'POST':
        form = ItemForm(data=request.POST)
```

```
        if form.is_valid():
            form.save(for_list=list_)
            return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

И у нас по-прежнему все работает:

```
Ran 24 tests in 0.111s

OK
```

и

```
Ran 4 tests in 14.367s

OK
```

Великолепно! Два наших представления теперь выглядят очень похоже на нормальные представления Django: они берут информацию из запроса пользователя, комбинируют ее с некой персонализированной логикой или информацией из URL (`list_id`), передают ее форме для валидации и возможного сохранения, а затем переадресуют либо выводят шаблон в качестве HTML.

В Django и в веб-программировании в целом формы и валидация имеют действительно важное значение, поэтому в следующей главе мы посмотрим, можно ли еще сильнее усложнить их.

Советы

Тонкие представления

Если перед вами сложные представления и вы должны писать для них много тестов, то самое время начать думать о том, можно ли эту программную логику переместить в другое место – например, в форму, как мы сделали здесь.

Еще одним возможным местом был бы собственный метод для класса модели. И, раз сложность приложения того требует, из специфических для Django файлов и для ваших собственных классов и функций, которые улавливают ключевую бизнес-логику.

Каждый тест должен тестировать одну единицу кода

Эвристика будет сомнительной, если в тесте имеется более одного утверждения. Иногда два утверждения тесно связаны, поэтому они должны быть вместе. Но часто ваш первый черновой тест заканчивается тем, что в итоге он тестирует многочисленные поведения. Имеет смысл переписать его как несколько тестов. Вспомогательные функции защищают от чрезмерного увеличения их размеров.

Глава 15

Более развитые формы

В этой главе рассмотрим использование более продвинутых форм. Мы помогли нашим пользователям избежать пустых элементов списка, а теперь поможем избежать повторяющихся элементов.

Эта глава погружается в более замысловатые подробности валидации форм Django, и вы получаете мое официальное разрешение ее пропустить, если уже все знаете об индивидуальной настройке форм Django или если читаете эту книгу ради TDD, а не ради Django.

Если вы все еще находитесь на этапе изучения Django, то здесь вы найдете полезный материал. Если вы хотите перескочить, то тоже все в порядке. Обязательно бегло просмотрите ремарку о глупости разработчика и резюме о тестировании представлений в конце главы.

Еще один ФТ на наличие повторяющихся элементов

Мы добавляем в `ItemValidationTest` второй метод тестирования:

functional_tests/test_list_item_validation.py (ch13l001)

```
def test_cannot_add_duplicate_items(self):
    '''тест: нельзя добавлять повторяющиеся элементы'''
    # Эдит открывает домашнюю страницу и начинает новый список
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys('Buy wellies')
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table('1: Buy wellies')

    # Она случайно пытается ввести повторяющийся элемент
    self.get_item_input_box().send_keys('Buy wellies')
    self.get_item_input_box().send_keys(Keys.ENTER)

    # Она видит полезное сообщение об ошибке
    self.wait_for(lambda: self.assertEqual(
        self.browser.find_element_by_css_selector('.has-error').text,
```

```
            "You've already got this in your list"
        ))
```

Зачем нужны два метода тестирования, вместо того чтобы расширить один или использовать новый файл и класс? Такое решение принимается по собственному усмотрению. Они оба тесно связаны, оба занимаются валидацией на одинаковом поле ввода, поэтому хранить их в одном файле кажется правильным. С другой стороны, они в достаточной мере логически разделены, чтобы имелся практический смысл хранить их в разных методах:

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: .has-error

Ran 2 tests in 9.613s
```

Итак, мы знаем, что первый из двух тестов теперь проходит. Вы спросите, есть ли способ выполнить только не работающий? Да, конечно:

```
$ python manage.py test functional_tests.\
test_list_item_validation.ItemValidationTest.test_cannot_add_duplicate_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: .has-error
```

Предотвращение дубликатов на уровне модели

Вот то, что мы и хотели сделать. Это новый тест, который проверяет, что повторяющиеся элементы в том же списке, поднимают ошибку:

lists/tests/test_models.py (ch09l028)

```
def test_duplicate_items_are_invalid(self):
    '''тест: повторы элементов не допустимы'''
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='bla')
    with self.assertRaises(ValidationError):
        item = Item(list=list_, text='bla')
        item.full_clean()
```

И, поняв его необходимость, мы добавляем еще один тест, чтобы удостовериться, что мы не перегнули палку в ограничениях целостности:

lists/tests/test_models.py (ch09l029)

```
def test_CAN_save_same_item_to_different_lists(self):
    '''тест: МОЖЕТ сохранить тот же элемент в разные списки'''
```

```
    list1 = List.objects.create()
    list2 = List.objects.create()
    Item.objects.create(list=list1, text='bla')
    item = Item(list=list2, text='bla')
    item.full_clean() # не должен поднять исключение
```

Я считаю нужным всегда оставлять небольшой комментарий к тестам, которые проверяют, что отдельный случай *не* должен поднять ошибку; иначе трудно увидеть, что именно тестируется.

```
AssertionError: ValidationError not raised
```

Если мы хотим сделать преднамеренно неверно, можно написать следующее:

lists/models.py (ch09l030)

```
class Item(models.Model):
    text = models.TextField(default='', unique=True)
    list = models.ForeignKey(List, default=None)
```

Это позволяет проверить, что второй тест действительно улавливает эту проблему:

```
Traceback (most recent call last):
  File "/.../superlists/lists/tests/test_models.py", line 62, in
test_CAN_save_same_item_to_different_lists
    item.full_clean() # не должен поднять исключение
    [...]
django.core.exceptions.ValidationError: {'text': ['Item with this Text already
exists.']}
```

Ремарка о том, когда выполнять проверку на глупость разработчика

В тестировании одно из решений, принимаемых по усмотрению, делается, когда требуется писать тесты, которые выглядят как «проверка, не совершили ли мы чего-то глупого». В целом их следует опасаться.

В данном случае мы написали тест, проверяющий, что невозможно сохранять повторяющиеся элементы в тот же список. И самый простой путь сделать так, чтобы тест прошел успешно в условиях, когда вы пишете наименьшее количество строк исходного кода, – сделать невозможным сохранение любых повторяющихся элементов, что оправдывает написание еще одного теста, хотя для нас будет «глупо» или «неправильно» это программировать.

Вместе с тем совершенно нереально написать тесты для всех возможных путей, которыми мы могли бы что-то запрограммировать неверно. Если у вас есть функция сумматор, которая добавляет два числа, вы можете написать пару тестов:

```
assert adder(1, 1) == 2
assert adder(2, 1) == 3
```

Однако вы имеете право предположить, что данная реализация не испорчена или не извращена преднамеренно:

```
def adder(a, b):
    '''сумматор'''
    # неправдоподобный программный код!
    if a == 3:
        return 666
    else:
        return a + b
```

Можно сказать и по-другому: вы не сделаете что-то *преднамеренно глупое*, но вы можете сделать что-то *нечаянно* глупое.

Точно так же, как и формы `ModelForm`, модели имеют класс `Meta`, где мы можем реализовать ограничение, говорящее, что элемент должен быть уникальным для конкретного списка, или, иными словами, что `text` и `list` должны быть уникальными вместе:

lists/models.py (ch09l031)

```
class Item(models.Model):
    text = models.TextField(default='')
    list = models.ForeignKey(List, default=None)

    class Meta:
        unique_together = ('list', 'text')
```

Возможно, сейчас вы захотите заглянуть в документацию Django по атрибутам модели `Meta`[1].

Небольшое отступление по поводу упорядочивания Queryset и представлений строковых значений

Тесты обнаруживают неожиданную неполадку:

```
======================================================================
FAIL: test_saving_and_retrieving_items
(lists.tests.test_models.ListAndItemModelsTest)
```

[1] См. https://docs.djangoproject.com/en/1.11/ref/models/options/

```
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/lists/tests/test_models.py", line 31, in
test_saving_and_retrieving_items
    self.assertEqual(first_saved_item.text, 'The first (ever) list item')
AssertionError: 'Item the second' != 'The first (ever) list item'
- Item the second
[...]
```

В зависимости от платформы и инсталлированной в ней СУБД SQLite, эту ошибку вы можете не увидеть. Так или иначе, продолжайте читать – программный код и тесты интересны сами по себе.

Это небольшая загадка. Немного отладочной работы на основе распечатки:

lists/tests/test_models.py

```
first_saved_item = saved_items[0]
print(first_saved_item.text)
second_saved_item = saved_items[1]
print(second_saved_item.text)
self.assertEqual(first_saved_item.text, 'The first (ever) list item')
```

Покажет нам...

```
.....Item the second
The first (ever) list item
F.....
```

Похоже, наше ограничение уникальности вмешалось в стандартное упорядочивание запросов, таких как `Item.objects.all()`. Хотя у нас уже есть неработающий тест, лучше добавить новый, который явным образом тестирует на упорядочивание:

lists/tests/test_models.py (ch09l032)

```
def test_list_ordering(self):
    '''тест упорядочения списка'''
    list1 = List.objects.create()
    item1 = Item.objects.create(list=list1, text='i1')
    item2 = Item.objects.create(list=list1, text='item 2')
    item3 = Item.objects.create(list=list1, text='3')
    self.assertEqual(
        Item.objects.all(),
        [item1, item2, item3]
    )
```

Это дает нам новую неполадку, но она не очень-то читается:

```
AssertionError: <QuerySet [<Item: Item object>, <Item: Item object>,
<Item:
Item object>]> != [<Item: Item object>, <Item: Item object>, <Item: Item
object>]
```

Нам нужно более хорошее представление строковых значений для объектов. Давайте добавим еще один модульный тест.

Обычно нужно остерегаться добавлять новые неработающие тесты, когда они уже и так есть. Это сильно усложняет чтение результата теста на выходе и просто нервирует. Сможем ли мы когда-нибудь вернуться к рабочему состоянию? В этом случае все тесты простые, поэтому я не переживаю.

lists/tests/test_models.py (ch13l008)

```
def test_string_representation(self):
    '''тест строкового представления'''
    item = Item(text='some text')
    self.assertEqual(str(item), 'some text')
```

Это дает нам:

```
AssertionError: 'Item object' != 'some text'
```

и еще две неполадки. Что ж, начнем их исправлять:

lists/models.py (ch09l034)

```
class Item(models.Model):
    [...]

    def __str__(self):
        return self.text
```

Раньше для версий Python 2.x в Django использовался метод представления строковых значений `__unicode__`. Как и подавляющую часть обработки строк, в Python 3 его упростили. Смотрите документацию Django (https://docs.djangoproject.com/en/1.11/topics/python3/%23str-and-unicode-methods).

Теперь у нас всего две неполадки, и тест упорядочивания содержит более читаемое сообщение о неполадке:

```
AssertionError: <QuerySet [<Item: i1>, <Item: item 2>, <Item: 3>]> !=
[<Item: i1>, <Item: item 2>, <Item: 3>]
```

Мы можем это исправить в `class Meta`:

lists/models.py (ch09l035)

```
class Meta:
    ordering = ('id',)
    unique_together = ('list', 'text')
```

А это будет работать?

```
AssertionError: <QuerySet [<Item: i1>, <Item: item 2>, <Item: 3>]> !=
[<Item: i1>, <Item: item 2>, <Item: 3>]
```

Сработало. Вы видите, что элементы *действительно* находятся в том же порядке, но тесты перепутаны. Я сталкиваюсь с этой проблемой постоянно – наборы queryset в Django плохо сравниваются со списками. В нашем тесте можно это исправить, конвертировав queryset в список[2]:

lists/tests/test_models.py (ch09l036)

```
self.assertEqual(
    list(Item.objects.all()),
    [item1, item2, item3]
)
```

И это работает; мы получаем комплект из полностью проходящих тестов:

```
OK
```

Новое написание старого теста модели

Приведенный выше многословный тест модели обнаружил неожиданный дефект, но пришла пора его переписать. Я написал его очень пространно, чтобы представить объектно-реляционный преобразователь Django (ORM), но теперь, когда у нас есть четкий тест на упорядочивание, мы можем получить такое же покрытие, используя несколько более коротких тестов. Удалите `test_saving_and_retrieving_items` и замените на следующий:

lists/tests/test_models.py (ch13l010)

```
class ListAndItemModelsTest(TestCase):
    '''тест моделей списка и элемента'''

    def test_default_text(self):
```

[2] Из инструментов Django для тестирования можно также обратиться к `assertSequenceEqual` из unittest и `assertQuerysetEqual`, хотя, признаюсь, был сильно озадачен, когда я в последний раз смотрел `assertQuerysetEqual`...

```
        '''тест заданного по умолчанию текста'''
        item = Item()
        self.assertEqual(item.text, '')

    def test_item_is_related_to_list(self):
        '''тест: элемент связан со списком'''
        list_ = List.objects.create()
        item = Item()
        item.list = list_
        item.save()
        self.assertIn(item, list_.item_set.all())

    [...]
```

Этого вполне хватит – подтверждения заданных по умолчанию значений атрибутов на недавно инициализированном объекте модели достаточно, чтобы выполнить санитарную проверку на то, что мы, вероятно, установили некоторые поля в *models.py*. Тест «элемент связан со списком» – это настоящий перестраховочный тест, чтобы удостовериться, что связь по внешнему ключу работает.

И пока мы тут, можно разбить этот файл на тесты для `Item` и тесты для `List` (всего один для второго из двух, `test_get_absolute_url`):

lists/tests/test_models.py (ch13l011)

```
class ItemModelTest(TestCase):
    '''тест модели элемента'''

    def test_default_text(self):
        [...]

class ListModelTest(TestCase):
    '''тест модели списка'''

    def test_get_absolute_url(self):
        '''тест: получен абсолютный url'''
        [...]
```

Это аккуратнее и чище:

```
$ python manage.py test lists
[...]
Ran 29 tests in 0.092s

OK
```

Некоторые ошибки целостности проявляются при сохранении

Финальная ремарка, прежде чем идти дальше. Помните, в главе 13 я упомянул, что некоторые ошибки целостности данных улавливаются при сохранении? Все зависит от того, обеспечивает ли база данных ограничение целостности на практике.

Попробуйте выполнить `makemigrations`, и вы увидите, что Django хочет добавить ограничение `unique_together` в саму базу данных, вместо того чтобы оставить его просто ограничением прикладного уровня:

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0005_auto_20140414_2038.py
    - Change Meta options on item
    - Alter unique_together for item (1 constraint(s))
```

Теперь, если мы изменим тест на повторяющиеся элементы, чтобы выполнить `.save` вместо `.full_clean`...

lists/tests/test_models.py

```
def test_duplicate_items_are_invalid(self):
    '''тест: повторы элементы недопустимы'''
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='bla')
    with self.assertRaises(ValidationError):
        item = Item(list=list_, text='bla')
        # item.full_clean()
        item.save()
```

Это даст:

```
ERROR: test_duplicate_items_are_invalid (lists.tests.test_models.ItemModelTest)
[...]
    return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: ограничение UNIQUE не сработало: lists_item.list_id,
lists_item.text
[...]
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Из SQLite всплывает ошибка, и она отличается от той, которую мы хотим, `IntegrityError` вместо `ValidationError`.

Давайте отменим изменения и увидим, что все тесты снова проходят:

```
$ python manage.py test lists
[...]
```

```
Ran 29 tests in 0.092s
OK
```

Пришла пора зафиксировать изменения на уровне модели:

```
$ git status # должна показать изменения в тестах + модели и новая миграция
# давайте дадим нашей новой миграции более подходящее имя
$ mv lists/migrations/0005_auto* lists/migrations/0005_list_item_unique_together.py
$ git add lists
$ git diff --staged
$ git commit -am "Реализована валидация наличия повторяющегося элемента на
уровне модели"
```

Экспериментирование с проверкой на наличие повторяющихся элементов на уровне представлений

Попробуем выполнить ФТ, чтобы просто посмотреть, где мы находимся:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: .has-error
```

Если вы не заметили, сайт показывает ошибку сервера с кодом 500. Быстрый модульный тест на уровне представления все прояснит:

lists/tests/test_views.py (ch13l014)

```
class ListViewTest(TestCase):
    '''тест представления списка'''
    [...]

    def test_for_invalid_input_shows_error_on_page(self):
        '''тест на недопустимый ввод: показывается ошибка на странице'''
        [...]

    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        '''тест: ошибки валидации повторяющегося элемента
           оканчиваются на странице списков'''
        list1 = List.objects.create()
        item1 = Item.objects.create(list=list1, text='textey')
        response = self.client.post(
            f'/lists/{list1.id}/',
            data={'text': 'textey'}
        )

        expected_error = escape("You've already got this in your list")
```

```
        self.assertContains(response, expected_error)
        self.assertTemplateUsed(response, 'list.html')
        self.assertEqual(Item.objects.all().count(), 1)
```

Дает:

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Мы хотим избежать ошибок целостности! В идеале вызов `is_valid` неким образом должен заметить ошибку дублирования, прежде чем мы попытаемся выполнить операцию сохранения, но для этого нашей форме нужно будет заранее знать, для какого списка эта операция применяется.

Пока поставим на этот тест декоратор пропуска:

lists/tests/test_views.py (ch13l015)

```
from unittest import skip
[...]

    @skip
    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
    '''тест: ошибки валидации повторяющегося элемента
       оканчиваются на странице списков'''
```

Более сложная форма для проверки на наличие повторяющихся значений

Для создания нового списка форме требуется знать только одно – текст нового элемента. Форме, которая проверяет уникальность элементов списка, также нужно знать список. Аналогично тому, как мы переопределили метод `save` на классе `ItemForm`, на этот раз мы переопределим конструктор на новом классе формы, чтобы он знал, к какому списку применяется.

Продублируем тесты для предыдущей формы, немного доработав код:

lists/tests/test_forms.py (ch13l016)

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR,
    ExistingListItemForm, ItemForm
)
[...]

class ExistingListItemFormTest(TestCase):
    '''тест формы элемента существующего списка'''

    def test_form_renders_item_text_input(self):
```

```
        '''тест: форма отображает текстовый ввод элемента'''
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_)
        self.assertIn('placeholder="Enter a to-do item"', form.as_p())

    def test_form_validation_for_blank_items(self):
        '''тест: валидация формы для пустых элементов'''
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_, data={'text': ''})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors['text'], [EMPTY_ITEM_ERROR])

    def test_form_validation_for_duplicate_items(self):
        '''тест: валидация формы для повторных элементов'''
        list_ = List.objects.create()
        Item.objects.create(list=list_, text='no twins!')
        form = ExistingListItemForm(for_list=list_, data={'text': 'no twins!'})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors['text'], [DUPLICATE_ITEM_ERROR])
```

Далее мы итеративно проходим несколько циклов TDD, пока не получим форму с собственным конструктором, который игнорирует свой аргумент `for_list`. (Я не буду показывать их все, но уверен, что вы их сделаете, не так ли? Помните, Билли следит за вами.)

lists/forms.py (ch09l071)

```
DUPLICATE_ITEM_ERROR = "You've already got this in your list"
[...]
class ExistingListItemForm(forms.models.ModelForm):
    '''форма для элемента существующего списка'''
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

В этой точке наша ошибка должна быть (для ModelForm не указан класс):

```
ValueError: ModelForm has no model class specified.
```

Затем давайте убедимся, что если заставить ее наследовать от существующей формы, это поможет исправить ошибку:

lists/forms.py (ch09l072)

```
class ExistingListItemForm(ItemForm):
    '''форма для элемента существующего списка'''
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

Отлично, это приводит нас всего к одной неполадке:

```
FAIL: test_form_validation_for_duplicate_items
(lists.tests.test_forms.ExistingListItemFormTest)
    self.assertFalse(form.is_valid())
AssertionError: True is not false
```

Следующий шаг требует некоторых знаний о внутреннем устройстве Django. Вы можете почитать об этом в документации Django по теме валидации модели и валидации формы[3].

Django использует метод под названием `validate_unique` как на формах, так и на моделях, и мы можем использовать оба варианта в сочетании с атрибутом `instance`:

lists/forms.py

```
from django.core.exceptions import ValidationError
[...]

class ExistingListItemForm(ItemForm):
    '''форма для элемента существующего списка'''

    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.instance.list = for_list

    def validate_unique(self):
        '''проверка уникальности'''
        try:
            self.instance.validate_unique()
        except ValidationError as e:
            e.error_dict = {'text': [DUPLICATE_ITEM_ERROR]}
            self._update_errors(e)
```

Здесь мы видим немного шаманства Django, но в сущности берем ошибку валидации, корректируем ее сообщение, а затем снова передаем ее в форму. И вуаля! Быстрая фиксация:

```
$ git diff
$ git commit -a
```

Использование существующей формы для элемента списка в представлении для списка

Теперь давайте убедимся, что мы сможем задействовать эту форму в нашем представлении.

[3] См. соответственно https://docs.djangoproject.com/en/1.11/ref/models/instances/%23validating-objects и https://docs.djangoproject.com/en/1.11/ref/forms/validation/

Удаляем декоратор пропуска и, пока мы здесь, можем применить новую константу. Аккуратно.

lists/tests/test_views.py (ch13l049)

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR,
    ExistingListItemForm, ItemForm,
)
[...]

    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        '''тест: ошибки валидации повторяющегося элемента
           оканчиваются на странице списков'''
        [...]
        expected_error = escape(DUPLICATE_ITEM_ERROR)
```

Это снова вызывает ошибку целостности:

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Исправление будет заключаться в переходе на использование нового класса формы. Прежде чем мы его реализуем, давайте найдем тесты, которыми проверяется класс формы, и скорректируем их:

lists/tests/test_views.py (ch13l050)

```
class ListViewTest(TestCase):
'''тест представления списка'''
[...]

    def test_displays_item_form(self):
        '''тест: отображается форма для элемента списка'''
        list_ = List.objects.create()
        response = self.client.get(f'/lists/{list_.id}/')
        self.assertIsInstance(response.context['form'], ExistingListItemForm)
        self.assertContains(response, 'name="text"')

    [...]

    def test_for_invalid_input_passes_form_to_template(self):
        '''тест на недопустимый ввод: передается форма в шаблон'''
        response = self.post_invalid_input()
        self.assertIsInstance(response.context['form'], ExistingListItemForm)
```

Это дает нам:

```
AssertionError: <ItemForm bound=False, valid=False, fields=(text)> is not an
instance of <class 'lists.forms.ExistingListItemForm'>
```

Поэтому мы можем скорректировать представление:

lists/views.py (ch13l051)

```
from lists.forms import ExistingListItemForm, ItemForm
[...]
def view_list(request, list_id):
    '''представление списка'''
    list_ = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=list_)
    if request.method == 'POST':
        form = ExistingListItemForm(for_list=list_, data=request.POST)
        if form.is_valid():
            form.save()
            [...]
```

И это *почти* все исправляет, за исключением неожиданной неполадки:

```
TypeError: save() missing 1 required positional argument: 'for_list'
```

Наш собственный метод `save` от родительского класса `ItemForm` больше не требуется. Давайте выполним на это быстрый модульный тест:

lists/tests/test_forms.py (ch13l053)

```
def test_form_save(self):
    '''тест сохранения формы'''
    list_ = List.objects.create()
    form = ExistingListItemForm(for_list=list_, data={'text': 'hi'})
    new_item = form.save()
    self.assertEqual(new_item, Item.objects.all()[0])
```

Можно сделать так, чтобы наша форма вызывала прародительский метод `save`:

lists/forms.py (ch13l054)

```
def save(self):
    return forms.models.ModelForm.save(self)
```

И мы у цели! Все модульные тесты проходят:

```
$ python manage.py test lists
[...]
Ran 34 tests in 0.082s

OK
```

Изложу здесь свое личное мнение: Я мог бы применить `super`, но предпочитаю не использовать этот метод, когда он требует аргументов, например, чтобы получить прародительский метод. Без аргументов это потрясающий способ получения непосредственного родителя. Все остальное слишком подвержено ошибкам, поэтому для остальных случаев я считаю его уродливым. YMMV[4].

Проходит и наш ФТ для валидации:

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
..
 ---------------------------------------------------------------------
Ran 2 tests in 12.048s

OK
```

И для окончательного подтверждения повторно выполняем все ФТ:

```
$ python manage.py test functional_tests
[...]
.....
 ---------------------------------------------------------------------
Ran 5 tests in 19.048s

OK
```

Ура! Самое время для заключительной фиксации и подведения итогов того, что мы узнали о тестировании представлений в последних нескольких главах.

Итоги: что мы узнали о тестировании Django

Сейчас наше приложение намного больше, чем стандартное приложение Django, оно реализует три общих уровня Django: модели, формы и представления. У нас больше нет страховочных тестов, и наш исходный код все сильнее похож на тот, который мы были бы рады видеть в реальном приложении.

У нас есть один файл модульных тестов для каждого ключевого файла исходного кода. Вот резюме самого большого из них (и находящегося на самом высоком уровне) – *test_views*:

[4] YMMV – сокращенно от Your Mileage May Vary – («У вас иной подход») – произносится программистами в свое оправдание перед оппонентом при наличии существенных расхождений в результатах работы одной и той же программы – *Прим. перев.*

Что тестировать в представлениях

Частичная распечатка, показывающая ключевые тесты и утверждения:

lists/tests/test_views.py

```
class ListViewTest(TestCase):
  def test_uses_list_template(self):
    response = self.client.get(f'/lists/{list_.id}/')  ❶
    self.assertTemplateUsed(response, 'list.html')  ❷
  def test_passes_correct_list_to_template(self):
    self.assertEqual(response.context['list'], correct_list)  ❸
  def test_displays_item_form(self):
    self.assertIsInstance(response.context['form'], ExistingListItemForm)  ❹
    self.assertContains(response, 'name=”text”')
  def test_displays_only_items_for_that_list(self):
    self.assertContains(response, 'itemey 1')  ❺
    self.assertContains(response, 'itemey 2')  ❺
    self.assertNotContains(response, 'other list item 1')  ❺
  def test_can_save_a_POST_request_to_an_existing_list(self):
    self.assertEqual(Item.objects.count(), 1)  ❻
    self.assertEqual(new_item.text, 'A new item for an existing list')  ❻
  def test_POST_redirects_to_list_view(self):
    self.assertRedirects(response, f'/lists/{correct_list.id}/')  ❻
  def test_for_invalid_input_nothing_saved_to_db(self):
    self.assertEqual(Item.objects.count(), 0)  ❻
  def test_for_invalid_input_renders_list_template(self):
    self.assertEqual(response.status_code, 200)
    self.assertTemplateUsed(response, 'list.html')  ❻
  def test_for_invalid_input_passes_form_to_template(self):
    self.assertIsInstance(response.context['form'], ExistingListItemForm)  ❼
  def test_for_invalid_input_shows_error_on_page(self):
    self.assertContains(response, escape(EMPTY_ITEM_ERROR))  ❼
  def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
    self.assertContains(response, expected_error)
    self.assertTemplateUsed(response, 'list.html')
    self.assertEqual(Item.objects.all().count(), 1)
```

❶ Использовать тестовый клиент Django.

❷ Проверить используемый шаблон. Затем проверить каждый элемент в контексте шаблона.

❸ Проверить, чтобы все объекты были правильными либо наборы queryset имели правильные элементы.

❹ Проверить, чтобы все формы имели правильный класс.

❺ Подумать о тестировании логики шаблона: любой оператор for или if может заслуживать минимального теста.

❻ В отношении представлений, которые обрабатывают POST-запросы, удостовериться, что тестируются оба случая: допустимый и недопустимый.

❼ Факультативно проверить на исправность, что форма выведена в качестве HTML и ее ошибки визуально отображаются.

Почему именно эти точки? Перейдите непосредственно к приложению B, и я покажу, почему их достаточно для того, чтобы обеспечить сохранение правильности наших представлений, в случае, если мы их перестроим, чтобы начать использовать представления, основанные на классах.

Далее мы попытаемся сделать валидацию данных более дружественной с помощью небольшого программного кода на стороне клиента. О, да вы же знаете, что это такое…

Глава 16

Пробуем окунуться, очень робко, в JavaScript

Если бы Господь хотел, чтобы мы наслаждались, то он не пожаловал бы нам свой бесценный дар неустанного страдания.

– Жан Кальвин (согласно описанию в «Кальвин и бурундуки»[1])

Наша новая программная логика валидации хороша. Но как было бы заманчиво, если бы сообщения об ошибке, связанные с повторяющимся элементом, будут исчезать, как только пользователь начинает решать проблему! Подобно тому, как это происходит с привлекательными ошибками валидации в HTML5? Для этого нам понадобился бы малюсенький кусочек JavaScript.

Мы крайне испорчены ежедневным программированием на таком жизнерадостном языке, как Python. JavaScript – это наше наказание. Хотя, как разработчику веб-приложений, вам решительно невозможно пройти мимо. Поэтому давайте окунем в него пальцы ног, причем очень-очень осторожно.

Я буду исходить из того, что вы знаете основы синтаксиса JavaScript. Если вы не читали «JavaScript: сильные стороны», пойдите и прямо сейчас сделайте себе копию! Это не очень объемная книга.

Начинаем с ФТ

Давайте добавим в класс `ItemValidationTest` новый функциональный тест:

functional_tests/test_list_item_validation.py (ch14l001)

```
def test_error_messages_are_cleared_on_input(self):
    '''тест: сообщения об ошибках очищаются при вводе'''
    # Эдит начинает список и вызывает ошибку валидации:
```

[1] См. http://onemillionpoints.blogspot.co.uk/2008/08/calvin-and-chipmunks.html

```
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys('Banter too thick')
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table('1: Banter too thick')
    self.get_item_input_box().send_keys('Banter too thick')
    self.get_item_input_box().send_keys(Keys.ENTER)

    self.wait_for(lambda: self.assertTrue(  ❶
        self.browser.find_element_by_css_selector('.has-error').is_displayed()  ❷
    ))

    # Она начинает набирать в поле ввода, чтобы очистить ошибку
    self.get_item_input_box().send_keys('a')

    # Она довольна от того, что сообщение об ошибке исчезает
    self.wait_for(lambda: self.assertFalse(
        self.browser.find_element_by_css_selector('.has-error').is_displayed()  ❷
    ))
```

❶ Мы используем еще одно обращение к нашей функции `wait_for`, на этот раз посредством `assertTrue`.

❷ `is_displayed()` сообщает, что элемент видим или невидим. Мы не можем опираться только на проверку присутствия элемента в DOM, потому что теперь мы начинаем скрывать элементы.

Как и полагается, это не срабатывает, но прежде чем мы пойдем дальше: если клюнуло трижды – рефакторизуй! У нас есть несколько мест, где мы находим элемент `.has-error` с использованием CSS. Давайте переместим его во вспомогательную функцию:

functional_tests/test_list_item_validation.py (ch14l002)

```
class ItemValidationTest(FunctionalTest):
    '''тест валидации элемента списка'''
    def get_error_element(self):
        '''получить элемент с ошибкой'''
        return self.browser.find_element_by_css_selector('.has-error')
    [...]
```

Я предпочитаю держать вспомогательные функции в классе ФТ, который их использует, и продвигать их в базовый класс, только когда они на самом деле необходимы в другом месте. Это не позволяет чрезмерно перегружать базовый класс. YAGNI.

И затем мы делаем три замены в *test_list_item_validation*, как тут:

functional_tests/test_list_item_validation.py (ch14l003)

```
    self.wait_for(lambda: self.assertEqual(
        self.get_error_element().text,
        "You've already got this in your list"
    ))
[...]
    self.wait_for(lambda: self.assertTrue(
        self.get_error_element().is_displayed()
    ))
[...]
    self.wait_for(lambda: self.assertFalse(
        self.get_error_element().is_displayed()
    ))
```

У нас ожидаемая неполадка:

```
$ python manage.py test functional_tests.test_list_item_validation
[...]
    self.get_error_element().is_displayed()
AssertionError: True is not false
```

Мы можем это зафиксировать, как первую «примерку» нашего ФТ.

Настройка элементарного исполнителя тестов на JavaScript

Выбор инструментов тестирования в мире Python и Django довольно прямолинеен. Стандартный программный пакет `unittest` идеально адекватен, и исполнитель тестов Django по умолчанию – тоже хороший выбор. Есть и несколько альтернатив: весьма популярен nose, Green – это новый парень в нашем квартале, а меня лично очень впечатляет pytest[2]. Главное, что есть четкий вариант по умолчанию, и это просто замечательно[3].

Но в мире JavaScript все не так! На работе мы используем YUI и Jest, но я решил выйти и посмотреть, есть ли какие-либо новые инструменты. И был завален вариантами: jsUnit, Qunit, Mocha, Chutzpah, Karma, Jasmine и еще много других. Причем на этом выбор не ограничивается. Когда я почти остановился на одном из них, Mocha[4], выяснилось, что теперь мне

[2] См., соответственно, http://nose.readthedocs.org/, https://github.com/CleanCut/green и http://pytest.org/

[3] Общеизвестно, что, когда вы начинаете искать инструменты Python для разработки, основанной на поведении (BDD), все становится гораздо запутанней.

[4] Исключительно потому, что он располагает исполнителем тестов NyanCat (https://mochajs.org/%23nyan).

нужно выбрать *платформу утверждений*, *генератор отчетов* и, возможно, *библиотеку по работе с объектами-имитациями*. И этому не видно конца!

В итоге я решил использовать QUnit[5], потому что он прост, на ощупь подобен модульным тестам Python и хорошо работает в паре с jQuery.

Создайте каталог с именем *tests* внутри *lists/static*, затем скачайте в него Qunit JavaScript и файлы CSS. Мы также поместим туда файл *tests.html*:

```
$ tree lists/static/tests/
lists/static/tests/
├── qunit-2.0.1.css
├── qunit-2.0.1.js
└── tests.html
```

Стереотипный код для HTML-файла QUnit выглядит следующим образом, включая тест на токсичность:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Javascript tests</title>
  <link rel="stylesheet" href="qunit-2.0.1.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="qunit-2.0.1.js"></script>

  <script>

QUnit.test("smoke test", function (assert) {
  assert.equal(1, 1, "Maths works!");
});

  </script>
</body>
</html>
```

Препарируя этот стереотипный код, важно уловить, что мы вытягиваем *qunit-2.0.1.js*, используя первый тег `<script>`, а затем используем второй, чтобы написать основную часть тестов.

Если вы откроете файл веб-браузером (для этого не надо выполнять сервер разработки, просто найдите файл на диске), то увидите нечто подобное тому, что на рис. 16.1.

[5] См. http://qunitjs.com/

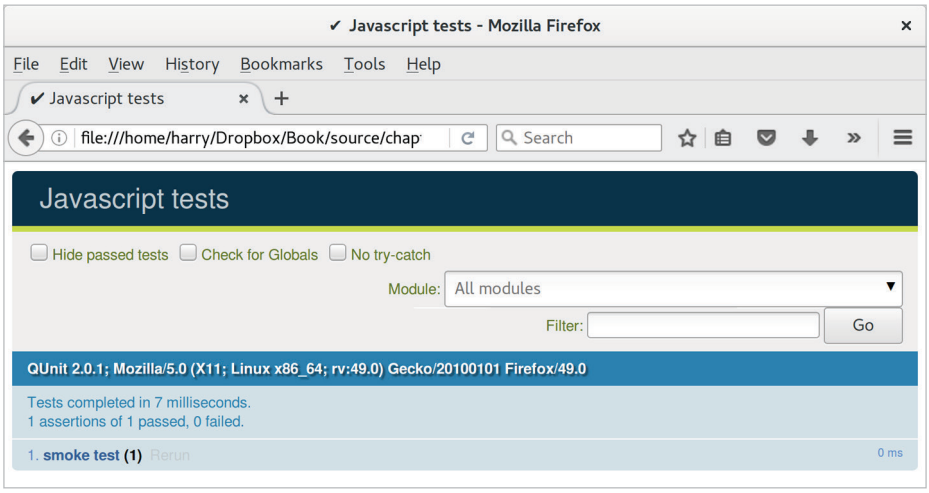


Рис. 16.1. Базовый экран QUnit

Глядя на сам тест, мы найдем много общих черт с тестами Python, которые мы писали до сих пор:

```
QUnit.test("smoke test", function (assert) {  ❶
    assert.equal(1, 1, "Maths works!");  ❷
});
```

❶ Функция `QUnit.test` определяет тестовый сценарий, немного как `def test_something(self)` в Python. Его первый аргумент – это имя теста, второй – функция для тела теста.

❷ Функция `assert.equal` – это утверждение, очень похожее на `assertEqual`; она сравнивает два аргумента. Правда, в отличие от Python, сообщение отображается как для неполадок, так и для успехов, поэтому должно формулироваться в положительном, а не в отрицательном тоне.

Почему бы не попытаться изменить эти аргументы, чтобы увидеть преднамеренную неполадку?

Использование элемента div для jQuery и фикстуры

Пора стать увереннее в возможностях нашей платформы тестирования и начать потихоньку использовать jQuery – почти незаменимую библиотеку, которая дает вам кроссбраузерно совместимый API для манипуляций с DOM.

Если вы никогда прежде не встречали jQuery, то я постараюсь давать достаточные пояснения в процессе работы, чтобы вы не заблудились. Но это не учебное руководство по jQuery. В какой-то момент при чтении этой главы вы можете посчитать нужным посвятить пару часов исследованию jQuery.

Скачайте последнюю версию jQuery с jquery.com (https://jquery.com/download/) и сохраните ее в папку *lists/static*.

Давайте применим его в нашем файле с тестами наряду с добавлением пары элементов HTML. Для начала посмотрим, сможем ли мы показывать и скрывать элемент, и напишем несколько утверждений о его видимости:

lists/static/tests/tests.html

```
  <div id="qunit-fixture"></div>

  <form>  ❶
    <input name="text" />
    <div class=”has-error”>Error text</div>
  </form>

  <script src="../jquery-3.1.1.min.js"></script>  ❷
  <script src="qunit-2.0.1.js"></script>

  <script>

QUnit.test("smoke test", function (assert) {
  assert.equal($('.has-error').is(':visible'), true);  ❸ ❹
  $('.has-error').hide();  ❺
  assert.equal($('.has-error').is(':visible'), false);  ❻
});

  </script>
```

❶ Элемент `<form>` и его содержимое представляют то, что будет на настоящей странице списка.

❷ Вот тут мы загружаем jQuery.

❸ Волшебство jQuery начинается здесь! Функция `$` – это швейцарский нож jQuery. Она используется для поиска фрагментов DOM. Его первым аргументом является селектор CSS; здесь мы говорим ей найти все элементы, которые имеют класс has-error. Она возвращает объект, который представляет один или несколько элементов

DOM. Этот объект, в свою очередь, имеет разные полезные методы, которые позволяют нам манипулировать ими либо узнавать об этих элементах.

❹ Один из таких элементов – `.is`, который определяет, соответствует ли элемент конкретному свойству CSS. Здесь мы используем `:visible`, чтобы проверить, показан элемент или скрыт.

❺ Затем мы используем метод jQuery `.hide()`, чтобы скрыть элемент div. За кулисами он динамически устанавливается на элементе `style="display: none"`.

❻ И наконец мы проверяем, что он работает, используя для этого второе утверждение `assert.equal`.

Если вы обновите страницу, то увидите, что все проходит:

Ожидаемые результаты из QUnit в браузере

```
2 assertions of 2 passed, 0 failed.
1. smoke test (2)
```

Самое время посмотреть, как работают фикстуры[6]. Давайте просто обманем этот тест:

lists/static/tests/tests.html

```
  <script>

QUnit.test("smoke test", function (assert) {
  assert.equal($('.has-error').is(':visible'), true);
  $('.has-error').hide();
  assert.equal($('.has-error').is(':visible'), false);
});
QUnit.test("smoke test 2", function (assert) {
  assert.equal($('.has-error').is(':visible'), true);
  $('.has-error').hide();
  assert.equal($('.has-error').is(':visible'), false);
});

  </script>
```

Немного неожиданно мы обнаруживаем, что один из них не срабатывает (см. рис. 16.2).

Первый тест прячет элемент div с ошибкой, поэтому, когда второй тест выполняется, он начинается с невидимого элемента.

[6] Фикстуры (англ. fixtures, оснастки) – это важная составляющая тестирования. Их основная задача – подготовить окружение с заранее фиксированным/известным состоянием для гарантии повторяемости процесса тестирования – *Прим. перев.*

Тесты QUnit не работают в предсказуемом порядке, поэтому вы не можете опираться на выполнение первого теста перед вторым. Попробуйте несколько раз нажать на обновление, и вы обнаружите, что тест, который не проходит, изменился...

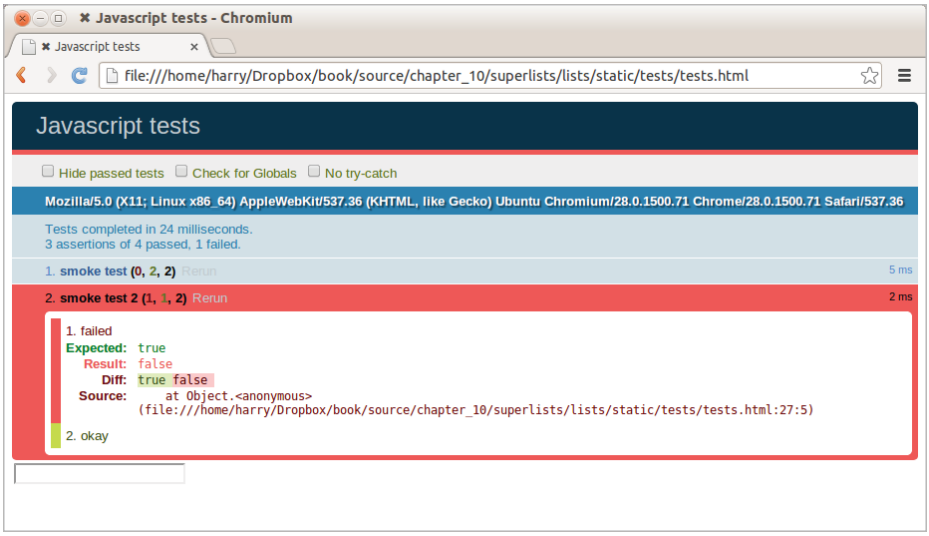


Рис. 16.2. Один из двух тестов не проходит

Нам нужен какой-то способ наведения порядка между тестами, что-то вроде методов установки `setUp` и демонтажа `tearDown`, либо как в исполнителе тестов Django, который обнуляет базу данных перед каждым тестом. Элемент div с именем `qunitfixture` – как раз то, что мы ищем. Переместите туда форму:

lists/static/tests/tests.html

```
<div id="qunit"></div>
<div id="qunit-fixture">
    <form>
        <input name="text" />
        <div class="has-error">Error text</div>
    </form>
</div>

<script src="../jquery-3.1.1.min.js"></script>
```

Как вы уже поняли, jQuery обнуляет содержимое div с фикстурами перед каждым тестом, и это возвращает нас к двум прекрасным, успешно работающим тестам:

```
4 assertions of 4 passed, 0 failed.
1. smoke test (2)
2. smoke test 2 (2)
```

Создание модульного теста на JavaScript для требуемой функциональности

Теперь, когда мы познакомились с инструментами тестирования на JavaScript, можно вернуться к одному тесту и заняться реальной задачей:

lists/static/tests/tests.html

```
  <script>

QUnit.test("errors should be hidden on keypress", function (assert) {
  $('input[name="text"]').trigger('keypress');  ❶
  assert.equal($('.has-error').is(':visible'), false);
});

  </script>
```

❶ Метод jQuery `.trigger` используется главным образом для тестирования. Он говорит «запусти событие JavaScript DOM на элементе (элементах)». Здесь мы используем событие *нажатие клавиши*, которое запускается браузером за кулисами всякий раз, когда пользователь что-то вводит в определенный элемент `input`.

Здесь jQuery таит большую сложность. Обратитесь к Quirksmode.org (http://www.quirksmode.org/dom/events/index.html) по поводу отвратительного разнообразия интерпретаций событий в разных браузерах. Причина такой популярности jQuery в том, что он заставляет весь этот хлам исчезнуть.

И это дает нам:

```
0 assertions of 1 passed, 1 failed.
1. errors should be hidden on keypress (1, 0, 1)
    1. failed
        Expected: false
        Result: true
```

Допустим, мы хотим содержать код в автономном файле JavaScript с именем *list.js*.

lists/static/tests/tests.html

```
  <script src="../jquery-3.1.1.min.js"></script>
  <script src="../list.js"></script>
  <script src="qunit-2.0.1.js"></script>

  <script>
    [...]
```

Вот минимальный код, чтобы этот тест проходил успешно:

lists/static/list.js

```
$('.has-error').hide();
```

И это работает...

```
1 assertions of 1 passed, 0 failed.
1. errors should be hidden on keypress (1)
```

Но есть очевидная проблема... Нам лучше добавить еще один тест:

lists/static/tests/tests.html

```
QUnit.test("errors should be hidden on keypress", function (assert) {
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test(“errors aren't hidden if there is no keypress”, function (assert) {
  assert.equal($('.has-error').is(':visible'), true);
});
```

Теперь мы получаем ожидаемую неполадку:

```
1 assertions of 2 passed, 1 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1, 0, 1)
    1. failed
        Expected: true
        Result: false
[...]
```

И можем сделать более реалистичную реализацию:

lists/static/list.js

```
$('input[name="text"]').on('keypress', function () {  ❶
  $('.has-error').hide();
});
```

❶ Эта строка говорит: найдите любой элемент `input`, чей атрибут `name` равен `text`, и добавьте слушатель событий, который реагирует *на* события нажатия клавиши. Слушатель событий является локальной (inline) функцией, которая скрывает все элементы, имеющие класс `.has-error`.

Это работает? Нет.

```
1 assertions of 2 passed, 1 failed.
1. errors should be hidden on keypress (1, 0, 1)
    1. failed
        Expected: false
        Result: true
[...]
2. errors aren't hidden if there is no keypress (1)
```

Проклятье! Почему это происходит?

Фикстуры, порядок выполнения и глобальное состояние: ключевые проблемы тестирования на JS

Одна из трудностей с JavaScript в целом и тестированием в частности – в понимании порядка выполнения программного кода: что и когда происходит. Когда выполняется программный код в *list.js* и когда выполняется каждый наш тест? И как это взаимодействует с глобальным состоянием, то есть с DOM веб-страницы? И как предполагается очищать фикстуры, которые мы уже видели, после каждого теста?

console.log для отладочной распечатки

Давайте добавим пару отладочных операторов `print` или операторов `console.log`:

lists/static/tests/tests.html

```
  <script>

console.log('qunit tests start');

QUnit.test("errors should be hidden on keypress", function (assert) {
  console.log('in test 1');
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
```

```
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  console.log('in test 2');
  assert.equal($('.has-error').is(':visible'), true);
});
  </script>
```

И то же самое – в нашем фактическом коде на js:

lists/static/list.js (ch14l015)

```
$('input[name="text"]').on('keypress', function () {
  console.log('in keypress handler');
  $('.has-error').hide();
});
console.log('list.js loaded');
```

Повторно выполните тесты, открыв консоль отладки браузера (обычно **Ctrl+Shift+I**), и вы увидите что-то вроде этого (рис. 16.3):

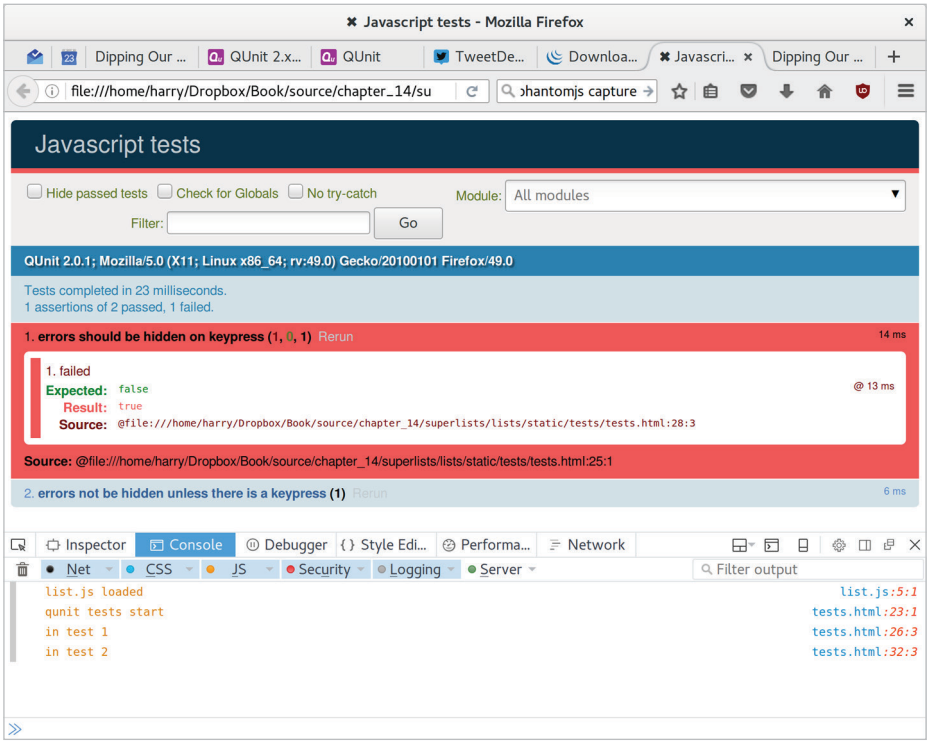


Рис. 16.3. Qunit-тесты с отладочными выходными данными операторов console.log.

Что же мы видим?

- *list.js* загружается первым. Поэтому наш слушатель событий должен быть прикреплен к элементу `input`.
- Затем загружается файл с QUnit-тестами.
- Затем выполняется каждый тест.

Но, если подумать, каждый тест будет обнулять элементы div с фикстурами, а значит, уничтожать и воссоздавать элемент `input`. Поэтому элемент `input`, который *list.js* видит и к которому прикрепляет слушателя событий, будет заменен на новый к тому времени, когда каждый тест выполняется.

Использование функции инициализации для большего контроля над временем выполнения

Нам нужен больший контроль над порядком выполнения JavaScript. Вместо того чтобы просто опираться на код в *list.js*, выполняемый всякий раз, когда он загружается тегом `<script>`, можно использовать общую схему, которая состоит в определении функции `initialize` и ее вызове в наших тестах, когда мы хотим (и позже в реальной ситуации).

lists/static/list.js

```
var initialize = function () {
  console.log('initialize called');
  $('input[name=”text”]').on('keypress', function () {
    console.log('in keypress handler');
    $('.has-error').hide();
  });
};
console.log('list.js loaded');
```

И в файле с тестами мы вызываем `initialize` с каждым тестом:

lists/static/tests/tests.html (ch14l017)

```
QUnit.test("errors should be hidden on keypress", function (assert) {
  console.log('in test 1');
  initialize();
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  console.log('in test 2');
  initialize();
```

```
  assert.equal($('.has-error').is(':visible'), true);
});
```

Теперь мы должны увидеть, что тесты проходят, и отладочные выходные данные должны иметь больше смысла:

```
2 assertions of 2 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)

list.js loaded
qunit tests start
in test 1
initialize called
in keypress handler
in test 2
initialize called
```

Ура! Давайте уберем операторы `console.log`.

lists/static/list.js

```
var initialize = function () {
  $('input[name="text"]').on('keypress', function () {
    $('.has-error').hide();
  });
};
```

И из тестов тоже…

lists/static/tests/tests.html

```
QUnit.test("errors should be hidden on keypress", function (assert) {
  initialize();
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  initialize();
  assert.equal($('.has-error').is(':visible'), true);
});
```

И для момента истины подтянем jQuery, наш сценарий, и вызовем функцию `initialize` на реальных страницах:

lists/templates/base.html (ch14l020)

```
  </div>
  <script src="/static/jquery-3.1.1.min.js"></script>
```

```
  <script src="/static/list.js"></script>

  <script>
    initialize();
  </script>

</body>
</html>
```

Размещение тегов загрузки сценариев в конце тела HTML – хорошая практика, поскольку это означает, что пользователю не приходится ожидать, когда весь ваш JavaScript загрузится, чтобы что-то увидеть на странице. Это также помогает удостовериться, что большинство DOM загрузилось до того, как любой сценарий начнет выполняться.

И-и-и... мы выполняем наш ФТ:

```
$ python manage.py test functional_tests.test_list_item_validation.\
ItemValidationTest.test_error_messages_are_cleared_on_input
[...]
Ran 1 test in 3.023s

OK
```

Ура! Это заслуживает фиксации!

```
$ git add lists/static
$ git commit -m "Добавлены jquery, qunit-тесты, list.js со слушателем нажатий"
```

Коломбо говорит: стереотипный код для onload и организация пространства имен

Да, и еще одна штука. Имя функции `initialize` слишком обобщенное. Что, если мы позже включим какой-то сторонний инструмент JavaScript, который тоже определяет функцию `initialize`? Давайте предоставим себе «пространство имен», которое вряд ли будет использовать кто-нибудь еще.

lists/static/list.js

```
window.Superlists = {};  1
window.Superlists.initialize = function () {  2
  $('input[name="text"]').on('keypress', function () {
```

```
    $('.has-error').hide();
  });
};
```

❶ Мы в явном виде объявляем объект свойством глобальной переменной `window`, давая ему имя, которое, по нашему мнению, никто больше не будет использовать.

❷ Затем мы делаем функцию `initialize` атрибутом этого объекта пространства имен.

В JavaScript есть намного более умные приемы для работы с пространствами имен, но все они гораздо сложнее, а я не такой уж эксперт, чтобы показывать вам, что куда. Если вы действительно хотите узнать больше, поищите *require.js* – это лучшее, что можно сделать или, по крайней мере, был таковым в прошедшую фемтосекунду существования JavaScript.

lists/static/tests/tests.html

```
  <script>
QUnit.test("errors should be hidden on keypress", function (assert) {
  window.Superlists.initialize();
  $('input[name="text"]').trigger('keypress');
  assert.equal($('.has-error').is(':visible'), false);
});

QUnit.test("errors aren't hidden if there is no keypress", function (assert) {
  window.Superlists.initialize();
  assert.equal($('.has-error').is(':visible'), true);
});
  </script>
```

Всякий раз, когда у вас есть какой-то JavaScript, который взаимодействует с DOM, хорошо бы обернуть его в некий стереотипный код onload, чтобы страница гарантированно полностью загружалась, прежде чем попытается что-либо сделать. Сейчас это работает, потому что мы поместили тег `<script>` прямо внизу страницы, но нельзя полагаться на это.

Стереотипный код JQuery `onload` минималистичен:

lists/templates/base.html

```
  <script>

$(document).ready(function () {
  window.Superlists.initialize();
```

```
});

  </script>
```

Читайте подробнее в документации jQuery по `.ready()`[7].

Тестирование на Javascript в цикле TDD

Вас, наверное, мучает вопрос, как эти тесты на JavaScript укладываются в наш двухконтурный цикл TDD? Ответ в том, что они играют точно ту же роль, что и наши модульные тесты на Python.

1. Написать ФТ и убедиться, что он не проходит.
2. Выяснить, какой программный код вам нужен далее: Python или JavaScript?
3. Написать модульный тест на любом языке и убедиться, что он не проходит.
4. Написать немного кода на любом языке и сделать так, чтобы этот тест прошел успешно.
5. Подчистить и повторить.

Хотите чуть больше попрактиковаться на JavaScript? Убедитесь, что у вас получится спрятать наши сообщения об ошибках, когда пользователь щелкает внутри элемента input, а также когда он просто набирает в нем текст. Также должно получиться успешно выполнить ФТ.

Мы почти готовы двинуться дальше, к третьей части книги. Последний шаг – развертывание нового исходного кода на наших серверах. Не забудьте сделать заключительную фиксацию, сначала включив *base.html*!

Несколько вещей, которые не удалось сделать

В этой главе я хотел осветить самые основы тестирования на JavaScript и то, как оно вписывается в наш поток операций TDD. Вот несколько указаний для дальнейшего исследования:

- В данный момент наш тест проверяет, что JavaScript работает на всего одной странице. Он работает, потому что мы включаем его в *base.html*, но если бы мы добавили его только в *home.html*, тесты все равно проходили бы успешно. Решение за вами, но вы могли бы написать здесь дополнительный тест.

[7] См. http://api.jquery.com/ready/

- При написании кода на JavaScript по максимуму пользуйтесь справочной информацией вашего редактора, чтобы не допустить появления распространенных подводных камней. Обратитесь к инструментам проверки синтаксиса/ошибок, таким как jslint и jshint, так называемым статическим анализаторам.
- QUnit ожидает, что вы выполняете свои тесты, используя фактический веб-браузер. Его преимущество в том, что достаточно легко можно создавать фикстуры HTML, соответствующие тому виду HTML, который содержит ваш сайт, относительно которых можно выполнять тесты. Кроме того, имеется возможность выполнять тесты JavaScript из командной строки. Мы увидим это на примере в главе 24.
- Новинками в мире разработки клиентских компонентов веб-приложения являются платформы MVC, такие как angular.js и React. В большинстве учебных руководств по их применению используется похожая на RSpec библиотека утверждений под названием Jasmine[8]. Если вы соберетесь использовать одну из них, то, вероятно, сочтете, что гораздо проще использовать Jasmine, а не Qunit.

В этой книге мы продолжим работать с увлекательными вещами, связанными с JavaScript! Взгляните на приложение, посвященное Rest API, когда будете к нему готовы.

Примечания по тестированию на JavaScript

- Одно из самых больших преимуществ Selenium состоит в том, что он позволяет тестировать работу исходного кода на JavaScript, так же, как он тестирует исходный код на Python.
- Существует много библиотек для выполнения тестов на JavaScript. Библиотека QUnit неразрывно связана с jQuery, и это главная причина, почему я ее выбрал.
- Независимо от того, какую библиотеку тестирования вы используете, вам всегда придется искать решения основной проблемы тестирования на JavaScript, которая заключается в управлении глобальным состоянием. Она включает в себя:
  - DOM / фикстуры HTML;
  - организацию пространства имен;
  - понимание порядка выполнения и его контроль.
- В действительности я вовсе не хочу сказать, что JavaScript ужасен. Он может быть довольно забавным. Но я повторю еще раз: обязательно прочтите книгу «*JavaScript: сильные стороны*».

[8] См. https://jasmine.github.io/

Глава 17

Развертывание нового программного кода

Самое время развернуть наш блестящий новый валидационный код на работающих серверах. Это даст нам шанс второй раз увидеть в действии наши автоматизированные сценарии развертывания.

Здесь я хочу сказать огромное спасибо Эндрю Годвину и всей команде Django. Вплоть до Django 1.7 у меня был длинный раздел, полностью посвященный миграциям. Теперь миграции реально работают, поэтому я смог полностью исключить этот раздел. Спасибо за отличную работу, ребята!

Развертывание на промежуточном сервере

Мы начинаем с промежуточного сервера:

```
$ git push
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
Disconnecting from superlists-staging.ottg.eu... done.
```

Перезапустите Gunicorn:

```
elspeth@server:$ sudo systemctl restart gunicorn-superlists-staging.ottg.eu
```

И выполните тесты относительно промежуточного сервера:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
OK
```

Развертывание на реальном сервере

Предполагая, что все в порядке, мы выполняем развертывание относительно реального сервера:

```
$ fab deploy:host=elspeth@superlists.ottg.eu
elspeth@server:$ sudo service gunicorn-superlists.ottg.eu restart
```

Что делать, если вы видите ошибку базы данных

Поскольку миграции вводят новое ограничение целостности, вы можете обнаружить, что применить их не удастся, потому что какие-то существующие данные нарушают это ограничение.

В этой точке у вас есть два варианта:

- Удалить базу данных на сервере и попробовать еще раз. В конце концов, это всего лишь игрушечный проект!
- Подробнее узнать о миграциях данных. Смотрите приложение D.

Итоги: маркировка нового релиза командой git tag

И теперь нужно промаркировать выпуск в системе управления версиями (VCS) – очень важно все время отслеживать то, что работает:

```
$ git tag -f LIVE # нуждается в -f, потому что мы заменяем старый тег
$ export TAG=`date +DEPLOYED-%F/%H%M`
$ git tag $TAG
$ git push -f origin LIVE $TAG
```

Не все любят использовать команду `push -f` и обновлять существующий тег. Вместо этого они используют некий номер версии для тегирования своих релизов. Используйте что угодно, лишь бы работало.

На этой ноте мы можем завершить вторую часть и пойти дальше, к более захватывающим темам, которые содержатся в третьей части. Ждете не дождетесь!

Анализ процедуры развертывания

Мы уже выполнили несколько развертываний, так что это самое подходящее время для небольшого резюме:

- Выполнить `git push` с последней версией кода.
- Выполнить развертывание на промежуточном сервере и запустить функциональные тесты относительно промежуточного сервера.
- Выполнить развертывание на реальном сервере.
- Промаркировать релиз.

Процедуры развертывания эволюционируют и становятся все более сложными по мере развития проектов. Они представляют ту область, которую трудно поддерживать при обилии ручных проверок и процедур, если с самого начала не постараться все автоматизировать. О процессе развертывания можно сказать еще очень многое, но эта задача выходит за рамки настоящей книги. Настоятельно рекомендую обратиться к приложению С, посвященному обеспечению работы при помощи Ansible, и почитать о непрерывном развертывании.

Часть III

Основы TDD и Django

«Что? Просто невероятно! Еще один раздел? Гарри, я измотан, уже триста страниц, я не справлюсь с еще целой грудой страниц книги. Особенно, если она называется продвинутой. Может, я как-нибудь обойдусь и пропущу ее, а?»

Ну уж нет, нельзя! Может, эта часть и называется продвинутой, но она полна действительно важных тем о методологии TDD и веб-разработке. Никак *нельзя ее пропускать*. Если уж на то пошло, она даже важнее двух первых.

Мы поговорим об интеграции и тестировании сторонних систем. Современная веб-разработка во многом касается многократного использования существующих компонентов. Мы охватим объекты-имитации и изоляцию тестов, которая в действительности лежит в основе методологии TDD и является методом, который используется для всех, кроме самых простых, кодовых баз. Мы поговорим об отладке на стороне сервера и фикстурах тестирования; о том, как устанавливать среду непрерывной интеграции. Ни одна из этих тем не является для вашего проекта шикарным дополнением за отдельную плату из разряда «не хотите – как хотите» – все они имеют жизненно важное значение!

В этой части книги кривая обучения неизбежно станет чуть более крутой. Возможно, вам придется перечитать этот раздел пару раз, прежде чем вы усвоите приведенный здесь материал. Или, возможно, вы решите, что ничего не заработает с первой попытки и вам потребуется заняться отладкой в одиночку. Будьте настойчивы! Чем труднее – тем слаще потом награда. И я всегда рад помочь: если вы зашли в тупик – просто напишите мне на адрес obeythetestinggoat@gmail.com.

Ну, давайте же, я обещаю: лучшее еще впереди!

Глава 18

Аутентификация пользователя, импульсное исследование и внедрение его результатов

Наш красивый сайт списков работает уже несколько дней, и пользователи начинают возвращаться к нам с отзывами. «Нам понравился этот сайт, – говорят они, – но мы продолжаем терять свои списки. Запоминать URL-адреса очень трудно. Было бы замечательно, если бы сайт помнил, какие списки мы создали».

Помните знаменитое высказывание Генри Форда про более быстрых лошадей?[1] Всегда, когда вы слышите пожелание пользователя, важно заглянуть немного глубже и подумать: в чем здесь реальная потребность? И как задействовать крутую новую технологию, которую я хотел бы испытать?

Совершенно очевидно, что в данном случае потребность – иметь некую учетную запись пользователя на сайте. Так что давайте без дальнейшей суматохи погрузимся в аутентификацию.

Естественно, мы не собираемся валандаться с запоминанием паролей самостоятельно – помимо того, что технология обеспечения безопасного хранения паролей пользователей безнадежно застряла в 90-х, она представляет собой настоящий кошмар, и мы скорее оставим ее кому-то другому. А сами воспользуемся чем-то более забавным под названием «беспарольная аутентификация».

(Если вы *настаиваете* на хранении своих паролей, то используемый по умолчанию модуль аутентификации auth в Django готов и с нетерпением ждет вас. Он изящен и прямолинеен, и я предоставлю вам право разобраться в нем самостоятельно.)

Беспарольная аутентификация

Какую систему аутентификации мы могли бы применить, чтобы избежать хранения паролей самостоятельно? OAuth? Openid? «Вход на основе учет-

[1] Здесь имеется в виду его высказывание «Если бы я спросил людей, чего они хотят, они бы попросили более быструю лошадь» – *Прим. перев.*

ной записи Facebook»? Тьфу. Для меня все они имеют недопустимый жуткий подтекст: с какой это стати Google или Facebook будет знать, на какие сайты я захожу и когда?

В первом издании я использовал экспериментальный проект под названием Persona, приготовленный несколькими замечательными техно-хиппи-идеалистами в Mozilla. Печально, но этот проект закрыт.

Вместо него я нашел забавный подход к аутентификации, который называется беспарольным, но его можно назвать «просто используй электронную почту».

Данную систему придумал кто-то, кто был недоволен необходимостью создавать новые пароли для большого количества веб-сайтов, кто обнаружил, что использует случайные одноразовые пароли, даже не пытаясь их запомнить, и пользуется функцией «я забыл свой пароль» всякий раз, когда ему нужно снова войти в систему. Вы можете прочитать об этом в статье, ссылка на которую приведена в сноске[2].

Суть идеи в том, чтобы для проверки идентификационных данных использовать электронную почту. Если вы собираетесь задействовать функцию «я забыл свой пароль», то вы так или иначе доверяете электронной почте, так почему бы просто не пройти весь путь до конца? Всякий раз, когда кто-то хочет войти в систему, мы генерируем для него уникальный URL-адрес, посылаем ему ссылку на этот URL по электронной почте, а затем он кликает по ней, чтобы зайти на сайт.

Это отнюдь не идеальная система, и по сути дела в ней много тонкостей, которые следует тщательно продумать, прежде чем она станет по-настоящему хорошим решением задачи авторизации пользователей для производственного веб-сайта. Но у нас просто забавный игрушечный проект, так что давайте-ка попробуем.

Разведочное программирование, или Импульсное исследование

Единственное, что я знал о беспарольной аутентификации до написания этой главы, – краткое описание, которое я привел выше. Я никогда не видел исходного кода, который был бы с ним вязан, и действительно не знал, откуда начинать его проектировать.

В главах 13 и 14 мы увидели, что можно применять модульное тестирование в качестве метода исследования нового API или инструмента. Но иногда нам нужно просто что-то набросать, без каких-либо тестов вообще, только чтобы посмотреть, что это работает, чтобы научиться использовать или почувствовать. Это просто замечательно. При изучении нового инструмента и при исследовании нового возможного решения, иногда имеет смысл отложить в сторону строгий процесс TDD и создать небольшой прототип без

[2] См. https://medium.com/@ninjudd/passwords-are-obsolete-9ed56d483eb%23.cx8iber30

тестов либо с очень небольшим числом тестов. Билли-тестировщик не возражает, если вы какое-то время будете смотреть на вещи с другой стороны.

Этот вид деятельности по прототипированию в силу хорошо известных причин часто называется импульсным исследованием[3] (spiking)[4].

Первое, что я сделал, – обратился к существующим пакетам аутентификации Python и Django, таким как django-allauth и python-social-auth[5], но на этом этапе они выглядели сверхсложными (и тем увлекательнее будет написание собственного исходного кода!)

Тогда я с головой погрузился в работу, и после нескольких тупиков и неправильных поворотов у меня было что-то, что почти работает. Я устрою для вас экскурсию, а затем мы пройдемся по всему процессу реализации с внедрением результатов импульсного исследования (de-spiking) – то есть заменим прототип протестированным, готовым к эксплуатации программным кодом.

Вы можете смело добавить этот код в собственный сайт и затем поиграть с настройками, попытаться войти в систему под своим адресом электронной почты и убедиться, что это действительно работает.

Открытие ветки для результатов импульсного исследования

Прежде чем приступить к импульсному исследованию, неплохо бы открыть новую ветку, чтобы вы по-прежнему использовали свою систему управления версиями, не переживая, что фиксации импульсов смешаются с производственным кодом:

```
$ git checkout -b passwordless-spike
```

Предлагаю отслеживать некоторые моменты, которые мы надеемся узнать из импульсного исследования.

- *Как отправлять электронные сообщения*
- *Генерирование и распознавание уникальных маркеров*
- *Как аутентифицировать кого-то в Django*
- *Какие шаги нужно пройти пользователю?*

[3] См. http://stackoverflow.com/questions/249969/why-are-tdd-spikes-called-spikes

[4] Импульсное исследование (spiking), по определению разработчиков, – это вид архитектурного проектирования, когда через систему пропускается импульс (spike), чтобы придать процессу разработки новое, зачастую неожиданное направление, обычно с использованием нестандартных или новых технологий или приемов – *Прим. перев.*

[5] См., соответственно, http://www.intenct.nl/projects/django-allauth/ и https://github.com/omab/python-social-auth

Авторизация на стороне клиента в пользовательском интерфейсе

Начнем с клиентской части: набросаем фактическую форму, чтобы вводить свой адрес электронной почты в навигационную панель, и ссылку выхода из системы для пользователей, которые уже аутентифицированы:

lists/templates/base.html (ch16l001)

```
<body>
  <div class="container">

    <div class="navbar">
      {% if user.is_authenticated %}
        <p>Logged in as {{ user.email}}</p>
        <p><a id="id_logout" href="{% url 'logout' %}">Log out</a></p>
      {% else %}
        <form method="POST" action ="{% url 'send_login_email' %}">
          Enter email to log in: <input name="email" type="text" />
          {% csrf_token %}
        </form>
      {% endif %}
    </div>

    <div class="row">
    [...]
```

Отправка электронных писем из Django

Принцип входа в систему будет примерно таким:

- Когда пользователь хочет войти в систему, мы генерируем уникальный секретный маркер, сохраняем его в базе данных, связанной с электронной почтой пользователя, и отправляем ему.
- Затем он проверяет свою электронную почту, где уже будет ссылка на URL-адрес, который включает этот маркер.
- Когда пользователь нажимает на ссылку, мы проверяем, существует ли маркер в базе данных, и если да, то он регистрируется как соответствующий пользователь.

Сначала мы готовим приложение для всего, что связано с учетными записями:

```
$ python manage.py startapp accounts
```

И подключаем *urls.py* по крайней мере с одним URL-адресом. В *superlists/urls.py* на верхнем уровне…

superlists/urls.py (ch16l003)

```
from django.conf.urls import include, url
from lists import views as list_views
from lists import urls as list_urls
from accounts import urls as accounts_urls

urlpatterns = [
    url(r'^$', list_views.home_page, name='home'),
    url(r'^lists/', include(list_urls)),
    url(r'^accounts/', include(accounts_urls)),
]
```

И в модуль учетных записей файла *urls.py*:

accounts/urls.py (ch16l004)

```
from django.conf.urls import url
from accounts import views

urlpatterns = [
    url(r'^send_email$', views.send_login_email, name='send_login_email'),
]
```

Вот представление, отвечающее за создание маркера, связанного с адресом электронной почты, который пользователь вставляет в регистрационную форму:

accounts/views.py (ch16l005)

```
import uuid
import sys
from django.shortcuts import render
from django.core.mail import send_mail

from accounts.models import Token

def send_login_email(request):
    '''выслать ссылку на логин по почте'''
    email = request.POST['email']
    uid = str(uuid.uuid4())
    Token.objects.create(email=email, uid=uid)
    print('saving uid', uid, 'for email', email, file=sys.stderr)
    url = request.build_absolute_uri(f'/accounts/login?uid={uid}')
    send_mail(
        'Your login link for Superlists',
        f'Use this link to log in:\n\n{url}',
```

```
        'noreply@superlists',
        [email],
    )
    return render(request, 'login_email_sent.html')
```

Чтобы это работало, нам потребуется замещающее сообщение, которое подтверждает, что электронное письмо было отправлено:

accounts/templates/login_email_sent.html (ch16l006)

```
<html>
<h1>Email sent</h1>

<p>Check your email, you'll find a message with a link that will log you
into the site.</p>

</html>
```

(Вы видите, какой этот код неуклюжий – мы хотим интегрировать этот шаблон с нашим *base.html* в реальной версии.)

Чтобы функция Django `send_mail` заработала, нужно сообщить Django адрес нашего почтового сервера. Пока я просто использую свою учетную запись gmail[6]. Вы можете использовать любой почтовый сервер, который вам нравится, – главное, чтобы он поддерживал SMTP.

superlists/settings.py (ch16l007)

```
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'obeythetestinggoat@gmail.com'
EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_PASSWORD')
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Если вы хотите использовать Gmail, вам, скорее всего, придется посетить страницу настроек безопасности учетных записей Google. Если вы используете двухфакторную авторизацию, нужно установить специфичный для приложения пароль (см. https://myaccount.google.com/apppasswords). Если нет, то вам все равно нужно разрешить доступ для менее безопасных приложений. Возможно, для этой цели стоит создать новую учетную запись Google.

[6] Разве я не написал целое введение, предупреждая о последствиях для конфиденциальности при входе в систему с использованием Google, чтобы потом взять и использовать **gmail**? Да, это противоречие (признаюсь, когда-нибудь я отойду от gmail!). Но в данном случае я его использую для тестирования и совершенно не навязываю Google своим читателям.

Использование переменных окружения для предотвращения секретов в исходном коде

Рано или поздно в каждом проекте нужно придумать способ справляться с секретами – почтовыми паролями или ключами API, которыми вы не хотите делиться со всем миром. Если у вас приватный репозиторий, вполне можно просто хранить их в Git, но зачастую дело не в этом. Это также пересекается с потребностью иметь разные настройки на сервере разработки и производственном сервере. (Помните, как в главе 11 мы решали этот вопрос с установкой SECRET_KEY в Django?)

Общая схема[7] подразумевает применение для этого вида настроек параметров конфигурации переменных окружения. Именно это я и делаю при помощи `os.environ.get`.

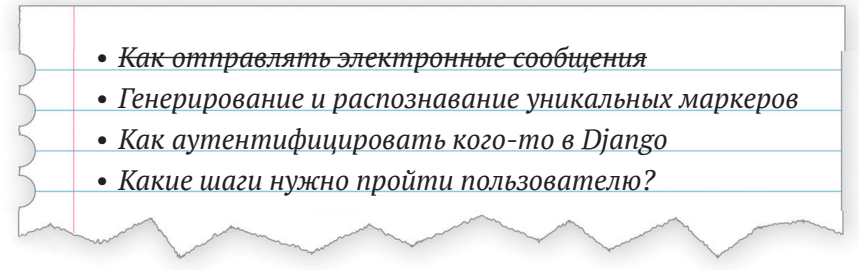
Чтобы это заработало, нужно установить переменную окружения в оболочке, в которой выполняется мой сервер разработки.

```
$ export EMAIL_PASSWORD="sekrit"
```

Позже мы также добавим ее на промежуточный сервер.

Хранение маркеров в базе данных

Как наши дела?



Нам потребуется модель для хранения маркеров в базе данных – они соединяют адрес электронной почты с уникальным идентификатором. Это довольно просто.

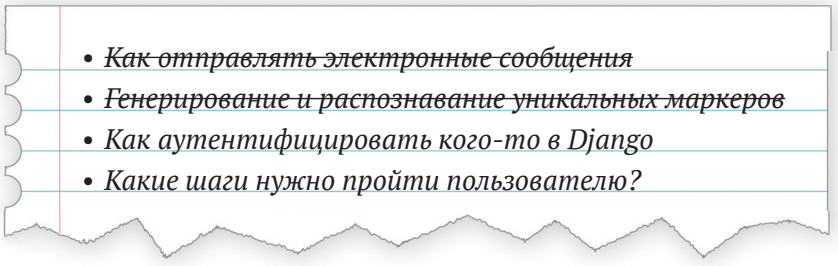
accounts/models.py (ch16l008)

```
from django.db import models

class Token(models.Model):
    '''маркер'''
```

[7] См. https://12factor.net/config

```
    email = models.EmailField()
    uid = models.CharField(max_length=255)
```

Индивидуализированные модели аутентификации



Пока мы возимся с моделями, давайте начнем экспериментировать с аутентификацией в Django. Первым делом нам нужна модель пользователя. В первом варианте данной главы индивидуализированные модели пользователей были в Django новинкой, поэтому я взял документацию по модулю Django auth[8] и попытался собрать самую простую из возможных:

accounts/models.py (ch16l009)

```
[...]
from django.contrib.auth.models import (
    AbstractBaseUser, BaseUserManager, PermissionsMixin
)

class ListUser(AbstractBaseUser, PermissionsMixin):
    '''пользователь списка'''
    email = models.EmailField(primary_key=True)
    USERNAME_FIELD = 'email'
    #REQUIRED_FIELDS = ['email', 'height']

    objects = ListUserManager()

    @property
    def is_staff(self):
        return self.email == 'harry.percival@example.com'

    @property
    def is_active(self):
        return True
```

[8] См. https://docs.djangoproject.com/en/1.11/topics/auth/customizing/

Это то, что я называю минимальной моделью пользователя! Одно поле и никакой ерунды с именем/фамилией/именем пользователя, и, главное, никакого пароля! Пусть этим занимаются другие!

Но, опять-таки, вы можете видеть, что этот исходный код еще не готов к производственной среде, начиная закомментированными строками и кончая жестко кодированным адресом электронной почты. Нам предстоит много чего подчистить, когда мы начнем внедрять результаты импульсного исследования.

Чтобы это заработало, вам нужен менеджер моделей пользователя:

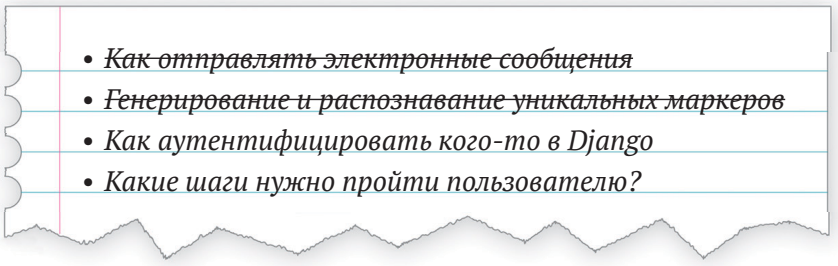
accounts/models.py (ch16l010)

```
[...]
class ListUserManager(BaseUserManager):
    '''менеджер пользователя списка'''

    def create_user(self, email):
        '''создать пользователя'''
        ListUser.objects.create(email=email)

    def create_superuser(self, email, password):
        '''создать суперпользователя'''
        self.create_user(email)
```

На данном этапе не стоит задумываться о том, что такое менеджер моделей. Пока он нам просто нужен и все. И все просто работает само по себе. Когда мы будем внедрять результаты импульсного исследования, мы изучим каждый фрагмент кода, который фактически перейдет в производственную среду, и убедимся, что полностью его понимаем.

Завершение индивидуализированной авторизации в Django

Мы почти у цели – наш последний шаг объединяет распознавание маркера и затем фактическую авторизацию пользователя. После его завершения мы в общем-то сможем вычеркнуть все пункты из нашего блокнота:



Вот представление, которое обрабатывает переход по ссылке в почте:

accounts/views.py (ch16l011)

```
import uuid
import sys
from django.contrib.auth import authenticate
from django.contrib.auth import login as auth_login
from django.core.mail import send_mail
from django.shortcuts import redirect, render
[...]

def login(request):
    '''регистрация в системе'''
    print('login view', file=sys.stderr)
    uid = request.GET.get('uid')
    user = authenticate(uid=uid)
    if user is not None:
        auth_login(request, user)
    return redirect('/')
```

Функция `authenticate` вызывает инфраструктуру аутентификации Django, которую мы конфигурируем с помощью индивидуализированного серверного процессора аутентификации, работа которого заключается в валидации uid и возврате пользователя с правильной электронной почтой.

Можно все это сделать и непосредственно в представлении, но нам не сложно придать вещам такую структуру, какую ожидает Django. Это способствует довольно аккуратному разделению компетенций.

accounts/authentication.py (ch16l012)

```
import sys
from accounts.models import ListUser, Token

class PasswordlessAuthenticationBackend(object):
    '''серверный процессор беспарольной аутентификации'''

    def authenticate(self, uid):
        '''авторизовать'''
        print('uid', uid, file=sys.stderr)
        if not Token.objects.filter(uid=uid).exists():
            print('no token found', file=sys.stderr)
            return None
        token = Token.objects.get(uid=uid)
        print('got token', file=sys.stderr)
        try:
```

```
            user = ListUser.objects.get(email=token.email)
            print('got user', file=sys.stderr)
            return user
        except ListUser.DoesNotExist:
            print('new user', file=sys.stderr)
            return ListUser.objects.create(email=token.email)

    def get_user(self, email):
        '''получить пользователя'''
        return ListUser.objects.get(email=email)
```

Опять-таки, там достаточно много отладочной распечатки и дублированного кода, то есть не того, что мы хотели бы иметь в производственной среде, но это работает...

Наконец, представление выхода из системы:

accounts/views.py (ch16l013)

```
from django.contrib.auth import login as auth_login, logout as auth_logout
[...]

def logout(request):
    '''выход из системы'''
    auth_logout(request)
    return redirect('/')
```

Добавим вход и выход из системы в наш *urls.py*.

accounts/urls.py (ch16l014)

```
from django.conf.urls import url
from accounts import views

urlpatterns = [
    url(r'^send_email$', views.send_login_email, name='send_login_email'),
    url(r'^login$', views.login, name='login'),
    url(r'^logout$', views.logout, name='logout'),
]
```

Почти у цели! В *settings.py* активируем серверный процессор аутентификации и новое приложение по работе с учетными записями:

```
INSTALLED_APPS = [
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
```

```
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
    'accounts',
]

AUTH_USER_MODEL = 'accounts.ListUser'
AUTHENTICATION_BACKENDS = [
    'accounts.authentication.PasswordlessAuthenticationBackend',
]

MIDDLEWARE = [
[...]
```

Быстрая команда `makemigrations`, чтобы материализовать модели маркера и пользователя:

```
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0001_initial.py
    - Create model ListUser
    - Create model Token
```

И команда `migrate`, чтобы создать базу данных:

```
$ python manage.py migrate
[...]
Running migrations:
   Applying accounts.0001_initial... OK
```

И на этом, пожалуй, все! Почему бы не запустить сервер разработки командой `runserver` и не посмотреть, как это все выглядит (рис. 18.1)?

Если вы получаете сообщение об ошибке `SMTPSenderRefused`, то не забудьте установить переменную окружения `EMAIL_PASSWORD` в оболочке, в которой выполняется `runserver`.

Собственно, и все! Некоторое время мне пришлось довольно усердно побороться, включая тыкание по пользовательскому интерфейсу Gmail, для обеспечения безопасности учетных записей, спотыкаясь о несколько пропущенных атрибутов на моей индивидуализированной модели пользователя (потому что я невнимательно прочел документацию), и даже один раз перейти на версию Django для сервера разработки, чтобы преодолеть дефект, который, к счастью, оказался несущественным.

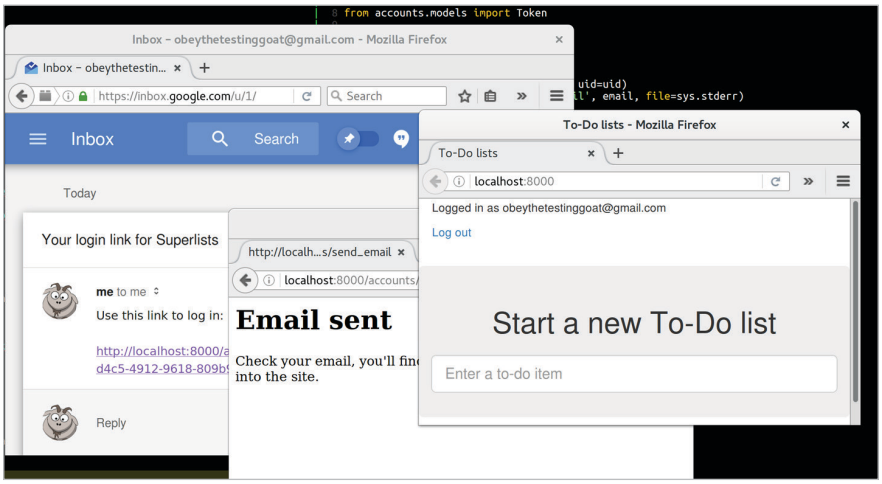


Рис. 18.1. Работает! Работает! Вухахахахаха!

Ремарка: вывод результатов в stderr

При проведении импульсного исследования критически важно иметь возможность видеть исключения, которые генерируются программным кодом. Досадно, но Django по умолчанию не отправляет все исключения в терминал, однако вы можете это сделать при помощи переменной `LOGGING` в *settings.py*:

superlists/settings.py (ch16l017)

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
        },
    },
    'root': {'level': 'INFO'},
}
```

Django использует скорее корпоративный пакет регистрации результатов выполнения из стандартной библиотеки Python, который, хотя и весьма полнофункционален, страдает от довольно крутой кривой обучения. Более подробно этот вопрос освещен в главе 21 и в документации Django (https://docs.djangoproject.com/en/1.11/topics/logging/).

Но теперь у нас есть рабочее решение! Давайте его зафиксируем на ветке для импульсного исследования:

```
$ git status
$ git add accounts
$ git commit -am "Импульсный код в индивидуализированном беспарольном
серверном процессоре аутентификации"
```

Пора внедрять результаты импульсного исследования!

Внедрение результатов импульсного исследования

Это значит, что вам нужно переписать исходный код прототипа, используя методологию TDD. Теперь у нас достаточно информации, чтобы сделать это, как надо. Так, каков наш первый шаг? Конечно же ФТ!

Пока мы останемся на ветке импульсного исследования, чтобы убедиться, что ФТ проходит относительно импульсного программного кода. Затем вернемся в ветку master и зафиксируем только ФТ.

Вот первая, простая версия ФТ.

functional_tests/test_login.py

```
from django.core import mail
from selenium.webdriver.common.keys import Keys
import re

from .base import FunctionalTest

TEST_EMAIL = 'edith@example.com'
SUBJECT = 'Your login link for Superlists'

class LoginTest(FunctionalTest):
    '''тест регистрации в системе'''

    def test_can_get_email_link_to_log_in(self):
        '''тест: можно получить ссылку по почте для регистрации'''
        # Эдит заходит на офигительный сайт суперсписков и впервые
        # замечает раздел "войти" в навигационной панели
        # Он говорит ей ввести свой адрес электронной почты, что она и делает
        self.browser.get(self.live_server_url)
        self.browser.find_element_by_name('email').send_keys(TEST_EMAIL)
        self.browser.find_element_by_name('email').send_keys(Keys.ENTER)

        # Появляется сообщение, которое говорит, что ей на почту
        # было выслано электронное письмо
```

```
        self.wait_for(lambda: self.assertIn(
            'Check your email',
            self.browser.find_element_by_tag_name('body').text
        ))

        # Эдит проверяет свою почту и находит сообщение
        email = mail.outbox[0]  ❶
        self.assertIn(TEST_EMAIL, email.to)
        self.assertEqual(email.subject, SUBJECT)

        # Оно содержит ссылку на url-адрес
        self.assertIn('Use this link to log in', email.body)
        url_search = re.search(r'http://.+/.+$', email.body)
        if not url_search:
            self.fail(f'Could not find url in email body:\n{email.body}')
        url = url_search.group(0)
        self.assertIn(self.live_server_url, url)

        # Эдит нажимает на ссылку
        self.browser.get(url)

        # Она зарегистрирована в системе!
        self.wait_for(
            lambda: self.browser.find_element_by_link_text('Log out')
        )
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertIn(TEST_EMAIL, navbar.text)
```

❶ Вы переживали, как мы собираемся решать задачу получения электронных писем в наших тестах? К счастью, пока мы можем это сделать обманным путем! При выполнении тестов Django предоставляет нам доступ к любым электронным письмам, которые сервер пытается отправить через атрибут `mail.outbox`. Мы оставим проверку реальных электронных писем на потом. (Но мы обязательно это сделаем!)

И если мы выполним ФТ, то он работает!

```
$ python manage.py test functional_tests.test_login
[...]
Not Found: /favicon.ico
saving uid [...]
login view
uid [...]
got token
```

```
new user

.
 ----------------------------------------------------------------------
Ran 1 test in 3.729s

OK
```

Вы можете увидеть немного отладочной информации, которую я оставил в своих реализациях импульсных представлений. Теперь пора обратить все наши временные изменения и внести их заново, одно за другим, на основе управляемого тестами метода.

Возвращение импульсного исходного кода в прежний вид

```
$ git checkout master # перейти назад на ветку master
$ rm -rf accounts # удалить любой след от импульсного кода
$ git add functional_tests/test_login.py
$ git commit -m "ФТ для регистрации по электронной почте"
```

Теперь мы выполним ФТ повторно и дадим ему управлять разработкой:

```
$ python manage.py test functional_tests.test_login
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [name="email"]
[...]
```

Первое, что он хочет, чтобы мы сделали, – добавить элемент input для адреса электронной почты. Bootstrap располагает несколькими встроенными классами для навигационных панелей. Воспользуемся ими и включим форму для регистрационной электронной почты:

lists/templates/base.html (ch16l020)

```
<div class="container">

  <nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
      <a class="navbar-brand" href="/">Superlists</a>
      <form class="navbar-form navbar-right" method="POST" action="#">
        <span>Enter email to log in:</span>
        <input class="form-control" name="email" type="text" />
        {% csrf_token %}
      </form>
    </div>
  </nav>

  <div class="row">
  [...]
```

ФТ не срабатывает, потому что форма действительно ничего не делает:

```
$ python manage.py test functional_tests.test_login
[...]
AssertionError: 'Check your email' not found in 'Superlists\nEnter email
to log in:\nStart a new To-Do list'
```

В этом месте я рекомендую заново внести настройки LOGGING, показанные ранее. Их тестирования в явном виде не требуется; наш текущий комплект тестов даст нам знать в маловероятном случае, если они что-либо повредят. Как мы выясним в главе 21, это будет полезно позже в целях отладки.

Самое время написать некоторый код Django. Начинаем с создания приложения под названием `accounts`, которое будет содержать все файлы, связанные с регистрацией в системе.

```
$ python manage.py startapp accounts
```

Вы даже можете выполнить фиксацию только для того, чтобы отличать прикладные файлы-заготовки от наших модификаций.

Далее давайте перестроим нашу минимальную модель пользователя, но теперь с тестами, и убедимся, что она получается круче, чем в импульсном исследовании.

Минимальная индивидуализированная модель пользователя

Встроенная в Django модель пользователя принимает разнообразные допущения о том, какую информацию о пользователях вы хотите отслеживать, начиная с имени и фамилии[9] в явной форме, заканчивая принуждением вас применять имя пользователя. Я принципиально не хочу хранить информацию о пользователе, только если в этом не будет крайней необходимости, поэтому модель пользователя, которая записывает адрес электронной почты и ничего другого, для меня звучит неплохо!

К настоящему времени, уверен, вы сможете справиться с созданием папки с тестами *tests* и ее файла `__init__.py`, удалением *tests.py* и последующим добавлением *test_models.py*, который будет содержать следующее:

[9] Решение, о котором видные специалисты по обслуживанию Django теперь, по их словам, сожалеют. Ведь не все имеют имя и фамилию.

accounts/tests/test_models.py (ch16l024)

```
from django.test import TestCase
from django.contrib.auth import get_user_model

User = get_user_model()

class UserModelTest(TestCase):
    '''тест модели пользователя'''

    def test_user_is_valid_with_email_only(self):
        '''тест: пользователь допустим только с электронной почтой'''
        user = User(email='a@b.com')
        user.full_clean() # не должно поднять исключение
```

Это дает нам ожидаемую неполадку:

```
django.core.exceptions.ValidationError: {'password': ['This field cannot be
blank.'], 'username': ['This field cannot be blank.']}
```

Пароль? Имя пользователя? Вот еще! А как насчет этого?

accounts/models.py

```
from django.db import models

class User(models.Model):
    '''пользователь'''
    email = models.EmailField()
```

И мы подключаем его внутрь *settings.py*, добавляя `accounts` в `INSTALLED_APPS` и переменную с именем `AUTH_USER_MODEL`:

superlists/settings.py (ch16l026)

```
INSTALLED_APPS = [
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
    'accounts',
]

AUTH_USER_MODEL = 'accounts.User'
```

Следующей неполадкой является ошибка базы данных:

```
django.db.utils.OperationalError: нет такой таблицы: accounts_user
```

Она, как обычно, подсказываем нам сделать миграцию... Когда мы пробуем это делать, Django жалуется, что в нашей индивидуализированной модели пользователя не хватает нескольких фрагментов метаданных:

```
$ python manage.py makemigrations
Traceback (most recent call last):
[...]
    if not isinstance(cls.REQUIRED_FIELDS, (list, tuple)):
AttributeError: type object 'User' has no attribute 'REQUIRED_FIELDS'
```

Набираем воздуха. Ну, давай же, Django, в ней всего одно поле, ты должен сам придумать ответы на эти вопросы. Вот, держи:

accounts/models.py

```
class User(models.Model):
    '''пользователь'''
    email = models.EmailField()
    REQUIRED_FIELDS = []
```

Еще один глупый вопрос?[10]

```
$ python manage.py makemigrations
[...]
AttributeError: type object 'User' has no attribute 'USERNAME_FIELD'
```

И мы проходим еще несколько шагов, пока не добираемся до:

accounts/models.py

```
class User(models.Model):
    '''пользователь'''
    email = models.EmailField()

    REQUIRED_FIELDS = []
    USERNAME_FIELD = 'email'
    is_anonymous = False
    is_authenticated = True
```

И теперь мы получаем немного другую ошибку:

[10] Вы можете спросить: если я считаю, что Django настолько глуп, почему я не отправлю запрос на включение сделанных изменений, чтобы это исправить? Это довольно простое исправление. Обещаю сделать это, как только закончу писать книгу. А пока достаточно разместить язвительные комментарии.

```
$ python manage.py makemigrations
SystemCheckError: System check identified some issues:

ERRORS:
accounts.User: (auth.E003) 'User.email' должно быть уникальным, потому что
ему назначено имя 'USERNAME_FIELD'.
```

Ну, самый простой способ ее исправить выглядит так:

accounts/models.py (ch16l028-1)

```
email = models.EmailField(unique=True)
```

Теперь миграция проходит успешно:

```
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0001_initial.py
    - Create model User
```

И тест тоже:

```
$ python manage.py test accounts
[...]
Ran 1 tests in 0.001s

OK
```

Но наша модель не такая простая, как могла бы быть. Она имеет поле для электронной почты, а также автоматически сгенерированное поле ID в качестве своего первичного ключа. Мы можем сделать его еще проще!

Тесты в качестве документирования

Давайте пойдем дальше и превратим почтовое поле в первичный ключ[11] и тем самым неявным образом удалим столбец с автоматически сгенерированным `id`.

Хотя мы можем просто взять и сделать это, при этом наш тест по-прежнему пройдет успешно, и заявить, что это была просто рефакторизация, все же лучше, если у нас будет отдельный тест:

accounts/tests/test_models.py (ch16l028-3)

```
def test_email_is_primary_key(self):
```

[11] В реальной ситуации адреса электронной почты могут оказаться не идеальным первичным ключом. Один читатель, очевидно, с глубокой эмоциональной травмой, написал мне полное слез сообщение о том, как они более десяти лет мучились, пытаясь справиться с эффектами, вызванными применением адресов электронной почты в качестве первичного ключа, которые привели к тому, что многопользовательское управление учетными записями стало невозможным. Так что, как всегда, YMMV, то есть у вас может быть иной подход.

```
    '''тест: адрес электронной почты является первичным ключом'''
    user = User(email='a@b.com')
    self.assertEqual(user.pk, 'a@b.com')
```

Он поможет в будущем нам напомнить, если мы когда-нибудь вернемся и снова посмотрим на исходный код.

```
    self.assertEqual(user.pk, 'a@b.com')
AssertionError: None != 'a@b.com'
```

Тесты могут стать формой документирования кода: они выражают ваши технические требования к отдельному классу или функции. Иногда, если вы забудете, почему сделали что-то именно так, возвращение назад и взгляд на тесты позволит вам получить ответ. Вот почему важно давать тестам явно прописанные, многословные имена методов.

А вот и реализация (можете свободно сначала проверить, что происходит с `unique=True`):

accounts/models.py (ch16l028-4)

```
email = models.EmailField(primary_key=True)
```

И мы не должны забывать корректировать миграции:

```
$ rm accounts/migrations/0001_initial.py
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0001_initial.py
    - Create model User
```

И оба наших теста проходят:

```
$ python manage.py test accounts
[...]
Ran 2 tests in 0.001s

OK
```

Модель маркера для соединения электронных адресов с уникальным идентификатором

Далее давайте создадим модель маркера. Вот короткий модульный тест, который выхватывает суть: нужно соединить электронную почту с уни-

кальным идентификатором, и этот ID не должен быть одинаковым два раза подряд:

accounts/tests/test_models.py (ch16l030)

```
from accounts.models import Token
[...]

class TokenModelTest(TestCase):
    '''тест модели маркера'''

    def test_links_user_with_auto_generated_uid(self):
        '''тест: соединяет пользователя с автогенерированным uid'''
        token1 = Token.objects.create(email='a@b.com')
        token2 = Token.objects.create(email='a@b.com')
        self.assertNotEqual(token1.uid, token2.uid)
```

Управление моделями Django на основе элементарных методов TDD сопряжено с перепрыгиванием через несколько обручей из-за миграции, поэтому мы увидим несколько таких итераций, как эта: минимальное изменение кода, выполнение миграции, получение новой ошибки, удаление миграций, воссоздание новых миграций, другое изменение кода и т. д.

```
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0002_token.py
    - Create model Token
$ python manage.py test accounts
[...]
TypeError: 'email' is an invalid keyword argument for this function
```

Доверяю вам в том, что вы добросовестно пройдете эти шаги. Помните, я не в состоянии следить за вами, а вот Билли-тестировщик может!

```
$ rm accounts/migrations/0002_token.py
$ python manage.py makemigrations
Migrations for 'accounts':
  accounts/migrations/0002_token.py
    - Create model Token
$ python manage.py test accounts
AttributeError: 'Token' object has no attribute 'uid'
```

В конечном счете вы дойдете до этого фрагмента кода…

accounts/models.py (ch16l033)

```
class Token(models.Model):
    '''маркер'''
```

```
    email = models.EmailField()
    uid = models.CharField(max_length=40)
```

И этой ошибки:

```
$ python manage.py test accounts
[...]

    self.assertNotEqual(token1.uid, token2.uid)
AssertionError: '' == ''
```

А здесь нам нужно решить, каким образом генерировать поле с произвольным уникальным идентификатором. Мы можем воспользоваться модулем Python `random`, однако Python на самом деле поставляется с еще одним, специально предназначенным для генерации уникальных идентификаторов модулем, который называется `uuid` (сокращение от англ. *universally unique id*, универсальный уникальный идентификатор).

Мы можем использовать его следующим образом:

accounts/models.py (ch16l035)

```
import uuid
[...]

class Token(models.Model):
    '''маркер'''
    email = models.EmailField()
    uid = models.CharField(default=uuid.uuid4, max_length=40)
```

После нескольких препирательств с миграциями это должно привести нас к тестам, котрые проходят успешно:

```
$ python manage.py test accounts
[...]
Ran 3 tests in 0.015s

OK
```

И это ставит нас на правильный путь! По крайней мере выполнен уровень моделей. В следующей главе мы займемся объектами-имитациями, ключевой техникой тестирования внешних зависимостей, таких как электронная почта.

Исследовательское программирование, импульсное исследование и внедрение его результатов

Импульсное исследование (spiking)

Разведочное программирование применяется с целью узнать, как работает новый API либо исследовать выполнимость нового решения. Импульсное исследование может проводиться без тестов. При этом неплохо писать свой импульсный код, используя новую ветку, и вернуться к главной ветке master при внедрении результатов импульсного исследования.

Внедрение результатов импульсного исследования (de-spiking)

Процесс переноса импульсного кода в производственную кодовую базу. Суть в том, чтобы выбросить весь старый импульсный код и начать заново с нуля, но теперь уже на основе методологии TDD. Не импульсный код нередко внешне может сильно отличаться от исходного импульсного кода – как правило, он намного аккуратнее и круче.

Написание вашего ФТ относительно импульсного кода

Хорошая это идея или нет – зависит от конкретных обстоятельств. Написание ФТ относительно импульсного кода может быть полезным хотя бы потому, что это поможет вам написать ФТ правильно. Задача выяснить, как тестировать ваш импульсный код, может быть столь же сложной, как и сам импульсный код. С другой стороны, это может вынудить вас реализовать повторное решение, которое будет очень похожим на ваш импульсный код (то, чего стоит опасаться).

Глава 19

Использование имитаций для тестирования внешних зависимостей или сокращения дублирования

В этой главе мы начнем тестировать части нашего кода, которые посылают электронные письма. В ФТ вы видели, что платформа Django позволяет получать любые электронные письма, которые она посылает, путем использования атрибута `mail.outbox`. Но в этой главе я хочу продемонстрировать очень важную технику тестирования под названием *имитация*, поэтому сейчас мы притворимся, что той прекрасной краткой формы в Django не существует.

Разве я сказал вам не использовать Django `mail.outbox`? Да нет же, это прекрасная краткая форма. Но также я хочу научить вас использовать объекты-имитации, потому что это полезный универсальный инструмент для модульного тестирования внешних зависимостей. Вы же не будете все время использовать только Django! И даже если примените эту инфраструктуру, вы можете не посылать электронные письма – любое взаимодействие со сторонним API является хорошим кандидатом на тестирование при помощи объектов-имитаций.

Перед началом: получение базовой инфраструктуры

Давайте сначала просто получим базовое представление и структуру URL-адреса. Для этого можно просто протестировать, что наш новый

URL-адрес, который будет отправляться по электронной почте для входа в систему, в конечном счете перенаправит назад на домашнюю страницу:

accounts/tests/test_views.py

```
from django.test import TestCase

class SendLoginEmailViewTest(TestCase):
    '''тест представления, которое отправляет
       сообщение для входа в систему'''

    def test_redirects_to_home_page(self):
        '''тест: переадресуется на домашнюю страницу'''
        response = self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        })
        self.assertRedirects(response, '/')
```

Подключите `include` в *superlists/urls.py* плюс URL-адрес в *accounts/urls.py* и получите успешно выполненный тест при помощи чего-то вроде этого:

accounts/views.py

```
from django.core.mail import send_mail
from django.shortcuts import redirect

def send_login_email(request):
    '''отправить сообщение для входа в систему'''
    return redirect('/')
```

Пока я добавил импорт функции `send_mail` как заготовку.

```
$ python manage.py test accounts
[...]
Ran 4 tests in 0.015s

OK
```

Порядок, начальная точка готова, теперь займемся имитированием!

Создание имитаций вручную, или Обезьянья заплатка

Когда мы вызываем `send_mail` в реальной ситуации, мы ожидаем, что Django установит связь с нашим почтовым провайдером и пошлет фактическое электронное письмо через общедоступный Интернет. Это не то, что мы хотим всегда получать в наших тестах. Аналогичная проблема воз-

никает всегда, когда у вас есть программный код, который имеет внешние побочные эффекты – вызов API, отсылка твита или SMS-сообщения или чего бы то ни было. В модульных тестах мы не хотим отсылать реальные твиты или вызовы API по Интернет. Но при этом мы по-прежнему хотим иметь способ убедиться, что наш код правильный. Объекты-имитации[1] решают эту задачу.

Одно из самых больших преимуществ Python – его динамический характер, который очень упрощает создание объектов-имитаций или того, что иногда называют внесением в программный код обезьяньих заплаток, или патчей[2]. Предположим, что в качестве первого шага мы хотим получить некий программный код, который вызывает `send_mail` с правильной темой, адресом отправителя и получателя. Он будет выглядеть примерно так:

accounts/views.py

```
def send_login_email(request):
    '''отправить сообщение для входа в систему'''
    email = request.POST['email']
    # send_mail(
    #     'Your login link for Superlists',
    #     'body text tbc',
    #     'noreply@superlists',
    #     [email],
    # )
    return redirect('/')
```

Как это протестировать, не вызывая реальную функцию `send_mail`? Ответ: наш тест может попросить, чтобы Python подменил функцию `send_mail` поддельной (имитационной) версией во время выполнения, прежде чем мы вызовем представление `send_login_email`. Вот как это выглядит:

accounts/tests/test_views.py (ch17l005)

```
from django.test import TestCase
import accounts.views  ❷

class SendLoginEmailViewTest(TestCase):
    '''тест представления, которое отправляет
```

[1] Я использую обобщенный термин «объект-имитация», но энтузиастам тестирования нравится различать другие типы общих категорий инструментов тестирования под названием «имитированные реализации», в том числе шпионы, фейки и заглушки. Разница между ними для данной книги не имеет значения, но если вы хотите дойти до конкретики, обратитесь к этому потрясающему wiki-справочнику Джастина Серлза (Justin Searls) (https://github.com/testdouble/contributing-tests/wiki/Test-Double). Предупреждение: абсолютно шокирующая подборка великолепного материала для тестирования.

[2] См. https://en.wikipedia.org/wiki/Monkey_patch

```
    сообщение для входа в систему'''
[...]

def test_sends_mail_to_address_from_post(self):
    '''тест: отправляется сообщение на адрес из метода post'''
    self.send_mail_called = False

    def fake_send_mail(subject, body, from_email, to_list):  ❶
        '''поддельная функция send_mail'''
        self.send_mail_called = True
        self.subject = subject
        self.body = body
        self.from_email = from_email
        self.to_list = to_list

    accounts.views.send_mail = fake_send_mail  ❷

    self.client.post('/accounts/send_login_email', data={
        'email': 'edith@example.com'
    })

    self.assertTrue(self.send_mail_called)
    self.assertEqual(self.subject, 'Your login link for Superlists')
    self.assertEqual(self.from_email, 'noreply@superlists')
    self.assertEqual(self.to_list, ['edith@example.com'])
```

❶ Мы определяем функцию `fake_send_mail`, которая похожа на реальную функцию `send_mail`, но она всего лишь сохраняет некоторую информацию о том, как ее вызвали, с использованием нескольких переменных, заданных на свойстве `self`.

❷ Перед тем как мы исполним тестируемый код путем вызова `self.client.post`, мы подменяем реальную функцию `accounts.views.send_mail` поддельной версией – это сводится к простой операции присваивания ей значения.

Важно понять, что здесь не происходит никакого волшебства – мы просто используем в своих интересах динамический характер Python и правила определения области видимости.

Вплоть до фактического вызыва функции можно изменять переменные, к которым она имеет доступ, при условии, что мы обращаемся к правильному пространству имен (вот почему мы импортируем верхне-уровневый модуль accounts, чтобы быть в состоянии спуститься к модулю `accounts.views`, являющемуся областью видимости, в котором будет выполняться функция `accounts.views.send_login_email`).

Это работает не только внутри модульных тестов. Реализовывать подобного рода «обезьяньи заплатки» можно в любом виде программного кода на Python!

Может понадобиться некоторое время, чтобы это усвоить. Убедите себя, что все это не является абсолютно сумасшедшей идеей, прежде чем прочтете пару фрагментов с более подробной информацией.

- Почему мы используем свойство `self` как способ передачи информации? Это просто вспомогательная переменная, которая доступна как в области видимости функции `fake_send_mail`, так и за ее пределами. Можно использовать любой мутабельный (непостоянный) объект, например список или словарь, при условии, что мы вносим локальные изменения в существующую переменную, которая существует за пределами нашей поддельной функции. (Вы можете поэкспериментировать с разными способами реализации и посмотреть, что работает, а что нет.)
- Слово «перед» имеет критически важное значение! Трудно сосчитать, сколько раз я сидел и удивлялся, почему имитация не работает, пока не вспоминал, что я не организовал имитацию *перед* тем, как вызвать тестируемый программный код.

Давайте убедимся, что наш изготовленный вручную объект-имитация позволит нам сделать тест-драйв какого-нибудь программного кода:

```
$ python manage.py test accounts
[...]
    self.assertTrue(self.send_mail_called)
AssertionError: False is not true
```

Поэтому вызовем `send_mail` наивным образом:

accounts/views.py

```
def send_login_email(request):
    '''отправить сообщение для входа в систему'''
    send_mail()
    return redirect('/')
```

Это дает:

```
TypeError: fake_send_mail() missing 4 required positional arguments:
'subject', 'body', 'from_email', and 'to_list'
```

Похоже, обезьянья заплатка работает! Мы вызвали `send_mail`, и она ушла в функцию `fake_send_mail`, которая требует больше аргументов. Давайте попробуем вот это:

accounts/views.py

```
def send_login_email(request):
    '''отправить сообщение для входа в систему'''
    send_mail('subject', 'body', 'from_email', ['to email'])
    return redirect('/')
```

Это дает:

```
    self.assertEqual(self.subject, 'Your login link for Superlists')
AssertionError: 'subject' != 'Your login link for Superlists'
```

Работает вполне прилично. И теперь можно выполнить что-то вроде этого:

accounts/views.py

```
def send_login_email(request):
    '''отправить сообщение для входа в систему'''
    email = request.POST['email']
    send_mail(
        'Your login link for Superlists',
        'body text tbc',
        'noreply@superlists',
        [email]
    )
    return redirect('/')
```

и дойти до успешно работающих тестов!

```
$ python manage.py test accounts
Ran 5 tests in 0.016s

OK
```

Блестяще! Нам удалось написать тесты к программному коду, который в обычных условиях[3] будет выходить и пытаться посылать реальные электронные письма через Интернет, и благодаря «обезьяньей» имитации функции `send_email` мы одинаково способны писать тесты и программный код.

Библиотека Mock

Популярный программный пакет *mock* для создания объектов-имитаций был добавлен в стандартную библиотеку в качестве составной части начи-

[3] Да, я знаю, что Django уже имитирует электронные сообщения, используя `mail.outbox`, но, повторюсь, давайте притворимся, что он этого не делает. А что если вы используете Flask? Или как быть, если это вызов API, а не электронная почта?

ная с Python 3.3[4]. Он предоставляет волшебный объект под названием `Mock`. Испытайте его в оболочке Python:

```
>>> from unittest.mock import Mock
>>> m = Mock()
>>> m.any_attribute
<Mock name='mock.any_attribute' id='140716305179152'>
>>> type(m.any_attribute)
<class 'unittest.mock.Mock'>
>>> m.any_method()
<Mock name='mock.any_method()' id='140716331211856'>
>>> m.foo()
<Mock name='mock.foo()' id='140716331251600'>
>>> m.called
False
>>> m.foo.called
True
>>> m.bar.return_value = 1
>>> m.bar(42, var='thing')
1
>>> m.bar.call_args
call(42, var='thing')
```

Разве это не волшебный объект? Ведь он откликается на любой запрос в отношении атрибута или вызова метода с другими имитациями, которые можно сконфигурировать с целью возврата конкретных значений для своих вызовов, и позволяет проинспектировать то, с чем он был вызван. Полезная штука, которую можно задействовать в модульных тестах!

Использование unittest.patch

И как будто этого было недостаточно, модуль `mock` также обеспечивает вспомогательную функцию под названием `patch`, с помощью которой можно ставить обезьяньи заплатки, что раньше мы делали вручную.

Вскоре я объясню, как это все работает, но сначала посмотрим на нее в действии:

accounts/tests/test_views.py (ch17l007)

```
from django.test import TestCase
from unittest.mock import patch
[...]

    @patch('accounts.views.send_mail')
    def test_sends_mail_to_address_from_post(self, mock_send_mail):
```

[4] В Python 2 его можно установить командой `pip install mock`.

```
        '''тест: отправляется сообщение на адрес из метода post'''
        self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        })

        self.assertEqual(mock_send_mail.called, True)
        (subject, body, from_email, to_list), kwargs = mock_send_mail.
call_args
        self.assertEqual(subject, 'Your login link for Superlists')
        self.assertEqual(from_email, 'noreply@superlists')
        self.assertEqual(to_list, ['edith@example.com'])
```

Выполнив тесты повторно, вы увидите, что они по-прежнему проходят. И поскольку мы всегда с подозрением относимся к любому тесту, который по-прежнему проходит после большого изменения, давайте преднамеренно его нарушим, только чтобы убедиться:

accounts/tests/test_views.py (ch17l008)

```
self.assertEqual(to_list, ['schmedith@example.com'])
```

Добавим в представление небольшой отладочный оператор печати:

accounts/views.py (ch17l009)

```
def send_login_email(request):
    '''отправить сообщение для входа в систему'''
    email = request.POST['email']
    print(type(send_mail))
    send_mail(
        [...]
```

И выполним тесты снова:

```
$ python manage.py test accounts
[...]
<class 'function'>
<class 'unittest.mock.MagicMock'>
[...]
AssertionError: Lists differ: ['edith@example.com'] !=
['schmedith@example.com']
[...]

Ran 5 tests in 0.024s

FAILED (failures=1)
```

Разумеется, тест не проходит. Сразу перед сообщением об ошибке мы видим, что, когда мы распечатываем тип функции `send_mail`, в первом модульном тесте это нормальная функция, но во втором модульном тесте – объект-имитация.

Давайте удалим преднамеренную ошибку и займемся тем, что именно тут происходит:

accounts/tests/test_views.py (ch17l011)

```
@patch('accounts.views.send_mail')  ❶
def test_sends_mail_to_address_from_post(self, mock_send_mail):  ❷
    '''тест: отправляется сообщение на адрес из метода post'''
    self.client.post('/accounts/send_login_email', data={
        'email': 'edith@example.com'  ❸
    })

    self.assertEqual(mock_send_mail.called, True)  ❹
    (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args  ❺
    self.assertEqual(subject, 'Your login link for Superlists')
    self.assertEqual(from_email, 'noreply@superlists')
    self.assertEqual(to_list, ['edith@example.com'])
```

❶ Декоратор `patch` принимает имя объекта с точечной записью в обезьянью заплатку. Это эквивалент ручной подмены `send_mail` в `accounts.views`. Преимущество декоратора в том, что, во-первых, он автоматически подменяет цель его имитацией, а во-вторых – автоматически откладывает исходный объект в конец! (В противном случае объект остается с заплаткой для остальной части прогона теста, что может вызвать проблемы в других тестах.)

❷ Затем `patch` внедряет объект-имитацию в тест как аргумент метода тестирования. Можно выбрать любое имя, которое мы для него хотим. Я обычно использую форму записи `mock_` плюс первоначальное имя объекта.

❸ Вызываем тестируемую функцию как обычно, но ко всему, что есть внутри этого метода тестирования, применяется наша имитация, поэтому представление не вызовет реальный `send_mail`, вместо этого будет видеть `mock_send_mail`.

❹ Теперь можно сделать выводы о том, что произошло с объектом-имитацией во время теста. Мы видим, что он вызывается…

❺ …и мы также можем распаковать его различные позиционные и именованные аргументы вызова и исследовать то, с чем она была вызвана. (Чуть позже обсудим `call_args` немного подробнее.)

Ясно как божий день? Нет? Не волнуйтесь, мы сделаем еще пару тестов с имитациями и убедимся, что в них становится больше смысла по мере увеличения частоты их использования.

Продвижение ФТ чуть дальше вперед

Вернемся к ФТ и посмотрим, где он не срабатывает.

```
$ python manage.py test functional_tests.test_login
[...]
AssertionError: 'Check your email' not found in 'Superlists\nEnter email
to log in:\nStart a new To-Do list'
```

Предоставление адреса электронной почты в настоящее время не имеет никакого эффекта, потому что форма никуда не отправляет данные. Подключим ее в *base.html*[5]*:*

lists/templates/base.html (ch17l012)

```
<form class="navbar-form navbar-right"
      method="POST"
      action="{% url 'send_login_email' %}">
```

Это помогает? Нет, та же самая ошибка. Почему? Поскольку в действительности мы не отображаем сообщение об успехе, после того как посылаем пользователю электронное письмо. Давайте добавим для него тест:

Тестирование инфраструктуры сообщений Django

Мы будем применять инфраструктуру сообщений Django, которая часто используется для отображения эфемерных сообщений об «успехе» или «предупреждении», чтобы показывать результаты действия. Взгляните на документацию о сообщениях Django[6], если вы с ней еще не сталкивались.

Процесс тестирования сообщений Django немножко закручен: нам приходится передавать `follow=True` в тестовый клиент, поручая ему получать страницу после переадресации с кодом 302, и исследовать ее контекст на наличие списка сообщений (который мы должны преобразовать в список, чтобы он начал вести себя так, как надо). Вот как это выглядит:

accounts/tests/test_views.py (ch17l013)

```
def test_adds_success_message(self):
    '''тест: добавляется сообщение об успехе'''
    response = self.client.post('/accounts/send_login_email', data={
```

[5] Я разбил тег формы на три строки, чтобы он удачно вписался в страницу книги. Если вы его не встречали прежде, он может выглядеть для вас странновато, но это допустимая HTML-разметка. Хотя вы не обязаны ее использовать, если она вам не нравится :)

[6] См. https://docs.djangoproject.com/en/1.11/ref/contrib/messages/

```
            'email': 'edith@example.com'
        }, follow=True)

        message = list(response.context['messages'])[0]
        self.assertEqual(
            message.message,
            "Проверьте свою почту, мы отправили Вам ссылку, \
которую можно использовать для входа на сайт."
        )
        self.assertEqual(message.tags, "success")
```

Это дает:

```
$ python manage.py test accounts
[...]
    message = list(response.context['messages'])[0]
IndexError: list index out of range
```

И мы можем заставить тест пройти успешно при помощи:

accounts/views.py (ch17l014)

```
from django.contrib import messages
[...]

def send_login_email(request):
    '''отправить сообщение для входа в систему'''
    [...]
    messages.success(
        request,
        "Проверьте свою почту, мы отправили Вам ссылку, \
которую можно использовать для входа на сайт."
    )
    return redirect('/')
```

Имитации могут привести к сильной привязке к конкретной реализации

Эта вставка – совет по тестированию на промежуточном уровне. Если в первый раз он пройдет мимо вас, вернитесь и взгляните на него еще раз, когда закончите чтение этой главы и главы 23.

Как я уже сказал, тестирование сообщений немножко закручено; мне потребовались несколько попыток, чтобы сделать все верно. На самом деле мы отказались от тестирования таким образом и решили использовать только имитации. Давайте посмотрим, как это будет выглядеть:

accounts/tests/test_views.py (ch17l014-2)

```
    from unittest.mock import patch, call
    [...]

        @patch('accounts.views.messages')
        def test_adds_success_message_with_mocks(self, mock_messages):
            response = self.client.post('/accounts/send_login_email', data={
                'email': 'edith@example.com'
            })

            expected = "Проверьте свою почту, мы отправили Вам ссылку, \
которую можно использовать для входа на сайт."
            self.assertEqual(
                mock_messages.success.call_args,
                call(response.wsgi_request, expected),
            )
```

Мы имитируем модуль `messages` и проверяем, что функция `messages.success` вызвана с правильными аргументами: первоначальный запрос и сообщение, которое мы хотим.

Вы могли бы заставить этот тест пройти успешно, используя точно такой же исходный код, что и выше. Хотя есть одна загвоздка: инфраструктура сообщений предоставляет вам более одного способа достигнуть того же результата. Я мог бы написать исходный код так:

accounts/views.py (ch17l014-3)

```
    messages.add_message(
        request,
        messages.SUCCESS,
        "Проверьте свою почту, мы отправили Вам ссылку, \
которую можно использовать для входа на сайт."
    )
```

И первоначальный тест без имитаций по-прежнему будет проходить. Но наш тест с имитациями не срабатывает, потому что мы больше не вызываем messages.success, а вызываем messages.add_message. Несмотря на то что конечный результат тот же самый и наш «правильный» тест нарушен.

Именно это имеют в виду, когда говорят, что использование имитаций может привести к сильно связанной реализации. Обычно мы говорим, что лучше тестировать поведение, а не детали реализации. Тестируйте то, что происходит, а не то, как вы это делаете. Нередко имитации допускают слишком много ошибок на стороне «как», а не на стороне «что».

Более детальное обсуждение аргументов «за» и «против» имитаций можно найти в следующих главах.

Добавление сообщений в HTML

Что происходит в функциональном тесте дальше? В принципе, пока ничего. В сущности, нам нужно добавить в страницу сообщения. Что-то вроде этого:

lists/templates/base.html (ch17l015)

```
  [...]
  </nav>

  {% if messages %}
    <div class="row">
      <div class="col-md-8">
        {% for message in messages %}
          {% if message.level_tag == 'success' %}
            <div class="alert alert-success">{{ message }}</div>
          {% else %}
            <div class="alert alert-warning">{{ message }}</div>
          {% endif %}
        {% endfor %}
      </div>
    </div>
  {% endif %}
```

А теперь мы продвинулись немного дальше? Да!

```
$ python manage.py test accounts
[...]
Ran 6 tests in 0.023s

OK

$ python manage.py test functional_tests.test_login
[...]
AssertionError: 'Use this link to log in' not found in 'body text tbc'
```

Нам нужно заполнить основной текст электронной почты ссылкой, которую пользователь может использовать для входа на сайт.

Хотя давайте пока пойдем обманным путем, заменив значение в представлении:

accounts/views.py

```
    send_mail(
        'Your login link for Superlists',
        'Use this link to log in',
        'noreply@superlists',
```

```
        [email]
    )
```

Это продвигает ФТ чуть дальше:

```
$ python manage.py test functional_tests.test_login
[...]
AssertionError: Could not find url in email body:
Use this link to log in
```

Начало с URL-адреса для входа в систему

Мы оказываемся перед необходимостью создать нечто вроде URL! Давайте сделаем это, но, опять-таки, он будет создан, чтобы только обмануть:

accounts/tests/test_views.py (ch17l017)

```
class LoginViewTest(TestCase):
    '''тест представления входа в систему'''

    def test_redirects_to_home_page(self):
        '''тест: переадресуется на домашнюю страницу'''
        response = self.client.get('/accounts/login?token=abcd123')
        self.assertRedirects(response, '/')
```

Вообразим, что мы передаем маркер как параметр GET, после `?`. Пока делать ничего не требуется.

Уверен, вы разберетесь и найдете стереотипный код для базового URL-адреса и его представления, пройдя через ошибки, которые показаны ниже:

- Нет URL:

  ```
  AssertionError: 404 != 302 : Response didn't redirect as expected:
  Response code was 404 (expected 302)
  ```

- Нет представления:

  ```
  AttributeError: module 'accounts.views' has no attribute 'login'
  ```

- Представление нарушено:

  ```
  ValueError: The view accounts.views.login didn't return an
  HttpResponse object.
  It returned None instead.
  ```

- OK!

  ```
  $ python manage.py test accounts
  [...]
  Ran 7 tests in 0.029s

  OK
  ```

Теперь мы можем дать пользователям ссылку, по которой они будут заходить на сайт. Правда, пока мы мало чего этим добъемся, ведь у нас по-прежнему нет маркера для передачи пользователю.

Подтверждение отправки пользователю ссылки с маркером

Вернувшись в представление `send_login_email`, мы протестировали поля темы, отправителя и получателя электронного сообщения. Тело сообщения – это та часть, куда мы включим маркер или URL, который можно использовать для входа в систему. Давайте для этого детально распишем два теста:

accounts/tests/test_views.py (ch17l021)

```
from accounts.models import Token
[...]

    def test_creates_token_associated_with_email(self):
        '''тест: создается маркер, связанный с электронной почтой'''
        self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        })
        token = Token.objects.first()
        self.assertEqual(token.email, 'edith@example.com')

    @patch('accounts.views.send_mail')
    def test_sends_link_to_login_using_token_uid(self, mock_send_mail):
        '''тест: отсылается ссылка на вход в систему, используя uid маркера'''
        self.client.post('/accounts/send_login_email', data={
            'email': 'edith@example.com'
        })

        token = Token.objects.first()
        expected_url = f'http://testserver/accounts/login?token={token.uid}'
        (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args
        self.assertIn(expected_url, body)
```

Первый тест довольно прямолинеен, он проверяет, что маркер, который мы создаем в базе данных, связан с адресом электронной почты из POST-запроса.

Второй – тест с использованием имитаций. Мы снова имитируем функцию `send_mail`, используя декоратор `patch`, но на этот раз из всех аргументов вызова нас интересует аргумент `body`.

Эти тесты сейчас не сработают, потому что никакого маркера мы не создаем:

```
$ python manage.py test accounts
[...]
AttributeError: 'NoneType' object has no attribute 'email'
[...]
AttributeError: 'NoneType' object has no attribute 'uid'
```

Можно заставить первый тест пройти успешно, создав маркер:

accounts/views.py (ch17l022)

```
from accounts.models import Token
[...]

def send_login_email(request):
    '''отправить сообщение для входа в систему'''
    email = request.POST['email']
    token = Token.objects.create(email=email)
    send_mail(
        [...]
```

Теперь второй тест фактически предлагает использовать маркер в теле электронной почты:

```
[...]
AssertionError:
'http://testserver/accounts/login?token=[...]
not found in 'Use this link to log in'

FAILED (failures=1)
```

Поэтому можно вставить маркер в нашу электронную почту, как тут:

accounts/views.py (ch17l023)

```
from django.core.urlresolvers import reverse
[...]

def send_login_email(request):
    '''отправить сообщение для входа в систему'''
    email = request.POST['email']
    token = Token.objects.create(email=email)
    url = request.build_absolute_uri(  ❶
        reverse('login') + '?token=' + str(token.uid)
    )
    message_body = f'Use this link to log in:\n\n{url}'
    send_mail(
        'Your login link for Superlists',
        message_body,
```

```
        'noreply@superlists',
        [email]
    )
    [...]
```

❶ `request.build_absolute_uri` заслуживает упоминания – это один из приемов, который применяется в Django для создания «полного» URL-адреса, включая доменное имя и часть с http(s). Есть и другие приемы, но они обычно сопряжены с обращением к инфраструктуре сайтов sites и довольно быстро становятся переусложненными. Если немного погуглить, можно найти широкое обсуждение этого вопроса.

Есть еще два фрагмента мозаики. Нам нужен серверный процессор аутентификации, задачей которого будет исследование маркеров на допустимость и затем возврат соответствующих пользователей. Также нам нужно, чтобы представление входа в систему фактически регистрировало пользователей в системе, когда они проходят аутентификацию.

Создание индивидуализированного серверного процессора аутентификации на основе результатов импульсного исследования

Следующий на очереди индивидуализированный серверный процессор аутентификации. Вот как он выглядел в импульсном исследовании:

```
class PasswordlessAuthenticationBackend(object):

    def authenticate(self, uid):
        '''аутентифицировать'''
        print('uid', uid, file=sys.stderr)
        if not Token.objects.filter(uid=uid).exists():
            print('no token found', file=sys.stderr)
            return None
        token = Token.objects.get(uid=uid)
        print('got token', file=sys.stderr)
        try:
            user = ListUser.objects.get(email=token.email)
            print('got user', file=sys.stderr)
            return user
        except ListUser.DoesNotExist:
            print('new user', file=sys.stderr)
            return ListUser.objects.create(email=token.email)

    def get_user(self, email):
```

```
        '''получить пользователя'''
        return ListUser.objects.get(email=email)
```

Расшифруем:

- Мы берем uid и подтверждаем его существование в базе данных.
- Возвращаем `None`, если он не существует.
- Если он существует, извлекаем адрес электронной почты и либо находим существующего пользователя с этим адресом, либо создаем нового.

Еще один тест для каждого случая

Эмпирическое правило для такого рода тестов заключается в следующем: любой оператор `if` означает дополнительный тест, и любой блок `try/except` тоже означает дополнительный тест. Таким образом, получается порядка трех тестов. Что, если сделать так?

accounts/tests/test_authentication.py

```
from django.test import TestCase
from django.contrib.auth import get_user_model
from accounts.authentication import PasswordlessAuthenticationBackend
from accounts.models import Token

User = get_user_model()

class AuthenticateTest(TestCase):
    '''тест аутентификации'''

    def test_returns_None_if_no_such_token(self):
        '''тест: возвращается None, если нет такого маркера'''
        result = PasswordlessAuthenticationBackend().authenticate(
            'no-such-token'
        )
        self.assertIsNone(result)

    def test_returns_new_user_with_correct_email_if_token_exists(self):
        '''тест: возвращается новый пользователь с правильной
           электронной почтой, если маркер существует'''
        email = 'edith@example.com'
       token = Token.objects.create(email=email)
       user = PasswordlessAuthenticationBackend().authenticate(token.uid)
       new_user = User.objects.get(email=email)
       self.assertEqual(user, new_user)

    def test_returns_existing_user_with_correct_email_if_token_exists(self):
```

```
        '''тест: возвращается существующий пользователь с правильной
           электронной почтой, если маркер существует'''
        email = 'edith@example.com'
        existing_user = User.objects.create(email=email)
        token = Token.objects.create(email=email)
        user = PasswordlessAuthenticationBackend().authenticate(token.uid)
        self.assertEqual(user, existing_user)
```

В *authenticate.py* у нас будет лишь небольшая заготовка:

accounts/authentication.py

```
class PasswordlessAuthenticationBackend(object):
    '''беспарольный серверный процессор аутентификации'''

    def authenticate(self, uid):
        '''аутентифицировать'''
        pass
```

Посмотрим, как наши дела?

```
$ python manage.py test accounts
.FE.........
======================================================================
ERROR: test_returns_new_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/accounts/tests/test_authentication.py", line 21, in
test_returns_new_user_with_correct_email_if_token_exists
    new_user = User.objects.get(email=email)
[...]
accounts.models.DoesNotExist: User matching query does not exist.
======================================================================
FAIL: test_returns_existing_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/accounts/tests/test_authentication.py", line 30, in
test_returns_existing_user_with_correct_email_if_token_exists
    self.assertEqual(user, existing_user)
AssertionError: None != <User: User object>

 ----------------------------------------------------------------------
Ran 12 tests in 0.038s

FAILED (failures=1, errors=1)
```

Вот первая примерка:

accounts/authentication.py (ch17l026)

```
from accounts.models import User, Token

class PasswordlessAuthenticationBackend(object):
    '''беспарольный серверный процессор аутентификации'''

    def authenticate(self, uid):
        '''аутентифицировать'''
        token = Token.objects.get(uid=uid)
        return User.objects.get(email=token.email)
```

Это приводит к одному работающему тесту, но нарушает другой:

```
$ python manage.py test accounts
ERROR: test_returns_None_if_no_such_token
(accounts.tests.test_authentication.AuthenticateTest)

accounts.models.DoesNotExist: Token matching query does not exist.

ERROR: test_returns_new_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
[...]
accounts.models.DoesNotExist: User matching query does not exist.
```

Давайте исправим каждую из этих неполадок по очереди:

accounts/authentication.py (ch17l027)

```
    def authenticate(self, uid):
        '''аутентифицировать'''
        try:
            token = Token.objects.get(uid=uid)
             return User.objects.get(email=token.email)
        except Token.DoesNotExist:
             return None
```

Эта версия сводит количество неполадок до одной:

```
ERROR: test_returns_new_user_with_correct_email_if_token_exists
(accounts.tests.test_authentication.AuthenticateTest)
[...]
accounts.models.DoesNotExist: User matching query does not exist.

FAILED (errors=1)
```

И теперь можно обработать заключительный случай вот так:

accounts/authentication.py (ch17l028)

```
    def authenticate(self, uid):
        '''аутентифицировать'''
        try:
            token = Token.objects.get(uid=uid)
            return User.objects.get(email=token.email)
        except User.DoesNotExist:
            return User.objects.create(email=token.email)
        except Token.DoesNotExist:
            return None
```

Эта версия аккуратнее и круче, чем в импульсном программном коде!

Метод get_user

Мы справились с функцией `authenticate`, которую Django будет использовать для регистрации новых пользователей в системе. Вторая часть протокола, которую нам нужно реализовать, – метод `get_user`, работа которого заключается в том, чтобы получать пользователя на основе его уникального идентификатора (адреса электронной почты) либо возвращать `None`, если этот метод не может его найти (взгляните еще раз на импульсный код выше, если вам понадобится напоминание).

Вот пара тестов для этих двух технических требований:

accounts/tests/test_authentication.py (ch17l030)

```
class GetUserTest(TestCase):
    '''тест получения пользователя'''

    def test_gets_user_by_email(self):
        '''тест: получает пользователя по адресу электронной почты'''
        User.objects.create(email='another@example.com')
        desired_user = User.objects.create(email='edith@example.com')
        found_user = PasswordlessAuthenticationBackend().get_user(
            'edith@example.com'
        )
        self.assertEqual(found_user, desired_user)

    def test_returns_None_if_no_user_with_that_email(self):
        '''тест: возвращается None, если нет пользователя с
           таким адресом электронной почты'''
        self.assertIsNone(
            PasswordlessAuthenticationBackend().get_user('edith@example.com')
        )
```

И наша первая неполадка:

```
AttributeError: 'PasswordlessAuthenticationBackend' object has no
attribute 'get_user'
```

Давайте тогда создадим заготовку:

accounts/authentication.py (ch17l031)

```
class PasswordlessAuthenticationBackend(object):
    '''беспарольный серверный процессор аутентификации'''

    def authenticate(self, uid):
        '''аутентифицировать'''
        [...]

    def get_user(self, email):
        '''получить пользователя'''
        pass
```

Теперь мы получаем:

```
    self.assertEqual(found_user, desired_user)
AssertionError: None != <User: User object>
```

И (шаг за шагом, только чтобы убедиться, что наш тест не работает именно так, как мы и полагаем):

accounts/authentication.py (ch17l033)

```
    def get_user(self, email):
        '''получить пользователя'''
        return User.objects.first()
```

Это дает нам возможность миновать первое утверждение и дойти до:

```
    self.assertEqual(found_user, desired_user)
AssertionError: <User: User object> != <User: User object>
```

Поэтому мы вызываем `get` с электронной почтой в качестве аргумента:

accounts/authentication.py (ch17l034)

```
    def get_user(self, email):
        '''получить пользователя'''
        return User.objects.get(email=email)
```

Теперь наш тест для случая `None` не срабатывает:

```
ERROR: test_returns_None_if_no_user_with_that_email
[...]
```

```
accounts.models.DoesNotExist: User matching query does not exist.
```

Он предлагает нам завершить метод вот так:

accounts/authentication.py (ch17l035)

```
    def get_user(self, email):
        '''получить пользователя'''
        try:
            return User.objects.get(email=email)
        except User.DoesNotExist:
            return None  ❶
```

❶ Здесь можно бы применить `pass`, и тогда функция по умолчанию вернет `None`. Однако, поскольку нам специально нужна функция, которая возвращает `None`, здесь применимо правило Дзэн «Явное лучше, чем неявное».

Это приводит нас к тому, что тесты проходят:

```
OK
```

И теперь у нас работающий серверный процессор аутентификации!

Использование серверного процессора аутентификации в представлении входа в систему

Заключительный шаг – применение серверного процессора в представлении входа в систему. Сначала добавляем его в *settings.py*:

superlists/settings.py (ch17l036)

```
AUTH_USER_MODEL = 'accounts.User'
AUTHENTICATION_BACKENDS = [
    'accounts.authentication.PasswordlessAuthenticationBackend',
]

[...]
```

Затем напишем немного тестов в отношении того, что должно произойти в представлении, снова оглядываясь на импульсный программный код:

accounts/views.py

```
def login(request):
    '''зарегистрировать вход в систему'''
    print('login view', file=sys.stderr)
    uid = request.GET.get('uid')
    user = auth.authenticate(uid=uid)
    if user is not None:
```

```
        auth.login(request, user)
    return redirect('/')
```

Нам нужно, чтобы представление вызывало `django.contrib.auth.authenticate`, а затем, если эта функция возвращает пользователя, мы вызываем `django.contrib.auth.login`.

Самое время обратиться к документации Django по аутентификации (см. https://docs.djangoproject.com/en/1.11/topics/auth/default/%23how-to-log-a-user-in), чтобы получить немного контекстной информации.

Еще одна причина использовать имитации: устранение повторов

До сих пор мы использовали имитации для тестирования внешних зависимостей, подобных функции Django отправки почтовых сообщений. Главная причина использовать имитацию состояла в том, чтобы изолировать нас от внешних побочных эффектов – в данном случае, чтобы предотвратить отсылку фактических электронных писем во время тестирования.

В этом разделе мы рассмотрим иной способ использования имитаций. Здесь у нас нет побочных эффектов, о которых нужно беспокоиться, но по-прежнему имеются причины, почему мы, возможно, захотим использовать имитацию.

Способ тестирования этого представления входа в систему без использования имитаций состоял бы в том, чтобы убедиться, действительно ли оно зарегистрировало вход пользователя в систему, выполнив проверку, и были ли пользователю назначены данные cookie аутентифицированного сеанса в правильных обстоятельствах.

Но наш серверный процессор аутентификации в действительности имеет несколько разных путей выполнения кода: 1) он возвращает `None` для недопустимых маркеров, 2) возвращает существующих пользователей, если они уже существуют, и 3) создает новых пользователей для допустимых маркеров, если они еще не существуют. Таким образом, чтобы полностью протестировать это представление, мне пришлось бы написать тесты для всех трех случаев.

Один из убедительных аргументов использовать имитации состоит в том, что они уменьшают дублирование программного кода от теста к тесту. Иными словами, это способ избежать комбинаторного взрыва.

Вдобавок ко всему использование функции `Django auth.authenticate` вместо вызова собственного кода напрямую имеет актуальное значение: в будущем это позволит добавлять новые серверные процессоры.

Поэтому в данном случае (в отличие от примера с сообщениями, приведенного ранее во вставке) реализация действительно имеет значение, а использование имитации спасет нас от дублирования программного кода в тестах. Вот как это выглядит:

accounts/tests/test_views.py (ch17l037)

```
from unittest.mock import patch, call
[...]

    @patch('accounts.views.auth')  ❶
    def test_calls_authenticate_with_uid_from_get_request(self, mock_auth):  ❷
        '''тест: вызывается authenticate с uid из GET-запроса'''
        self.client.get('/accounts/login?token=abcd123')
        self.assertEqual(
            mock_auth.authenticate.call_args,  ❸
            call(uid='abcd123')  ❹
        )
```

❶ Мы ожидаем, что модуль `django.contrib.auth` будет использован в *views.py*, и мы его там имитируем. Обратите внимание: на этот раз мы имитируем не функцию, а целый модуль, а следовательно, неявным образом имитируем все функции (и любые другие объекты), который этот модуль содержит.

❷ Как обычно, имитируемый объект внедряется в наш метод тестирования.

❸ На этот раз мы сымитировали модуль, а не функцию. Поэтому исследуем `call_args` не из модуля `mock_auth`, а из функции `mock_auth.authenticate`. Поскольку все атрибуты имитации – это дополнительные имитации, то это тоже является имитацией. Здесь можно увидеть, почему mock-объекты так удобны по сравнению с попытками создать свои собственные.

❹ Теперь вместо «распаковки» аргументов вызова мы используем функцию `call` как более аккуратный способ сообщить, с какими аргументами ее нужно было вызвать, т. е., с маркером из GET-запроса (см. вставку ниже).

По поводу имитации call_args

Свойство `call_args` на mock-объекте представляет собой позиционные и именованные аргументы, с которыми mock-объект был вызван. Это специальный тип объекта «вызова», который по сути является кортежем (`positional_args`, `keyword_args`). Элемент `positional_args` как таковой и сам представляет собой кортеж, состоящий из набора позиционных аргументов. Элемент `keyword_args` является словарем.

```
    >>> from unittest.mock import Mock, call
    >>> m = Mock()
    >>> m(42, 43, 'positional arg 3', key='val', other_kwarg=666)
    <Mock name='mock()' id='139909729163528'>

    >>> m.call_args
    call(42, 43, 'positional arg 3', key='val', other_kwarg=666)

    >>> m.call_args == ((42, 43, 'positional arg 3'), {'key': 'val',
'other_kwarg': 666})
    True
    >>> m.call_args == call(42, 43, 'positional arg 3', key='val',
other_kwarg=666)
    True
```

Поэтому в нашем тесте мы могли бы сделать следующее:

```
    self.assertEqual(
        mock_auth.authenticate.call_args,
        ((,), {'uid': 'abcd123'})
    )
    # либо следующее
    args, kwargs = mock_auth.authenticate.call_args
    self.assertEqual(args, (,))
    self.assertEqual(kwargs, {'uid': 'abcd123')
```

Но вы сами видите: использование вспомогательной функции `call` гораздо круче.

Что происходит, когда мы выполняем тест? Первая ошибка будет такой:

```
$ python manage.py test accounts
[...]
AttributeError: <module 'accounts.views' from
'/.../superlists/accounts/views.py'> does not have the attribute 'auth'
```

Модуль foo не имеет атрибута bar – это типичная ошибка теста, в котором используются имитации. Она говорит о том, что вы пытаетесь сымитировать что-то, что еще не существует (или еще не импортировано) в целевом модуле.

Если мы импортируем `django.contrib.auth`, ошибка изменяется:

accounts/views.py (ch17l038)

```
from django.contrib import auth, messages
[...]
```

Теперь мы получаем:

```
AssertionError: None != call(uid='abcd123')
```

Она сообщает, что представление вообще не вызывает функцию `auth.authenticate`. Давайте это исправим, но сделаем это преднамеренно неправильно, только чтобы посмотреть:

accounts/views.py (ch17l039)

```
def login(request):
    '''зарегистрировать вход в систему'''
    auth.authenticate('Ба-бах!')
    return redirect('/')
```

Действительно, ба-бах!

```
$ python manage.py test accounts
[...]
AssertionError: call('Ба-бах!') != call(uid='abcd123')
[...]
FAILED (failures=1)
```

Передадим функции `authenticate` аргументы, которые она ожидает:

accounts/views.py (ch17l040)

```
def login(request):
    '''зарегистрировать вход в систему'''
    auth.authenticate(uid=request.GET.get('token'))
    return redirect('/')
```

Это снова приводит нас к тому, что тесты проходят:

```
$ python manage.py test accounts
[...]
Ran 15 tests in 0.041s

OK
```

Использование mock.return_value

Далее нам нужно проверить, что, если функция `authenticate` возвращает пользователя, то мы передаем это значение в `auth.login`. Давайте посмотрим, как выглядит этот тест:

accounts/tests/test_views.py (ch17l041)

```
@patch('accounts.views.auth')  ❶
def test_calls_auth_login_with_user_if_there_is_one(self, mock_auth):
    '''тест: вызывается auth_login с пользователем, если такой имеется'''
    response = self.client.get('/accounts/login?token=abcd123')
    self.assertEqual(
        mock_auth.login.call_args,  ❷
        call(response.wsgi_request, mock_auth.authenticate.return_value)  ❸
    )
```

❶ Мы снова имитируем модуль `contrib.auth`.

❷ На этот раз исследуем аргументы вызова для функции `auth.login`.

❸ Проверяем, что она вызывается с объектом запроса, который видим в представлении, и с объектом «пользователь», который возвращается функцией `authenticate`. Поскольку функция `authenticate` также имитируется, мы можем использовать ее специальный атрибут `return_value`.

Вызвав имитацию, вы получаете еще одну имитацию. Но вы также можете получить копию этой возвращенной имитации из исходной имитации, которую вы вызвали. Н-да, трудновато объяснить это, не повторяя слово «имитация» много раз! Возможно, тут пригодится еще одна небольшая консольная иллюстрация:

```
>>> m = Mock()
>>> thing = m()
>>> thing
<Mock name='mock()' id='140652722034952'>
>>> m.return_value
<Mock name='mock()' id='140652722034952'>
>>> thing == m.return_value
True
```

В любом случае, что же мы получаем, если выполним тест?

```
$ python manage.py test accounts
[...]
    call(response.wsgi_request, mock_auth.authenticate.return_value)
AssertionError: None != call(<WSGIRequest: GET '/accounts/login?t[...]
```

Этот результат говорит, что мы пока вообще не вызываем `auth.login`. Давайте попробуем это сделать. Как всегда, сначала преднамеренно неправильно!

accounts/views.py (ch17l042)

```
def login(request):
    '''зарегистрировать вход в систему'''
    auth.authenticate(uid=request.GET.get('token'))
    auth.login('ack!')
    return redirect('/')
```

Ты смотри, действительно!

```
TypeError: login() missing 1 required positional argument: 'user'
[...]
AssertionError: call('ack!') != call(<WSGIRequest: GET
'/accounts/login?token=[...]
```

Давайте это исправим:

accounts/views.py (ch17l043)

```
def login(request):
    '''зарегистрировать вход в систему'''
    user = auth.authenticate(uid=request.GET.get('token'))
    auth.login(request, user)
    return redirect('/')
```

Получаем неожиданную жалобу:

```
ERROR: test_redirects_to_home_page (accounts.tests.test_views.LoginViewTest)
[...]
AttributeError: 'AnonymousUser' object has no attribute '_meta'
```

Все потому, что мы по-прежнему вызываем `auth.login` без разбора для любого типа пользователя, это вызывает проблемы в нашем исходном тесте на переадресацию, который в настоящее время не имитирует `auth.login`. Мы должны добавить оператор `if` (и, следовательно, еще один тест) и, раз мы тут, узнаем об установке заплаток на уровне класса.

Установка заплатки на уровне класса

Мы хотим добавить еще один тест с еще одним `@patch('accounts.views.auth')`, и это уже начинает повторяться. Применим правило трех поклевок и переместим декоратор заплатки на уровень класса. Это даст эффект имитации `accounts.views.auth` в каждом отдельном методе тестирования в этом классе. Это также означает, что в наш исходный тест переадресации теперь также будет внедрена переменная `mock_auth`:

accounts/tests/test_views.py (ch17l044)

```
@patch('accounts.views.auth')  ❶
```

```
class LoginViewTest(TestCase):

    def test_redirects_to_home_page(self, mock_auth):  ❷
        '''тест: переадресуется на домашнюю страницу'''
        [...]

    def test_calls_authenticate_with_uid_from_get_request(self, mock_auth):  ❸
        '''тест: вызывается authenticate с uid из GET-запроса'''
        [...]

    def test_calls_auth_login_with_user_if_there_is_one(self, mock_auth):  ❸
        '''тест: вызывается auth_login с пользователем, если такой имеется'''
        [...]

    def test_does_not_login_if_user_is_not_authenticated(self, mock_auth):
        '''тест: не регистрируется в системе, если пользователь
           Не аутентифицирован'''
        mock_auth.authenticate.return_value = None  ❹
        self.client.get('/accounts/login?token=abcd123')
        self.assertEqual(mock_auth.login.called, False)  ❺
```

❶ Перемещаем заплатку на уровень класса...

❷ А значит, в первый метод тестирования внедрен дополнительный аргумент.

❸ Можно удалить декораторы из всех остальных тестов.

❹ В новом тесте мы явным образом устанавливаем `return_value` на имитации `auth.authenticate` *перед* вызовом `self.client.get`.

❺ Добавляем утверждение, что, если `authenticate` не возвращает `None`, то нам вообще не следует вызывать `auth.login`.

Это очищает побочную неполадку и дает нам конкретную ожидаемую неполадку, с которой мы продолжим работу:

```
    self.assertEqual(mock_auth.login.called, False)
AssertionError: True != False
```

Заставляем тест пройти успешно вот так:

accounts/views.py (ch17l045)

```
def login(request):
    '''зарегистрировать вход в систему'''
    user = auth.authenticate(uid=request.GET.get('token'))
    if user:
        auth.login(request, user)
```

```
    return redirect('/')
```

Итак, мы уже у цели?

Момент истины: пройдет ли ФТ?

Думаю, мы почти готовы попробовать наш функциональный тест!

Только сначала убедимся, что наш базовый шаблон показывает разную панель навигации для зарегистрированных и незарегистрированных пользователей (на который опирается наш ФТ):

lists/templates/base.html (ch17l046)

```
<nav class="navbar navbar-default" role="navigation">
  <div class="container-fluid">
    <a class="navbar-brand" href="/">Superlists</a>
    {% if user.email %}
      <ul class="nav navbar-nav navbar-right">
        <li class="navbar-text">Logged in as {{ user.email }}</li>
        <li><a href="#">Log out</a></li>
      </ul>
    {% else %}
      <form class="navbar-form navbar-right"
            method="POST"
            action="{% url 'send_login_email' %}">
        <span>Enter email to log in:</span>
        <input class="form-control" name="email" type="text" />
        {% csrf_token %}
      </form>
    {% endif %}
  </div>
</nav>
```

И посмотрим, будет ли…

```
$ python manage.py test functional_tests.test_login
Internal Server Error: /accounts/login
[...]
  File "/.../superlists/accounts/views.py", line 31, in login
    auth.login(request, user)
[...]
ValueError: The following fields do not exist in this model or are m2m fields:
last_login
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: Log out
```

Не может быть! Что-то не так. Но если допустить, что вы сохранили конфигурацию `LOGGING` в *settings.py*, вы увидите обратную трассировку с объяснениями, как выше. Она говорит что-то о поле `last_login`.

Мне кажется[7], что это дефект в самой платформе Django, однако если по существу, то инфраструктура аутентификации ожидает, что модель пользователя будет иметь поле `last_login`. У нас такого нет. Но не беспокойтесь! Есть способ справиться с этой ошибкой.

Давайте напишем модульный тест, который воспроизведет этот дефект. Поскольку он связан с нашей индивидуализированной моделью пользователя, самым подходящим для него местом будет *test_models.py*:

accounts/tests/test_models.py (ch17l047)

```
from django.test import TestCase
from django.contrib import auth
from accounts.models import Token

User = auth.get_user_model()

class UserModelTest(TestCase):
    '''тест модели пользователя'''

    def test_user_is_valid_with_email_only(self):
        '''тест: пользователь допустим только с электронной почтой'''
        [...]

    def test_email_is_primary_key(self):
        '''тест: адрес электронной почты является первичным ключом'''
        [...]

    def test_no_problem_with_auth_login(self):
        '''тест: проблем с auth_login нет'''
        user = User.objects.create(email='edith@example.com')
        user.backend = ''
        request = self.client.request().wsgi_request
        auth.login(request, user) # не должно поднять исключение
```

Создаем объект запроса и пользователя и передаем их в функцию `auth.login`.

Это вызовет ошибку:

```
    auth.login(request, user) # не должно поднять исключение
[...]
ValueError: The following fields do not exist in this model or are m2m fields:
last_login
```

[7] См. https://code.djangoproject.com/ticket/26823

Для целей настоящей книги конкретная причина этого дефекта не имеет значения, но если вам любопытно понять, что именно тут происходит, просмотрите исходные строки Django, перечисленные в обратной трассировке, и почитайте документацию Django о сигналах[8].

Общий вывод: это можно исправить вот так:

accounts/models.py (ch17l048)

```
import uuid
from django.contrib import auth
from django.db import models

auth.signals.user_logged_in.disconnect(auth.models.update_last_login)

class User(models.Model):
    '''пользователь'''
    [...]
```

И как теперь выглядит наш ФТ?

```
$ python manage.py test functional_tests.test_login
[...]
.
 ---------------------------------------------------------------------
Ran 1 test in 3.282s

OK
```

Теоретически – работает! Работает ли на практике?

Ничего себе! Вы можете в это поверить? Я – с трудом! Самое время осмотреться в ручном режиме при помощи команды `run server`:

```
$ python manage.py runserver
[...]
Internal Server Error: /accounts/send_login_email
Traceback (most recent call last):
  File "/.../superlists/accounts/views.py", line 20, in send_login_email

ConnectionRefusedError: [Errno 111] Connection refused
```

Когда вы попытаетесь выполнить это вручную, то, вероятно, получите ошибку, что и я. Две возможные проблемы:

8 См. https://docs.djangoproject.com/en/1.11/topics/signals/

- Во-первых, нам нужно повторно добавить конфигурацию электронной почты в *settings.py*.
- Во-вторых, нам нужно экспортировать (командой `export`) пароль электронной почты в нашей оболочке.

superlists/settings.py (ch17l049)

```
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'obeythetestinggoat@gmail.com'
EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_PASSWORD')
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

и

```
$ export EMAIL_PASSWORD="sekrit"
$ python manage.py runserver
```

Затем вы увидите что-то вроде того, что показано на рис. 19.1.

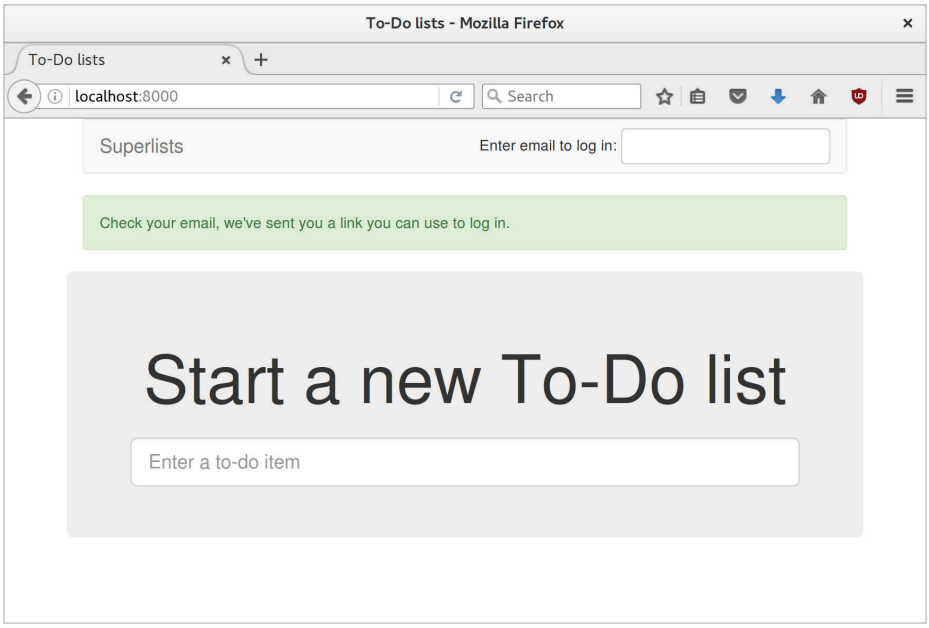


Рис. 19.1. Проверьте свою почту, мы отправили вам ссылку, которую можно использовать для входа на сайт

Класс!

Я откладывал фиксацию вплоть до этого момента, только чтобы удостовериться, что все работает. В этой точке можно сделать серию отдельных фиксаций: одну для представления входа в систему, одну для серверного процессора аутентификации, одну для модели пользователя и одну для

подключения шаблона. Или же остановиться на том, что, раз они все взаимосвязаны и ни один компонент не будет работать без других, то с таким же успехом можно сделать одну большую фиксацию:

```
$ git status
$ git add .
$ git diff --staged
$ git commit -m "Индивидуализированный беспарольный серверный процессор
аутентификации + индивидуализированная модель пользователя"
```

Завершение ФТ, тестирование выхода из системы

Последнее, что мы должны сделать, прежде чем будем закругляться, – протестировать ссылку на выход из системы. Расширим ФТ с помощью пары дополнительных шагов:

functional_tests/test_login.py (ch17l050)

```
    [...]
    # Она зарегистрировалась в системе!
    self.wait_for(
        lambda: self.browser.find_element_by_link_text('Log out')
    )
    navbar = self.browser.find_element_by_css_selector('.navbar')
    self.assertIn(TEST_EMAIL, navbar.text)

    # Теперь она выходит из системы
    self.browser.find_element_by_link_text('Log out').click()

    # Она вышла из системы
    self.wait_for(
        lambda: self.browser.find_element_by_name('email')
    )
    navbar = self.browser.find_element_by_css_selector('.navbar')
    self.assertNotIn(TEST_EMAIL, navbar.text)
```

Мы видим, что тест не проходит, потому что не работает кнопка выхода из системы:

```
$ python manage.py test functional_tests.test_login
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: [name="email"]
```

Реализация кнопки выхода из системы на самом деле очень проста: можно использовать встроенное в Django представление выхода из систе-

мы[9], которое очищает сеанс пользователя сверху-вниз и переадресует его на страницу по нашему выбору:

accounts/urls.py (ch17l051)

```
from django.contrib.auth.views import logout
[...]

urlpatterns = [
    url(r'^send_login_email$', views.send_login_email, name='send_login_email'),
    url(r'^login$', views.login, name='login'),
    url(r'^logout$', logout, {'next_page': '/'}, name='logout'),
]
```

При этом в *base.html* мы просто превращаем выход из системы в реальную URL-ссылку:

lists/templates/base.html (ch17l052)

```
<li><a href="{% url 'logout' %}">Log out</a></li>
```

Это позволяет нам добиться того, чтобы ФТ прошел полностью успешно. Теперь у нас комплект из полностью проходящих тестов:

```
$ python manage.py test functional_tests.test_login
[...]
OK
$ python manage.py test
[...]
Ran 59 tests in 78.124s

OK
```

Здесь мы даже близко не подошли к реально безопасной или приемлемой регистрации в системе. Поскольку это всего лишь демонстрационное приложение, приведенное в качестве примера для книги, оставим его в этом состоянии. В реальной ситуации необходимо исследовать возможность использования дополнительных мер безопасности и решить вопрос удобства применения, чтобы считать работу законченной. Здесь мы опасно приблизились к выпуску нашего собственного шифрокода, и опора на более устоявшуюся систему регистрации в системе была бы намного безопаснее.

В следующей главе мы начнем пробовать найти применение нашему приложению регистрации. Тем временем сделайте фиксацию и наслаждайтесь этим резюме:

[9] См. http://bit.ly/SuI0hA

Об имитации в Python

Имитация и внешние зависимости

Мы используем имитацию в модульных тестах, когда у нас есть внешняя зависимость, которую мы не хотим применять в тестах. Имитация используется для моделирования стороннего API. Хотя в Python есть возможность выпускать свои собственные имитации, инфраструктура для их создания (модуль mock) обеспечивает много полезных укороченных путей, которые упростят написание и, что еще более важно, чтение ваших тестов.

Установка обезьяньих заплаток

Подмена объекта в пространстве имен в ходе выполнения. Этот прием используется в модульных тестах для подмены реальной функции, которая имеет нежелательные побочные эффекты, фиктивным (имитирующим) объектом, используя декоратор patch.

Библиотека Mock

Майкл Фурд (Michael Foord) (который раньше работал на компанию, ставшую инициатором создания PythonAnywhere, непосредственно перед тем, как я к ней присоединился) написал превосходную библиотеку Mock, которая теперь интегрирована в стандартную библиотеку Python 3. Она содержит подавляющую часть всего того, что может понадобиться при создании имитаций на Python.

Декоратор patch

unittest.mock обеспечивает функцию под название patch, которую можно использовать для имитации любого объекта из тестируемого модуля. Обычно она используется в качестве декоратора на методе тестирования либо даже на уровне класса, где она применяется ко всем методам тестирования этого класса.

Имитации могут привести к сильной привязке к конкретной реализации

Как мы видели во вставке, посвященной сообщениям, имитации могут привести к тому, что исходный код будет сильно привязан к реализации. Поэтому не используйте их без серьезных причин.

Имитации могут спасти от дублирования программного кода в тестах

С другой стороны, нет никакого смысла в дублировании всех тестов для функции в высокоуровневой части кода, который ее использует. Использование имитации в этом случае снижает дублирование.

Более подробное обсуждение аргументов «за» и «против» имитаций совсем близко. Продолжайте читать!

Глава 20

Тестовые фикстуры и декоратор для явных ожиданий

Теперь, когда у нас есть функционирующая система аутентификации, мы хотим использовать ее для идентификации пользователей и иметь возможность показывать им все списки, которые они создали.

Для этого необходимо написать функциональные тесты, которые включают зарегистрированного пользователя. Мы не хотим заставлять каждый тест проходить через долгую свистопляску с регистрацией по электронной почте, нужна возможность пропустить эту часть.

Это связано с разделением компетенций. Функциональные тесты отличаются от модульных тем, что в них обычно нет единственного тестирующего утверждения. Однако концептуально они должны тестировать единицу программного кода. Нет никакой потребности в том, чтобы каждый ФТ тестировал механизмы входа/выхода из системы. Если мы придумаем, как «обмануть» и пропустить эту часть, мы не будем тратить время на ожидание, когда же завершатся повторяющиеся пути тестирования.

Не перегните палку при ликвидации повторов в функциональных тестах. Одно из преимуществ ФТ в том, что он может поймать странные и непредсказуемые взаимодействия между разными элементами приложения.

Эта глава была только что переписана для нового издания, поэтому сообщите мне по obeythetestinggoat@gmail.com, если обнаружите какие-либо проблемы или у вас возникли предложения по ее улучшению!

Пропуск регистрации в системе путем предварительного создания сеанса

Вполне нормально, когда пользователь возвращается на сайт и по-прежнему имеет данные cookie, то есть он «предварительно аутентифициро-

ван», и это вовсе не невыполнимая затея. Вот как это можно организовать:

functional_tests/test_my_lists.py

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY, get_user_model
from django.contrib.sessions.backends.db import SessionStore
from .base import FunctionalTest

User = get_user_model()

class MyListsTest(FunctionalTest):
    '''тест приложения “Мои списки”'''

    def create_pre_authenticated_session(self, email):
        '''создать предварительно аутентифицированный сеанс'''
        user = User.objects.create(email=email)
        session = SessionStore()
        session[SESSION_KEY] = user.pk  ❶
        session[BACKEND_SESSION_KEY] = settings.AUTHENTICATION_BACKENDS[0]
        session.save()
        ## установить cookie, которые нужны для первого посещения домена.
        ## страницы 404 загружаются быстрее всего!
        self.browser.get(self.live_server_url + "/404_no_such_url/")
        self.browser.add_cookie(dict(
            name=settings.SESSION_COOKIE_NAME,
            value=session.session_key,  ❷
            path='/',
        ))
```

❶ Мы создаем объект-сеанс в базе данных. Сеансовый ключ – это первичный ключ объекта-пользователя (который фактически представлен его адресом электронной почты).

❷ Затем мы добавляем cookie в браузер, который совпадает с сеансом на сервере: во время нашего следующего визита на сайт сервер должен распознать нас как зарегистрированного пользователя.

Обратите внимание: в том виде, как сейчас, это будет работать только потому, что мы используем класс `LiveServerTestCase`. Поэтому объекты `User` и `Session`, которые мы создаем, в конечном итоге окажутся в той же базе данных, что и сервер тестирования. Позже нам нужно будет это модифицировать и для работы относительно базы данных на промежуточном сервере.

Сеансы Django: как данные cookie сообщают серверу, что пользователь аутентифицирован

Попытка объяснить сеансы, данные cookie и аутентификацию в Django.

Поскольку HTTP не имеет внутреннего состояния, серверам нужен способ распознавать разных клиентов в каждом отдельном запросе. IP-адреса могут использоваться совместно, поэтому обычно каждому клиенту дается уникальный идентификатор сеанса, который он хранит в формате cookie и представляет с каждым запросом. Сервер сохранит этот ID в каком-то месте (по умолчанию – в базе данных), и затем он может распознавать каждый запрос, который поступает от конкретного клиента.

Если вы входите на сайт через сервер разработки, то вы фактически можете взглянуть на ID сеанса вручную, если нужно. Он по умолчанию хранится как ключ `sessionid`. Посмотрите на рис. 20.1.

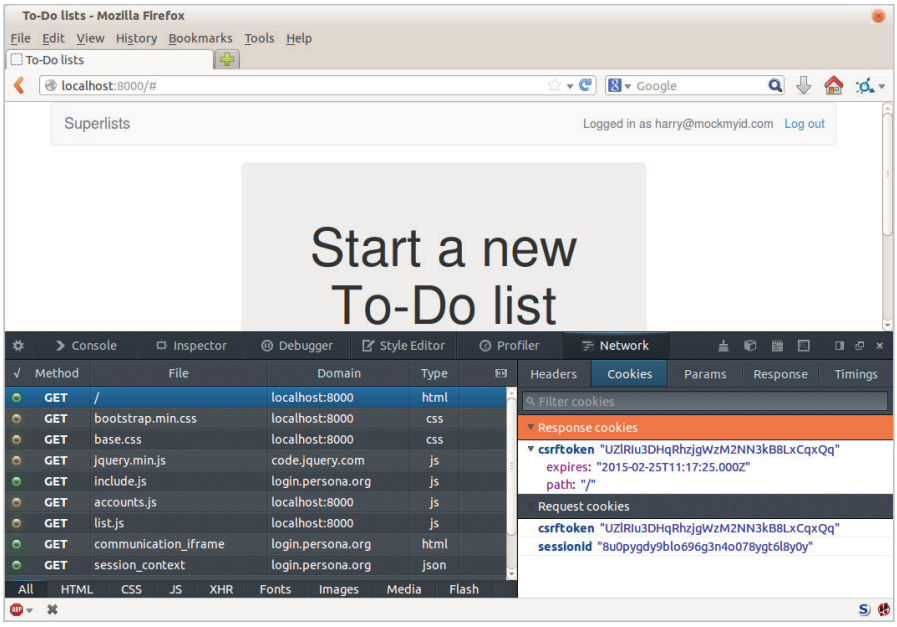


Рис. 20.1. Исследование данных cookie сеанса на панели инструментов Debug

Такие сеансовые данные cookie устанавливаются для всех посетителей сайта Django, зарегистрированных и нет.

Когда мы хотим распознать посетителя как зарегистрированного и аутентифицированного пользователя, опять-таки, вместо того чтобы с каждым запросом просить клиента отправлять свое имя пользователя и пароль, сервер фактически может просто отметить сеанс этого клиента как аутентифицированный и связать его с ID пользователя в своей базе данных.

Сеанс – это структура данных, похожая на словарь, и идентификатор пользователя хранится с ключом, заданным `django.contrib.auth.SESSION_KEY`. Если нужно, вы можете его проверить в консоли с помощью команды `manage.py shell`:

```
$ python manage.py shell
[...]
In [1]: from django.contrib.sessions.models import Session

# подставьте здесь id сеанса из cookie вашего браузера
In [2]: session = Session.objects.get(
    session_key="8u0pygdy9blo696g3n4o078ygt6l8y0y"
)

In [3]: print(session.get_decoded())
{'_auth_user_id': 'obeythetestinggoat@gmail.com', '_auth_user_backend':
'accounts.authentication.PasswordlessAuthenticationBackend'}
```

В сеансе пользователя также можно хранить любую другую информацию, которая вам нужна для временного отслеживания некоторого состояния. Это работает и для незарегистрированных пользователей. Для этого внутри любого представления используйте `request.session`. Данные хранятся в виде словаря dict. Более подробная информация – в документации Django о сеансах (см. https://docs.djangoproject.com/en/1.11/topics/http/sessions/).

Проверка работоспособности решения

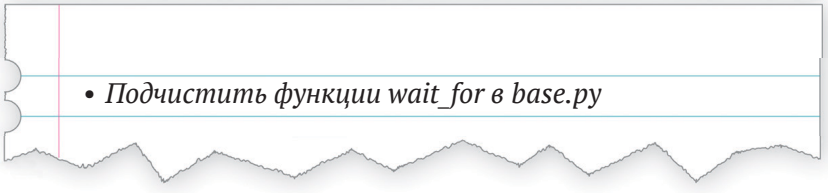
Чтобы проверить работоспособность этого решения, неплохо было бы применить часть кода из предыдущего теста. Давайте создадим пару функций под названием `wait_to_be_logged_in` и `wait_to_be_log ged_out`. Чтобы получать к ним доступ из другого теста, нужно будет подтянуть их в `FunctionalTest`. Мы слегка доработаем их, чтобы в качестве параметра они могли принимать произвольный адрес электронной почты:

functional_tests/base.py (ch18l002)

```
class FunctionalTest(StaticLiveServerTestCase):
    '''функциональный тест'''
    [...]

    def wait_to_be_logged_in(self, email):
        '''ожидать входа в систему'''
        self.wait_for(
           lambda: self.browser.find_element_by_link_text('Log out')
        )
        navbar = self.browser.find_element_by_css_selector('.navbar')
```

```
        self.assertIn(email, navbar.text)

    def wait_to_be_logged_out(self, email):
        '''ожидать выхода из системы'''
        self.wait_for(
            lambda: self.browser.find_element_by_name('email')
        )
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertNotIn(email, navbar.text)
```

Хм, неплохо, но я не вполне доволен объемом дублирования в функциях `wait_for`. Давайте сделаем пометку, чтобы вернуться к этому вопросу позже, и приведем этих помощников в рабочее состояние.



Сначала применим их в *test_login.py*:

functionaltests/testlogin.py (ch18l003)

```
    def test_can_get_email_link_to_log_in(self):
        '''тест: можно получить ссылку по почте для регистрации'''
        [...]
        # Она зарегистрировалась в системе!
        self.wait_to_be_logged_in(email=TEST_EMAIL)

        # Теперь она выходит из системы
        self.browser.find_element_by_link_text('Log out').click()

        # Она вышла из системы
        self.wait_to_be_logged_out(email=TEST_EMAIL)
```

Только чтобы проверить, что мы ничего не повредили, повторно выполняем тест входа в систему:

```
$ python manage.py test functional_tests.test_login
[...]
OK
```

Теперь мы можем написать заготовку для теста «Мои списки», чтобы убедиться, что мастер предварительно аутентифицированного сеанса работает:

functional_tests/test_my_lists.py (ch18l004)

```
    def test_logged_in_users_lists_are_saved_as_my_lists(self):
        '''тест: списки зарегистрированных пользователей
           сохраняются как «мои списки»'''
        email = 'edith@example.com'
        self.browser.get(self.live_server_url)
        self.wait_to_be_logged_out(email)

        # Эдит является зарегистрированным пользователем
        self.create_pre_authenticated_session(email)
        self.browser.get(self.live_server_url)
        self.wait_to_be_logged_in(email)
```

Это дает нам:

```
$ python manage.py test functional_tests.test_my_lists
[...]
OK
```

Самое время для фиксации:

```
$ git add functional_tests
$ git commit -m "test_my_lists: предсоздание сеансов, перемещение проверок
регистрации в базу"
```

Тестовые фикстуры JSON считаются вредными

Когда мы предварительно заполняем базу данных данными тестирования, как мы сделали здесь с объектом User и связанным с ним объектом Session, мы настраиваем тестовую фикстуру, или тестовое окружение.

Django поставляется со встроенной поддержкой сохранения объектов базы данных как JSON (используя команды manage.py dumpdata) и их автоматической загрузкой в прогоны тестов, используя атрибут fixtures класса в TestCase.

Все больше специалистов советуют не использовать фикстуры JSON. Их поддержка превращается в настоящий кошмар, когда ваша модель изменяется. Плюс читателю очень трудно определить, какие из множества значений атрибутов, определенных в JSON, имеют для тестируемого поведения критически важное значение, а какие являются просто заполнителями. Наконец, даже если тесты начнут обмениваться фикстурами, то рано или поздно одному из них потребуются немного отличающиеся версии данных и в конечном итоге вы будете копировать все вокруг, чтобы оставить их изолированными. И опять-таки очень трудно понять, что действительно относится к тесту, а что – просто случайность.

Обычно намного более однозначной будет простая загрузка данных, непосредственно через Django ORM.

Когда у вас определено больше пары полей на модели и/или несколько связанных между собой моделей, даже использование ORM может стать громоздким. Для этих случаев существует инструмент, который, по настойчивым заявлениям многих, называется `factory_boy` (https://factoryboy.readthedocs.org/).

Финальный вспомогательный метод явного ожидания: декоратор ожидания

До этого момента мы уже несколько раз применяли декораторы в исходном коде. Пора узнать, как они работают на практике. Для этого мы создадим свой собственный.

Для начала давайте представим, как бы мы хотели, чтобы наш декоратор работал. Было бы круто иметь возможность заменить всю индивидуализированную логику «ожидать/повторять/тайм-аут» в функции `wait_for_row_in_list_table` и локальную `self.wait_for` в функциях `wait_to_be_logged_in/out`. Что-то такое выглядело бы великолепно:

functional_tests/base.py (ch18l005)

```
@wait
def wait_for_row_in_list_table(self, row_text):
    '''ожидать строку в таблице списка'''
    table = self.browser.find_element_by_id('id_list_table')
    rows = table.find_elements_by_tag_name('tr')
    self.assertIn(row_text, [row.text for row in rows])

@wait
def wait_to_be_logged_in(self, email):
    '''ожидать входа в систему'''
    self.browser.find_element_by_link_text('Log out')
    navbar = self.browser.find_element_by_css_selector('.navbar')
    self.assertIn(email, navbar.text)

@wait
def wait_to_be_logged_out(self, email):
    '''ожидать выхода из системы'''
```

```
        self.browser.find_element_by_name('email')
        navbar = self.browser.find_element_by_css_selector('.navbar')
        self.assertNotIn(email, navbar.text)
```

Вы готовы за это взяться? Хотя декораторы с трудом укладываются в голове (уж я-то знаю, ведь мне потребовалось достаточно много времени, прежде чем я почувствовал себя с ними уютно, и я по-прежнему напрягаюсь всякий раз, когда создаю их), приятно, что мы уже окунулись в функциональное программирование, работая над нашим вспомогательным методом `self.wait_for`. Этот метод принимает в качестве аргумента функцию, и декоратор делает то же самое. Разница лишь в том, что декоратор фактически не выполняет программный код сам – он возвращает модифицированную версию функции, которая ему была передана.

Наш декоратор должен вернуть новую функцию, которая продолжит вызывать функцию, которая ему была передана, отлавливая наши обычные исключения, пока тайм-аут не истечет. Вот первая примерка:

functional_tests/base.py (ch18l006)

```
def wait(fn):  ❶
    def modified_fn():  ❸
        start_time = time.time()
        while True:  4
            try:
                return fn()  ❺
            except (AssertionError, WebDriverException) as e:  ❹
                if time.time() - start_time > MAX_WAIT:
                    raise e
                time.sleep(0.5)
    return modified_fn  ❷
```

❶ Декоратор представляет собой способ модификации функции; он принимает функцию как аргумент и…

❷ Возвращает другую, модифицированную (или декорированную) ее версию.

❸ Вот то место, где мы создаем модифицированную функцию.

❹ А это знакомый нам цикл, который продолжит выполняться бесконечно, отлавливая обычные исключения, пока тайм-аут не истечет.

❺ Как всегда, вызываем функцию и сразу возвращаемся, если исключения нет.

Это *почти* правильно, но не совсем. Попробуете его выполнить?

```
$ python manage.py test functional_tests.test_my_lists
[...]
```

```
    self.wait_to_be_logged_out(email)
TypeError: modified_fn() takes 0 positional arguments but 2 were given
```

В отличие от того, что происходит в `self.wait_for`, декоратор применяется к функциям, которые имеют аргументы:

functional_tests/base.py

```
@wait
def wait_to_be_logged_in(self, email):
    self.browser.find_element_by_link_text('Log out')
```

Функция `wait_to_be_logged_in` принимает `self` и `email` в качестве позиционных аргуметов. Но когда декорируется, она подменяется модифицированной функцией `modified_fn`, которая не принимает аргументов. Каким таким волшебным образом нам удается сделать так, что наша модифицированная функция `modified_fn` может обрабатывать те же самые аргументы, что и функция `fn`, которая была передана декоратору?

Ответ: немножечко волшебства Python, `*args` и `**kwargs`, более официально именуемые вариативными аргументами, или аргументами с переменным числом элементов, что совершенно очевидно (о чем я узнал совсем недавно).

functional_tests/base.py (ch18l007)

```
def wait(fn):
    def modified_fn(*args, **kwargs):  ❶
        start_time = time.time()
        while True:
            try:
                return fn(*args, **kwargs)  ❷
            except (AssertionError, WebDriverException) as e:
                if time.time() - start_time > MAX_WAIT:
                    raise e
                time.sleep(0.5)
    return modified_fn
```

❶ Используя `*args` и `**kwargs`, мы указываем, что функция `modified_fn` может принимать любые произвольные позиционные и именованные аргументы.

❷ После того как мы захватили их в функциональном определении, мы должны обеспечить передачу тех же самых параметров в `fn`, когда мы фактически ее вызываем.

Один из забавных способов ее применения – сделать декоратор, который изменяет аргументы функции. Но сейчас мы не будем вдаваться в эту тему. Главное, что теперь наш декоратор работает:

```
$ python manage.py test functional_tests.test_my_lists
[...]
OK
```

Знаете, что по-настоящему приносит удовлетворение? Мы можем использовать наш декоратор ожидания `wait` и для помощника `self.wait_for` тоже! Вот так:

functional_tests/base.py (ch18l008)

```
@wait
def wait_for(self, fn):
    return fn()
```

Прекрасно! Теперь вся логика «ожидания/повтора» инкапсулируется в единственном месте, и у нас есть крутой и простой способ применения этих ожиданий локально в ФТ с помощью функции `self.wait_for` или любой вспомогательной функции, используя декоратор `@wait`.

В следующей главе мы попробуем развернуть код на промежуточном сервере и применить на нем фикстуры с предаутентифицированным сеансом. Это поможет отловить парочку дефектов в программном коде!

Извлеченные уроки

Декораторы – это круто

Декораторы могут быть отличным способом обобщения разных уровней компетенций. Они позволяют писать тестовые утверждения и в то же самое время не думать о методах ожидания.

Повторы в ФТ ликвидируйте с осторожностью

ФТ не должны тестировать все до единой части вашего приложения. В нашем случае мы хотели избежать прохождения полного процесса входа в систему в отношении каждого ФТ, которому требуется аутентифицированный пользователь, поэтому применили тестовую фикстуру, чтобы «обмануть» тест и пропустить эту часть. В своих ФТ вы можете найти другие фрагменты, которые захотите пропустить. Однако должен предостеречь: функциональные тесты предназначены для того, чтобы отлавливать непредсказуемые взаимодействия между разными частями вашего приложения, поэтому не доводите ликвидацию повторов до крайности.

Тестовые фикстуры

Тестовые фикстуры относятся к данным тестирования, которые требуется организовать как предварительное условие, прежде чем тест будет запущен. Часто это означает заполнение базы данных некой информацией. Но, как вы убедились на примере данных cookie браузера, это может быть связано с другими типами предварительных условий.

Избегайте фикстур JSON

Django упрощает сохранение и восстановление информации из базы данных в формате JSON (и в других форматах) с использованием управляющих команд `dumpdata` и `loaddata`. Большинство специалистов не рекомендуют использовать их для тестовых фикстур, поскольку ими крайне сложно управлять, когда схема базы данных изменяется. Используйте ORM или инструмент, подобный `factory_boy` (см. https://factoryboy.readthedocs.org/).

Глава 21

Отладка на стороне сервера

Вытолкнув несколько уровней из стека задач, которые предстояло решить, мы получили хорошие вспомогательные функции ожидания. Для чего же мы их используем? Ах да, для ожидания входа в систему. А это для чего? Так мы же только что создали способ предварительной аутентификации пользователя.

Чтобы убедиться в пудинге, надо его попробовать: использование предварительного сервера для отлавливания финальных дефектов

Все они очень хороши для выполнения функциональных тестов локально. Но как они будут работать относительно промежуточного сервера? Давайте попытаемся развернуть наш сайт. По пути мы отловим неожиданный дефект (именно для этого и существует промежуточный этап!), а затем нам нужно будет придумать способ, как управлять базой данных на тестовом сервере.

```
$ cd deploy_tools
$ fab deploy --host=elspeth@superlists-staging.ottg.eu
[...]
```

И перезапустим Gunicorn…

```
elspeth@server:$ sudo systemctl daemon-reload
elspeth@server:$ sudo systemctl restart gunicorn-superlists-staging.ottg.eu
```

Вот что происходит при выполнении функциональных тестов:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test functional_tests
======================================================================
ERROR: test_can_get_email_link_to_log_in
(functional_tests.test_login.LoginTest)
 ----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "/.../functional_tests/test_login.py", line 22, in
  test_can_get_email_link_to_log_in
    self.assertIn('Check your email', body.text)
AssertionError: 'Check your email' not found in 'Server Error (500)'

======================================================================
ERROR: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/harry/.../book-example/functional_tests/test_my_lists.py",
  line 34, in test_logged_in_users_lists_are_saved_as_my_lists
    self.wait_to_be_logged_in(email)
  File "/worskpace/functional_tests/base.py", line 42, in wait_to_be_logged_in
    self.browser.find_element_by_link_text('Log out')
    [...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: {"method":"link text","selector":"Log out"}
Stacktrace:
[...]

 ----------------------------------------------------------------------
Ran 8 tests in 27.933s

FAILED (errors=2)
```

Мы не можем войти в систему – как с реальной почтовой системой, так и с предварительно аутентифицированным сеансом. Похоже, наша крутая новая система аутентификации валит сервер напрочь.

Давайте немного попрактикуемся в отладке на стороне сервера!

Настройка журналирования

Чтобы отследить эту проблему, нам нужно настроить Gunicorn для ведения журналов операций. Скорректируйте конфигурацию Gunicorn на сервере с помощью редакторов `vi` или `nano`:

server: /etc/systemd/system/gunicorn-superlists-staging.ottg.eu.service

```
ExecStart=/home/elspeth/sites/superlists-staging.ottg.eu/virtualenv/bin/gunicorn \
--bind unix:/tmp/superlists-staging.ottg.eu.socket \
--capture-output \
--access-logfile ../access.log \
--error-logfile ../error.log \
superlists.wsgi:application
```

Эта команда поместит журнал доступа и журнал ошибок в папку *~ sites/$SITENAME*.

Вам также следует убедиться, что *settings.py* по-прежнему содержит настройки LOGGING, которые фактически отправляют результаты в консоль:

superlists/settings.py

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
        },
    },
    'root': {'level': 'INFO'},
}
```

Мы снова перезапускаем Gunicorn, и затем либо повторно выполняем ФТ, либо просто пробуем войти в систему вручную. Пока это выполняется, мы можем понаблюдать за протоколом регистрации на сервере при помощи команд:

```
elspeth@server:$ sudo systemctl daemon-reload
elspeth@server:$ sudo systemctl restart gunicorn-superlists-staging.ottg.eu
elspeth@server:$ tail -f error.log # предполагает, что мы в папке ~/sites/$SITENAME
```

Вы увидите вот такую ошибку:

```
Internal Server Error: /accounts/send_login_email
Traceback (most recent call last):
  File "/home/elspeth/sites/superlists-staging.ottg.eu/virtualenv/lib/python3.6/[...]
    response = wrapped_callback(request, *callback_args, **callback_kwargs)
  File
"/home/elspeth/sites/superlists-staging.ottg.eu/source/accounts/views.py",
line 20, in send_login_email
    [email]
[...]
    self.connection.sendmail(from_email, recipients, message.as_bytes(linesep=\
r\n))
```

```
  File "/usr/lib/python3.6/smtplib.py", line 862, in sendmail
    raise SMTPSenderRefused(code, resp, from_addr)
smtplib.SMTPSenderRefused: (530, b'5.5.1 Authentication Required. Learn
more at\n5.5.1 https://support.google.com/mail/?p=WantAuthError [...]
- gsmtp', noreply@superlists)
```

Хм, похоже Gmail отказывается посылать наши электронные письма? И отчего это может быть? Ах да, мы же не сообщили серверу пароль!

Установка секретных переменных окружения на сервере

В главе о развертывании мы познакомились со способом установки секретных значений на сервере, которые мы используем для заполнения настроек секретного ключа Django `SECRET_KEY` – создание одноразовых файлов Python в файловой системе сервера и их импорта.

В предыдущих главах мы использовали переменные окружения в оболочках для сохранения почтового пароля, поэтому давайте повторим это на сервере. Мы можем установить переменную окружения в файле конфигурации Systemd:

server: /etc/systemd/system/gunicorn-superlists-staging.ottg.eu.service

```
[Service]
User=elspeth
Environment=EMAIL_PASSWORD=yoursekritpasswordhere
WorkingDirectory=/home/elspeth/sites/superlists-staging.ottg.eu/source
[...]
```

Что касается безопасности, то одно спорное преимущество использования этого файла конфигурации заключается в том, что мы можем ограничить его полномочия только чтением на уровне root. Это именно то, что мы не в состоянии сделать для файлов приложения с исходным кодом на Python.

Сохранив этот файл и исполнив рутинную свистопляску `daemon-reload` и `restart gunicorn`, мы можем повторно выполнить наши ФТ и...

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests

[...]
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_login.py", line 25, in
```

```
  test_can_get_email_link_to_log_in
    email = mail.outbox[0]
IndexError: list index out of range

[...]

selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: {"method":"link text","selector":"Log out"}
```

Та же неполадка `my_lists`, но у нас теперь больше информации в тесте регистрации в системе: ФТ проходит дальше, сайт теперь, похоже, посылает электронные письма правильно (и журнал сервера не показывает ошибок), но мы не можем проверить электронные сообщения в исходящей почте `mail.outbox`...

Адаптация ФТ для тестирования реальных электронных писем POP3

А-а. Это все объясняет. Теперь, когда мы работаем относительно реального сервера, а не `LiveServerTestCase`, мы больше не можем инспектировать локальную исходящую почту `django.mail.outbox` с целью просмотра отправленных электронных писем.

Прежде всего нам нужно будет знать, выполняем ли мы ФТ относительно промежуточного сервера или нет. Давайте сохраним переменную `staging_server`, определенную на свойстве `self` в *base.py*:

functional_tests/base.py (ch18l009)

```
    def setUp(self):
        '''установка'''
        self.browser = webdriver.Firefox()
        self.staging_server = os.environ.get('STAGING_SERVER')
        if self.staging_server:
            self.live_server_url = 'http://' + self.staging_server
```

Теперь создадим вспомогательную функцию, которая сможет получать реальную электронную почту из реального сервера электронной почты POP3 при помощи чудовищно мучительного POP3-клиента стандартной библиотеки Python:

functional_tests/test_login.py (ch18l010)

```
import os
import poplib
import re
import time
```

```
[...]

    def wait_for_email(self, test_email, subject):
        '''ожидать электронное сообщение'''
        if not self.staging_server:
            email = mail.outbox[0]
            self.assertIn(test_email, email.to)
            self.assertEqual(email.subject, subject)
            return email.body

        email_id = None
        start = time.time()
        inbox = poplib.POP3_SSL('pop.mail.yahoo.com')
        try:
            inbox.user(test_email)
            inbox.pass_(os.environ['YAHOO_PASSWORD'])
            while time.time() - start < 60:
                # получить 10 самых новых сообщений
                count, _ = inbox.stat()
                for i in reversed(range(max(1, count - 10), count + 1)):
                    print('getting msg', i)
                    _, lines, __ = inbox.retr(i)
                    lines = [l.decode('utf8') for l in lines]
                    print(lines)
                    if f'Subject: {subject}' in lines:
                        email_id = i
                        body = '\n'.join(lines)
                        return body
                time.sleep(5)
        finally:
            if email_id:
                inbox.dele(email_id)
            inbox.quit()
```

Для тестирования я использую учетную запись Yahoo, но подойдет любая почтовая служба, которая вам нравится, коль скоро она предлагает доступ по протоколу POP3. Вам потребуется установить переменную окружения `YAHOO_PASSWORD` в консоли, которая выполняет ФТ.

И затем мы вносим в ФТ остальную часть изменений, которые требуются в результате. Сначала заполнение переменной `test_email`, которая будет иметь разные значения для тестов на локальном и на промежуточном серверах:

functional_tests/test_login.py (ch18l011-1)

```
@@ -7,7 +7,7 @@ from selenium.webdriver.common.keys import Keys

from .base import FunctionalTest

-TEST_EMAIL = 'edith@example.com'
+
 SUBJECT = 'Your login link for Superlists'

@@ -33,7 +33,6 @@ class LoginTest(FunctionalTest):
                     print('getting msg', i)
                     _, lines, __ = inbox.retr(i)
                     lines = [l.decode('utf8') for l in lines]
-                    print(lines)
                     if f'Subject: {subject}' in lines:
                         email_id = i
                         body = '\n'.join(lines)

@@ -49,6 +48,11 @@ class LoginTest(FunctionalTest):
         # Эдит заходит на офигительный сайт суперсписков и впервые
         # замечает раздел «войти» в навигационной панели. Он говорит ей
         # ввести свой адрес электронной почты, что она и делает
+        if self.staging_server:
+            test_email = 'edith.testuser@yahoo.com'
+        else:
+            test_email = 'edith@example.com'
+
         self.browser.get(self.live_server_url)
```

И затем модификации, сопряженные с использованием этой переменной и вызовом нашей новой вспомогательной функции:

functional_tests/test_login.py (ch18l011-2)

```
@@ -54,7 +54,7 @@ class LoginTest(FunctionalTest):
             test_email = 'edith@example.com'
         self.browser.get(self.live_server_url)
-        self.browser.find_element_by_name('email').send_keys(TEST_EMAIL)
+        self.browser.find_element_by_name('email').send_keys(test_email)
         self.browser.find_element_by_name('email').send_keys(Keys.ENTER)

         # Появляется сообщение, которое говорит Эдит, что ей на почту
         # было выслано электронное письмо
@@ -64,15 +64,13 @@ class LoginTest(FunctionalTest):
```

```
        ))

        # Она проверяет свою почту и находит сообщение
-       email = mail.outbox[0]
-       self.assertIn(TEST_EMAIL, email.to)
-       self.assertEqual(email.subject, SUBJECT)
+       body = self.wait_for_email(test_email, SUBJECT)

        # Оно содержит ссылку на url-адрес
-       self.assertIn('Use this link to log in', email.body)
-       url_search = re.search(r'http://.+/.+$', email.body)
+       self.assertIn('Use this link to log in', body)
+       url_search = re.search(r'http://.+/.+$', body)
        if not url_search:
-           self.fail(f'Could not find url in email body:\n{email.body}')
+           self.fail(f'Could not find url in email body:\n{body}')
        url = url_search.group(0)
        self.assertIn(self.live_server_url, url)

@@ -80,11 +78,11 @@ class LoginTest(FunctionalTest):
        self.browser.get(url)

        # Она зарегистрировалась в системе!
-       self.wait_to_be_logged_in(email=TEST_EMAIL)
+       self.wait_to_be_logged_in(email=test_email)

        # Теперь она выходит из системы
        self.browser.find_element_by_link_text('Log out').click()

        # Она вышла из системы
-       self.wait_to_be_logged_out(email=TEST_EMAIL)
+       self.wait_to_be_logged_out(email=test_email)
```

Хотите верьте, хотите нет, но это действительно будет работать и даст нам ФТ, который сможет выполнять проверку на наличие регистрации в системе с участием реальных электронных писем!

Я на ваших глазах набросал этот программный код проверки адресов электронной почты, и он пока довольно уродлив и хрупок (одна из типичных проблем – он подхватывает неверный адрес электронной почты из предыдущего прогона теста). После небольшой чистки и еще нескольких циклов повторной обработки он превратится в нечто более надежное. Как вариант, службы вроде *mailinator.com* за небольшую плату предоставят вам холостые адреса электронной почты и API для их проверки.

Управление тестовой базой данных на промежуточном сервере

Теперь мы можем выполнить наши ФТ повторно и добраться до следующей неполадки: попытка создать предварительно аутентифицированные сеансы не работает, поэтому и тест «Мои списки» не срабатывает:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests

ERROR: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest)
[...]
selenium.common.exceptions.TimeoutException: Message: Could not find element
with id id_logout. Page text was:
Superlists
Sign in
Start a new To-Do list

Ran 8 tests in 72.742s

FAILED (errors=1)
```

Причина в том, что тестовая сервисная функция `create_pre_authenticated_session` работает только на локальной базе данных. Давайте узнаем, каким образом наши тесты будут управлять базой данных на сервере.

Управляющая команда Django для создания сеансов

Чтобы что-то выполнять на сервере, потребуется создать самодостаточный сценарий, который может выполняться из командной строки на сервере, скорее всего, посредством Fabric.

При попытке создать автономный сценарий, который работает с Django (то есть может обмениваться с базой данных и т. д.), возникает несколько хлопотных затруднений, которые необходимо исправить. Например, установка переменной окружения `DJANGO_SETTINGS_MODULE` и получение правильного системного пути посредством `sys.path`.

Вместо того чтобы возиться со всеми этими задачами, Django позволяет создать свои собственные управляющие команды (их можно выполнять при помощи `python manage.py`), которые за вас выполнят отладку пути. Они хранятся в папке *management/commands* внутри ваших приложений:

```
$ mkdir -p functional_tests/management/commands
$ touch functional_tests/management/__init__.py
$ touch functional_tests/management/commands/__init__.py
```

Стереотипный код в управляющей команде представлен классом, который наследует от `django.core.management.BaseCommand` и который определяет метод под названием `handle`:

functional_tests/management/commands/create_session.py

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY, get_user_model
User = get_user_model()
from django.contrib.sessions.backends.db import SessionStore
from django.core.management.base import BaseCommand

class Command(BaseCommand):
    '''команда'''

    def add_arguments(self, parser):
        '''добавить аргументы'''
        parser.add_argument('email')

    def handle(self, *args, **options):
        '''обработать'''
        session_key = create_pre_authenticated_session(options['email'])
        self.stdout.write(session_key)

def create_pre_authenticated_session(email):
    '''создать предварительно аутентифицированный сеанс'''
    user = User.objects.create(email=email)
    session = SessionStore()
    session[SESSION_KEY] = user.pk
    session[BACKEND_SESSION_KEY] = settings.AUTHENTICATION_BACKENDS[0]
    session.save()
    return session.session_key
```

Исходный код для функции `create_pre_authenticated_session` мы взяли из *test_my_lists.py*. Функция `handle` извлекает адрес электронной почты из синтаксического анализатора, а затем возвращает сеансовый ключ, который нам нужно будет добавить в данные cookie нашего браузера. Управляющая команда распечатывает его в командной оболочке. Попробуйте:

```
$ python manage.py create_session a@b.com
Unknown command: 'create_session'
```

Еще один шаг: нам нужно добавить `functional_tests` в настройки в файле *settings.py*, чтобы распознавать его как реальное приложение, которое наряду с тестами может иметь управляющие команды:

superlists/settings.py

```
+++ b/superlists/settings.py
@@ -42,6 +42,7 @@ INSTALLED_APPS = [
     'lists',
     'accounts',
+    'functional_tests',
]
```

Теперь это работает:

```
$ python manage.py create_session a@b.com
qnslckvp2aga7tm6xuivyb0ob1akzzwl
```

Если вы видите ошибку, которая сообщает, что таблица `auth_user` отсутствует, возможно, вам потребуется выполнить `manage.py migrate`. Если и она не работает, удалите файл `db.sqlite3` и снова выполните `migrate`, чтобы получить чистый лист.

Настройка ФТ для выполнения управляющей команды на сервере

Далее нам нужно скорректировать `test_my_lists`, чтобы он выполнял локальную функцию, когда мы находимся на локальном сервере, и выполнял управляющую команду на промежуточном сервере, если мы находимся на нем:

functional_tests/test_my_lists.py (ch18l016)

```
from django.conf import settings
from .base import FunctionalTest
from .server_tools import create_session_on_server
from .management.commands.create_session import create_pre_authenticated_session

class MyListsTest(FunctionalTest):
    '''тест приложения “Мои списки”'''

    def create_pre_authenticated_session(self, email):
        '''создать предварительно аутентифицированный сеанс'''
        if self.staging_server:
            session_key = create_session_on_server(self.staging_server, email)
        else:
            session_key = create_pre_authenticated_session(email)
        ## установить cookie, которые нужны для первого посещения домена.
```

```
        ## страницы 404 загружаются быстрее всего!
        self.browser.get(self.live_server_url + "/404_no_such_url/")
        self.browser.add_cookie(dict(
            name=settings.SESSION_COOKIE_NAME,
            value=session_key,
            path='/',
        ))

    [...]
```

Давайте также доработаем *base.py*, чтобы собрать немного больше информации, когда мы заполним переменную `self.against_staging`:

functional_tests/base.py (ch18l017)

```
from .server_tools import reset_database  ❶
[...]

class FunctionalTest(StaticLiveServerTestCase):
    '''функциональный тест'''

    def setUp(self):
        '''установка'''
        self.browser = webdriver.Firefox()
        self.staging_server = os.environ.get('STAGING_SERVER')
        if self.staging_server:
            self.live_server_url = 'http://' + self.staging_server
            reset_database(self.staging_server)  ❶
```

❶ Эта функция будет обнулять серверную базу данных в промежутках между каждым тестом. Я объясню логику программного кода создания сеанса, чтобы было понятнее, как это работает.

Использование fabric напрямую из Python

Вместо использования команды `fab` инструмент Fabric обеспечивает API, который позволяет выполнять серверные команды fabric непосредственно изнутри вашего исходного кода на Python. От вас только требуется сообщить ему имя хоста, к которому вы подключаетесь:

functional_tests/server_tools.py

```
from fabric.api import run
from fabric.context_managers import settings

def _get_manage_dot_py(host):
    '''получить manage точка py'''
```

```
    return f'~/sites/{host}/virtualenv/bin/python ~/sites/{host}/source/manage.py'

def reset_database(host):
    '''обнулить базу данных'''
    manage_dot_py = _get_manage_dot_py(host)
    with settings(host_string=f'elspeth@{host}'):  ❶
        run(f'{manage_dot_py} flush --noinput')  ❷

def create_session_on_server(host, email):
    '''создать сеанс на сервере'''
    manage_dot_py = _get_manage_dot_py(host)
    with settings(host_string=f'elspeth@{host}'):  ❶
        session_key = run(f'{manage_dot_py} create_session {email}')  ❷
        return session_key.strip()
```

❶ Это контекстный менеджер, который устанавливает имя узла в форме *user@server-address* (я использовал жестко кодированное серверное имя `elspeth`, поэтому скорректируйте, как надо).

❷ Затем, находясь в контекстном менеджере, мы можем легко вызывать команды fabric, как будто находимся в fabfile.

Резюме: создание сеансов локально по сравнению с промежуточным сервером

В этом и правда есть смысл? Возможно, небольшая ASCII-схема поможет разобраться:

Локально:

```
+-----------------------------------+     +-------------------------------------+
| MyListsTest                       | --> | .management.commands.create_session |
| .create_pre_authenticated_session |     |  .create_pre_authenticated_session  |
|           (локально)              |     |             (локально)              |
+-----------------------------------+     +-------------------------------------+
```

На промежуточном сервере:

```
+-----------------------------------+     +-------------------------------------+
| MyListsTest                       |     | .management.commands.create_session |
| .create_pre_authenticated_session |     |  .create_pre_authenticated_session  |
|           (локально)              |     |             (на сервере)            |
+-----------------------------------+     +-------------------------------------+
            |                                                 ^
            v                                                 |
+----------------------------+     +--------+     +-----------------------------+
| server_tools               | --> | fabric | --> | ./manage.py create_session  |
| .create_session_on_server  |     |        |     |       (на сервере)          |
|        (локально)          |     +--------+     +-----------------------------+
+----------------------------+
```

В любом случае давайте убедимся, что это работает. Прежде всего – локально, чтобы проверить, что мы ничего не повредили:

```
$ python manage.py test functional_tests.test_my_lists
[...]
OK
```

Затем – относительно сервера. Сначала отправим наш код в репозиторий:

```
$ git push # вам сначала потребуется зафиксировать изменения.
$ cd deploy_tools
$ fab deploy --host=superlists-staging.ottg.eu
```

И выполним тест:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test \
functional_tests.test_my_lists
[...]
[superlists-staging.ottg.eu] Executing task 'reset_database'
~/sites/superlists-staging.ottg.eu/source/manage.py flush --noinput
[superlists-staging.ottg.eu] out: Syncing...
[superlists-staging.ottg.eu] out: Creating tables ...
[...]
.
 ----------------------------------------------------------------------
Ran 1 test in 25.701s

OK
```

Выглядит неплохо! Можно повторно выполнить все тесты, чтобы убедиться...

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test \
functional_tests
[...]
[superlists-staging.ottg.eu] Executing task 'reset_database'
[...]
Ran 8 tests in 89.494s

OK
```

Ура!

Я показал один из способов управления тестовой базой данных, но вы можете поэкспериментировать с другими – например, если бы вы использовали MySQL или Postgres, то могли бы открыть SSH-туннель к серверу и перенаправить порты, чтобы обмениваться с базой данных напрямую. Затем при выполнении функциональных тестов вы могли бы исправить `settings.DATABASES`, чтобы обмениваться с туннелированным портом.

Предупреждение: избегайте выполнения тестового программного кода относительно действующего сервера

Теперь, когда у нас есть исходный код, который может непосредственно влиять на базу данных на сервере, мы забрались на опасную территорию. Нужно быть очень и очень осторожными, чтобы случайно не обнулить производственную базу данных в результате выполнения ФТ относительно неправильного хоста.

Находясь в этой точке, следует рассмотреть вопрос размещения неких подстраховок. Например, можно разместить промежуточный и производственный сайты на разных серверах, и сделать так, чтобы для аутентификации они использовали разные пары ключей с разными паролями.

Это такая же опасная территория, как и выполнение тестов относительно клонов производственных данных. В приложении D я рассказываю небольшую историю о случайной передаче клиентам тысяч дубликатов счетов. Учитесь на моих ошибках.

Внедрение программного кода журналирования

Прежде чем мы закончим, давайте внедрим наши настройки журналирования. Было бы полезно оставить наш новый код журналирования под версионным контролем, чтобы мы могли отлаживать любые проблемы с входом в систему. В конце концов, они могут свидетельствовать о том, что кто-то затеял что-то недоброе…

Начнем с сохранения конфигурации Gunicorn в файле шаблона в *deploy_tools*:

deploy_tools/gunicorn-systemd.template.service (ch18l020)

```
[...]
Environment=EMAIL_PASSWORD=SEKRIT
ExecStart=/home/elspeth/sites/SITENAME/virtualenv/bin/gunicorn \
    --bind unix:/tmp/SITENAME.socket \
    --access-logfile ../access.log \
    --error-logfile ../error.log \
    superlists.wsgi:application
[...]
```

И оставим небольшой файл примечаний по поводу обеспечения работы тестовой среды, где напомним о необходимости установить переменную окружения для почтового пароля посредством файла конфигурации Gunicorn:

deploy_tools/provisioning_notes.md (ch18l021)

```
## служба Systemd

* см. gunicorn-systemd.template.service
* заменить SITENAME на, например, staging.my-domain.com
* заменить SEKRIT почтовым паролем
[...]
```

Итоги

Процесс приведения нового программного кода в рабочее состояние на сервере практически всегда демонстрирует тенденцию зачищать некоторые дефекты, которые возникают в самый последний момент, и решать неожиданные проблемы. Нам пришлось выполнить небольшую работу, чтобы их преодолеть, и в результате мы получили несколько полезных вещей.

Теперь у нас есть прекрасный обобщенный декоратор ожидания `wait`, который будет хорошим Python'овским помощником для наших ФТ. У нас есть тестовые фикстуры, которые работают и локально, и на сервере, включая способность тестировать интеграцию реальной почты. И у нас есть более устойчивая конфигурация журналирования.

Но прежде чем мы сможем развернуть фактически действующий сайт, не мешало бы дать пользователям то, чего они хотят. Следующая глава описывает, как предоставить им возможность сохранять свои списки на странице «Мои списки».

Уроки, полученные при отлавливании дефектов на промежуточном сервере

Фикстуры также должны работать удаленно

`LiveServerTestCase` упрощает процесс взаимодействия с тестовой базой данных при помощи объектно-реляционного преобразователя (ORM) Django для тестов, работающих локально. Взаимодействие с базой данных на промежуточном сервере не такое прямолинейное. Одно из решений – применение инструмента fabric и управляющих команд Django, как я показал. Но вам следует изучить, что для вас подходит лучше всего – SSH-туннели, например.

Будьте очень осторожны при обнулении данных на серверах

Команда, которая может удаленно стереть всю базу данных на одном из ваших серверов, – опасное оружие, и надо быть действительно уверенным в том, что она никогда случайным образом не затронет производственные данные.

Журналирование имеет критически важное значение для отладки проблем на сервере

Как минимум, вы захотите иметь возможность наблюдать любые сообщения об ошибках, которые генерируются сервером. В отношении более щекотливых дефектов вы также захотите иметь возможность периодически делать «отладочную распечатку» и обеспечивать их запись в отдельный файл.

Глава 22

Завершение приложения «Мои списки»: TDD с подходом «снаружи внутрь»

В этой главе речь пойдет о подходе, который называется «TDD снаружи внутрь». В значительной степени именно этим мы занимались все время. Наш двухконтурный процесс TDD, в котором мы сначала пишем функциональный тест и затем модульные тесты, уже является проявлением подхода «снаружи внутрь»: мы разрабатываем систему с внешней стороны и наращиваем код уровнями. Теперь я буду говорить прямо, и затрону некоторые распространенные вопросы.

Альтернатива: «изнутри наружу»

Альтернативой подходу «снаружи внутрь» является работа «изнутри наружу». Так интуитивно работает большинство людей до встречи с методологией TDD. Когда выработана структура кода, нередко возникает естественное желание реализовать его, начиная с самых внутренних, самых низкоуровневых компонентов.

Например, при нашей текущей задаче предоставления пользователям страницы «Мои списки», состоящей из сохраняемых списков неотложных дел, искушение состоит в том, чтобы начать с добавления атрибута «владелец» в модельный объект `List`, обосновывая это тем, что атрибут такого рода будет востребован очевидным образом. Как только он занял свое место, мы модифицируем, пользуясь новым атрибутом, более периферийные уровни кода, такие как представления и шаблоны. И наконец добавляем маршрутизацию URL-адресов, чтобы направлять к новому представлению.

Вам комфортно, потому что вы никогда не работаете с фрагментом кода, который зависит от чего-то, что еще не было реализовано. Каждый фрагмент работы внутри является прочной основой, на которой можно надстроить следующий уровень.

Но такого рода работа изнутри наружу имеет несколько слабых мест.

Почему «снаружи внутрь» предпочтительнее?

Самая очевидная проблема подхода «изнутри-наружу» заключается в том, что он заставляет нас отклоняться от потока операций методологии TDD. Первая ошибка нашего функционального теста может произойти из-за отсутствующей маршрутизации URL-адресов. Но мы решаем это проигнорировать и начинаем напропалую добавлять атрибуты в наши модельные объекты базы данных.

У нас могут возникнуть идеи относительно нового желательного поведения внутренних уровней , таких как модели базы данных. Часто эти идеи действительно довольно хороши, но фактически это всего лишь догадки о том, что требуется на самом деле, потому что мы еще не создали внешние уровни, которые будут их использовать.

Одна из возможных проблем состоит в создании внутренних компонентов, более общих или наделенных бо́льшими свойствами, чем нам нужно фактически. По сути, это пустая трата времени и дополнительный источник сложностей для вашего проекта. Еще одна типичная проблема: вы создаете внутренние компоненты с помощью программного интерфейса (API), удобные для его собственного устройства. Но позже может оказаться, что они не соответствуют вызовам, которые ваши внешние уровни хотели бы делать… И что еще хуже, как вы позже поймете сами, – в конечном итоге вы можете прийти к внутренним компонентам, не решающим задачу, которую внешние уровни ожидают уже решенной.

Работа снаружи-внутрь, напротив, позволяет использовать каждый уровень, чтобы сформировать представление о самом удобном API, который вы захотите от уровня ниже его. Посмотрим на это в действии.

ФТ для «Моих списков»

При работе над следующим функциональным тестом мы начинаем с наиболее внешнего уровня (уровня презентации), затем переходим к функциям представления (или контроллерам) и, наконец, к самым внутренним уровням, которые в данном случае будут модельным программным кодом.

Мы знаем, что наш исходный код для `create_pre_authenticated_session` теперь работает, поэтому можем просто написать ФТ, который будет искать страницу «Мои списки»:

functional_tests/test_my_lists.py (ch19l001-1)

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
    '''тест: списки зарегистрированных пользователей
       сохраняются как "Мои списки"'''
    # Эдит является зарегистрированным пользователем
```

```
self.create_pre_authenticated_session('edith@example.com')

# Эдит открывает домашнюю страницу и начинает новый список
self.browser.get(self.live_server_url)
self.add_list_item('Reticulate splines')
self.add_list_item('Immanentize eschaton')
first_list_url = self.browser.current_url

# Она замечает ссылку на "Мои списки" в первый раз.
self.browser.find_element_by_link_text('My lists').click()

# Она видит, что ее список находится там, и он назван
# на основе первого элемента списка
self.wait_for(
    lambda: self.browser.find_element_by_link_text('Reticulate splines')
)
self.browser.find_element_by_link_text('Reticulate splines').click()
self.wait_for(
    lambda: self.assertEqual(self.browser.current_url, first_list_url)
)
```

Мы создаем список с парой элементов и проверяем, что он появляется на новой странице «Мои списки» и назван он по первому элементу в списке.

Давайте удостоверимся, что это действительно работает, создав второй список и убедившись, что он тоже появляется на странице «Мои списки». ФТ продолжается, и пока мы тут, мы проверяем, что только зарегистрировавшиеся пользователи могут видеть страницу «Мои списки»:

functional_tests/test_my_lists.py (ch19l001-2)

```
[...]
self.wait_for(
    lambda: self.assertEqual(self.browser.current_url, first_list_url)
)

# Она решает начать еще один список, чтобы только убедиться
self.browser.get(self.live_server_url)
self.add_list_item('Click cows')
second_list_url = self.browser.current_url

# Под заголовком "Мои списки" появляется ее новый список
self.browser.find_element_by_link_text('My lists').click()
self.wait_for(
    lambda: self.browser.find_element_by_link_text('Click cows')
)
```

```
    self.browser.find_element_by_link_text('Click cows').click()
    self.wait_for(
        lambda: self.assertEqual(self.browser.current_url, second_list_url)
    )

    # Она выходит из системы. Опция "Мои списки" исчезает
    self.browser.find_element_by_link_text('Log out').click()
    self.wait_for(lambda: self.assertEqual(
        self.browser.find_elements_by_link_text('My lists'),
         []
    ))
```

Наш ФТ использует новый вспомогательный метод – `add_list_item`, который абстрагирует детали ввода текста в нужное поле. Мы определяем его в *base.py*:

functional_tests/base.py (ch19l001-3)

```
from selenium.webdriver.common.keys import Keys
[...]

    def add_list_item(self, item_text):
        '''добавить элемент списка'''
        num_rows = len(self.browser.find_elements_by_css_selector('#id_
list_table tr'))
        self.get_item_input_box().send_keys(item_text)
        self.get_item_input_box().send_keys(Keys.ENTER)
        item_number = num_rows + 1
        self.wait_for_row_in_list_table(f'{item_number}: {item_text}')
```

И раз мы тут, можем применить его в нескольких других ФТ, вот как здесь:

functional_tests/test_list_item_validation.py

```
    self.add_list_item('Buy wellies')
```

Полагаю, это делает функциональные тесты намного более читаемыми. Всего я внес 6 изменений – убедитесь сами, и вы со мной согласитесь.

Сделаем быстрый прогон всех ФТ, фиксацию и затем вернемся к ФТ, с которым работаем. Первая ошибка должна выглядеть так:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: My lists
```

Внешний уровень: презентация и шаблоны

Этот тест в настоящее время не проходит, сообщая, что он не может найти ссылку со словами My Lists (Мои списки). Мы можем обратиться к решению этой проблемы на уровне презентации, в *base.html*, в нашей навигационной панели. Вот минимальное изменение кода:

lists/templates/base.html (ch19l002-1)

```
{% if user.email %}
  <ul class="nav navbar-nav navbar-left">
    <li><a href="#">My lists</a></li>
  </ul>
  <ul class="nav navbar-nav navbar-right">
    <li class="navbar-text">Logged in as {{ user.email }}</li>
    <li><a href="{% url 'logout' %}">Log out</a></li>
  </ul>
```

Конечно, эта ссылка фактически никуда не ведет, но в действительности она приводит к следующей ниже неполадке:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
   lambda: self.browser.find_element_by_link_text('Reticulate splines')
[...]
selenium.common.exceptions.NoSuchElementException: Message: Невозможно
локализовать элемент: Reticulate splines
```

Которая говорит, что нам необходимо создать страницу, которая перечисляет все списки пользователя по названию. Давайте начнем с основ – URL-адреса и шаблона-заготовки страницы.

Опять-таки, мы можем пойти снаружи внутрь, начиная с уровня презентации только с URL-адресом и ничем иным:

lists/templates/base.html (ch19l002-2)

```
<ul class="nav navbar-nav navbar-left">
  <li><a href="{% url 'my_lists' user.email %}">My lists</a></li>
</ul>
```

Спуск на один уровень вниз к функциям представления (к контроллеру)

Это вызовет ошибку шаблона, поэтому мы начнем перемещаться с уровня презентации и URL-адресов вниз к уровню контроллера, к функциям представления Django.

Как всегда, мы начинаем с теста:

lists/tests/test_views.py (ch19l003)

```
class MyListsTest(TestCase):

    def test_my_lists_url_renders_my_lists_template(self):
        response = self.client.get('/lists/users/a@b.com/')
        self.assertTemplateUsed(response, 'my_lists.html')
```

Это дает:

```
AssertionError: Отсутствуют шаблоны для вывода отклика в качестве HTML
```

И мы ее исправляем, все еще находясь на уровне презентации, в *urls.py*:

lists/urls.py

```
urlpatterns = [
    url(r'^new$', views.new_list, name='new_list'),
    url(r'^(\d+)/$', views.view_list, name='view_list'),
    url(r'^users/(.+)/$', views.my_lists, name='my_lists'),
]
```

Это дает нам неполадку теста, которая сообщает, что именно мы должны сделать, когда спустимся к следующему уровню:

```
AttributeError: module 'lists.views' has no attribute 'my_lists'
```

Мы движемся с уровня презентации к уровню представлений и создаем минимальную заготовку:

lists/views.py (ch19l005)

```
def my_lists(request, email):
    return render(request, 'my_lists.html')
```

И минимальный шаблон:

lists/templates/my_lists.html

```
{% extends 'base.html' %}

{% block header_text %}My Lists{% endblock %}
```

Это приводит нас к тому, что модульные тесты проходят, но ФТ остается в той же точке, говоря, что страница «Мои списки» пока еще не показывает списки. ФТ хочет, чтобы они были гиперссылками, названными по первому элементу:

```
$ python3 manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: Reticulate splines
```

Еще один проход снаружи внутрь

На каждом этапе мы по-прежнему позволяем ФТ управлять тем, какое конструктивное улучшение мы вносим.

Снова начав на внешнем уровне, в шаблоне, мы начинаем писать исходный код шаблона, который мы хотели бы применить, чтобы заставить страницу «Мои списки» работать так, как мы хотим. В силу этого мы начинаем конкретизировать API, который мы хотим от исходного кода на уровнях ниже.

Быстрая реструктуризация иерархии наследования шаблонов

Сейчас в нашем базовом шаблоне нет такого места, куда можно поместить любое новое содержимое. Кроме того, странице «Мои списки» не нужна новая форма для элемента, поэтому ее мы тоже поместим в блок, сделав необязательной:

lists/templates/base.html (ch19l007-1)

```
<div class="row">
  <div class="col-md-6 col-md-offset-3 jumbotron">
    <div class="text-center">
      <h1>{% block header_text %}{% endblock %}</h1>
      {% block list_form %}
        <form method="POST" action="{% block form_action %}{% endblock %}">
          {{ form.text }}
          {% csrf_token %}
          {% if form.errors %}
            <div class="form-group has-error">
              <div class="help-block">{{ form.text.errors }}</div>
            </div>
          {% endif %}
        </form>
      {% endblock %}
    </div>
  </div>
</div>
```

lists/templates/base.html (ch19l007-2)

```
    <div class="row">
      <div class="col-md-6 col-md-offset-3">
        {% block table %}
        {% endblock %}
      </div>
    </div>

    <div class="row">
      <div class="col-md-6 col-md-offset-3">
        {% block extra_content %}
        {% endblock %}
      </div>
    </div>

  </div>
  <script src="/static/jquery-3.1.1.min.js"></script>
  [...]
```

Конструирование API при помощи шаблона

Тем временем в *my_lists.html* мы переопределяем форму для списка `list_form` и говорим, что она не должна быть пустой.

lists/templates/my_lists.html

```
{% extends 'base.html' %}

{% block header_text %}My Lists{% endblock %}

{% block list_form %}{% endblock %}
```

И затем мы можем работать внутри блока для дополнительного наполнения `extra_content`:

lists/templates/my_lists.html

```
[...]

{% block list_form %}{% endblock %}

{% block extra_content %}
  <h2>{{ owner.email }}'s lists</h2>  ❶
  <ul>
    {% for list in owner.list_set.all %}  ❷
      <li><a href="{{ list.get_absolute_url }}">{{ list.name }}</a></li>  ❸
```

```
    {% endfor %}
  </ul>
{% endblock %}
```

В этом шаблоне мы приняли несколько проектных решений, которые будут фильтроваться по пути вниз через весь код:

❶ Мы хотим, чтобы в шаблоне переменная с именем `owner` представляла пользователя.

❷ Мы хотим иметь возможность выполнять итерацию по списку, создаваемому пользователем, используя `owner.list_set.all` (я случайно узнал, что мы их получаем бесплатно из Django ORM).

❸ Мы хотим использовать `list.name` для распечатки названия списка, который в настоящее время указывается как текст его первого элемента.

TDD на основе подхода «снаружи внутрь» иногда называют фантазийным программированием[1], то есть руководимым желанием, и вы видите почему. Мы начинаем писать код на более высоких уровнях, основываясь на том, что нам нужно, чтобы у нас было на более низких уровнях, даже если этого еще не существует!

Можем выполнить ФТ повторно, чтобы убедиться, что мы ничего не нарушили, и увидеть, что мы продвинулись дальше:

```
$ python manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: Reticulate splines

 ----------------------------------------------------------------------
Ran 8 tests in 77.613s

FAILED (errors=1)
```

Не продвинулись. Но по крайней мере мы ничего не нарушили. Пора сделать фиксацию:

```
$ git add lists
$ git diff --staged
$ git commit -m "URL, представление-заготовка и примерные шаблоны для my_lists"
```

[1] В оригинале используется термин *wishful thinking*, который может обозначать и фантазийное или благонамеренное мышление, и принятие желаемого за действительное – *Прим. перев.*

Спуск к следующему уровню: что именно представление передает в шаблон

Теперь нашему уровню представлений нужно откликнуться на требования, которые мы установили на уровне шаблона, передав его объектам, в которых он нуждается. В этом случае владелец списка:

lists/tests/test_views.py (ch19l011)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]
class MyListsTest(TestCase):
    '''тест "моих списков"'''

    def test_my_lists_url_renders_my_lists_template(self):
        '''тест: url-адрес для "моих списков" отображает
           соответствующий им шаблон'''
        [...]

    def test_passes_correct_owner_to_template(self):
        '''тест: передается правильный владелец в шаблон'''
        User.objects.create(email='wrong@owner.com')
        correct_user = User.objects.create(email='a@b.com')
        response = self.client.get('/lists/users/a@b.com/')
        self.assertEqual(response.context['owner'], correct_user)
```

Дает:

```
KeyError: 'owner'
```

Поэтому:

lists/views.py (ch19l012)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

def my_lists(request, email):
    '''мои списки'''
    owner = User.objects.get(email=email)
    return render(request, 'my_lists.html', {'owner': owner})
```

Это приводит к тому, что новый тест проходит, но мы также увидим ошибку из предыдущего теста. Нам просто нужно также добавить для него пользователя:

lists/tests/test_views.py (ch19l013)

```
    def test_my_lists_url_renders_my_lists_template(self):
        '''тест: url-адрес для "моих списков" отображает
           соответствующий им шаблон'''
        User.objects.create(email='a@b.com')
        [...]
```

И мы добираемся до OK:

```
OK
```

Следующее техническое требование из уровня представлений: новые списки должны записывать владельца

Есть еще одна часть кода на уровне представлений, которой нужно будет использовать нашу модель: нам требуется какой-то способ присваивать недавно созданные списки владельцу, если текущий пользователь зарегистрирован на сайте.

Вот первая попытка написать тест:

lists/tests/test_views.py (ch19l014)

```
class NewListTest(TestCase):
    '''тест моего списка'''
    [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        '''тест: владелец сохраняется, если
           пользователь аутентифицирован'''
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)  ❶
        self.client.post('/lists/new', data={'text': 'new item'})
        list_ = List.objects.first()
        self.assertEqual(list_.owner, user)
```

❶ `force_login()` – это способ, которым вы заставляете тестового клиента выполнять запросы с зарегистрированным пользователем.

Тест не проходит:

```
AttributeError: объект 'List' не имеет атрибута 'owner'
```

Чтобы это исправить, можно попытаться написать код вот так:

lists/views.py (ch19l015)

```
def new_list(request):
    '''новый список'''
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

Но это на самом деле работать не будет, потому что пока еще мы не знаем, как сохранить владельца списка:

```
    self.assertEqual(list_.owner, user)
AttributeError: 'List' object has no attribute 'owner'
```

Момент принятия решения: перейти к следующему уровню с неработающим тестом или нет

Чтобы получить работающий тест в том виде, как он написан сейчас, мы должны спуститься вниз на уровень модели. Однако для этого придется проделать дополнительную работу с неработающим тестом, который неидеален.

Альтернативой будет написание теста по-новому, чтобы сделать его более изолированным от уровня ниже, с помощью имитаций.

С одной стороны, использование имитаций требует намного больше усилий и может привести к тестам, которые труднее читать. С другой стороны – представьте, что ваше приложение приобрело более сложный характер и что между внешней и внутренней частями есть еще несколько уровней. Представьте, что вы оставили три, четыре или пять уровней тестов как есть, все они не работают, ожидают своей очереди, пока мы спускаемся на нижний уровень, чтобы реализовать критический функционал. Пока тесты не проходят, мы не уверены, что этот уровень действительно работает – не важно, на собственных условиях или нет. Мы должны ждать, пока не доберемся до нижнего уровня.

Это точка принятия решения, с которой вы, скорее всего, столкнетесь в своих проектах. Давайте исследуем оба подхода. Начнем с укороченного пути, оставив неработающий тест как есть. В следующей главе мы вернемся к этой конкретной точке и займемся исследованием того, как бы все пошло, если бы мы использовали больше изоляции.

Давайте сделаем фиксацию и затем пометим фиксацию *тегом*, чтобы запомнить нашу позицию для следующей главы:

```
$ git commit -am "Представление new_list пытается присвоить владельцу, но не может"
$ git tag revisit_this_point_with_isolated_tests
```

Спуск к уровню модели

Структура кода на основе принципа «снаружи внутрь» вычленила два требования к уровню модели: мы хотим иметь возможность присваивать владельца списку, используя атрибут `.owner`, и возможность получать доступ к владельцу списка при помощи API `owner.list_set.all`.

Давайте напишем для этого тест:

lists/tests/test_models.py (ch19l018)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

class ListModelTest(TestCase):
    '''тест модели списка'''

    def test_get_absolute_url(self):
        '''тест: получен абсолютный url'''
        [...]

    def test_lists_can_have_owners(self):
        '''тест: списки могут иметь владельцев'''
        user = User.objects.create(email='a@b.com')
        list_ = List.objects.create(owner=user)
        self.assertIn(list_, user.list_set.all())
```

И это дает нам новую неполадку модульного теста:

```
    list_ = List.objects.create(owner=user)
    [...]
TypeError: 'owner' является недопустимым именованным аргументом для этой функции
```

Наивная реализация была бы такой:

```
from django.conf import settings
[...]

class List(models.Model):
    '''список'''
    owner = models.ForeignKey(settings.AUTH_USER_MODEL)
```

Но мы хотим, чтобы владелец списка был необязательным. Явное лучше, чем неявное, и тесты являются формой документирования, поэтому давайте создадим тест и для этого тоже:

lists/tests/test_models.py (ch19l020)

```
    def test_list_owner_is_optional(self):
        '''тест: владелец списка является необязательным'''
        List.objects.create() # не должно поднимать исключение
```

Правильная реализация будет такой:

lists/models.py

```
from django.conf import settings
[...]

class List(models.Model):
    '''список'''
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, blank=True, null=True)

    def get_absolute_url(self):
        '''тест: получен абсолютный url'''
        return reverse('view_list', args=[self.id])
```

Теперь выполнение тестов дает обычную ошибку базы данных:

```
    return Database.Cursor.execute(self, query, params)
django.db.utils.OperationalError: нет такого столбца: lists_list.owner_id
```

Потому что нам нужно сделать дополнительные миграции:

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0006_list_owner.py
    - Add field owner to list
```

Мы почти у цели, еще пара неполадок:

```
ERROR: test_redirects_after_POST (lists.tests.test_views.NewListTest)
[...]
ValueError: Не удается присвоить "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser объект 0x7f364795ef90>>":
"List.owner" должен быть экземпляром "User".
ERROR: test_can_save_a_POST_request (lists.tests.test_views.NewListTest)

[...]
ValueError: Не удается присвоить "<SimpleLazyObject:
```

```
<django.contrib.auth.models.AnonymousUser объект 0x7f364795ef90>>":
"List.owner" должен быть экземпляром "User".
```

Теперь мы поднимаемся назад на уровень представлений, проводя небольшую подчистку. Обратите внимание, что они находятся в старом тесте для представления `new_list`, когда у нас нет зарегистрированного пользователя. Мы должны сохранять владельца списка, только когда пользователь фактически зарегистрирован. Атрибут `.is_authenticated`, который мы определили в главе 19, теперь пришелся кстати (когда пользователи не зарегистрированы, Django представляет их используя класс `AnonymousUser`, чей атрибут `.is_authenticated` всегда равняется `False`):

lists/views.py (ch19l023)

```
    if form.is_valid():
        list_ = List()
        if request.user.is_authenticated:
            list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        [...]
```

И это приводит нас к тесту, который проходит!

```
$ python manage.py test lists
[...]
.....................................
 ----------------------------------------------------------------------
Ran 39 tests in 0.237s

OK
```

Самое время для фиксации:

```
$ git add lists
$ git commit -m "Списки могут иметь владельцев, которые сохраняются при
создании."
```

Финальный шаг: подача .name API из шаблона

Последнее, что требует наша структура кода «снаружи внутрь», вытекает из шаблонов, которым нужна была возможность получать доступ к имени списка, основываясь на тексте его первого элемента:

lists/tests/test_models.py (ch19l024)

```
    def test_list_name_is_first_item_text(self):
        '''тест: имя списка является текстом первого элемента'''
```

```
        list_ = List.objects.create()
        Item.objects.create(list=list_, text='first item')
        Item.objects.create(list=list_, text='second item')
        self.assertEqual(list_.name, 'first item')
```

lists/models.py (ch19l025)

```
    @property
    def name(self):
        '''имя'''
        return self.item_set.first().text
```

Декоратор @property в Python

На случай, если вы его раньше не встречали, декоратор @property трансформирует метод, определенный на классе, чтобы во внешнем мире представлять его как атрибут.

Это мощная функциональная возможность языка, которая упрощает реализацию «утиной типизации» (то есть неявной) с целью изменения реализации свойства без изменения интерфейса класса. Другими словами, если мы решаем изменить атрибут `.name`, чтобы он стал реальным, определенным на модели атрибутом, который хранится как текст в базе данных, мы можем это сделать совершенно прозрачно. Что касается остальной части кода, то он по-прежнему будет в состоянии обращаться к `.name` и получать имя списка без необходимости знать о реализации. Несколько лет назад Реймонд Геттингер провел великолепный и понятный для начинающих инструктаж по данной теме на конференции Pycon (см. https://www.youtube.com/watch?v=HTLu2DFOdTg), который я настоятельно рекомендую (кроме того, в нем приводится миллион хороших практических приемов конструирования Python'овских классов).

Конечно, на шаблонном языке Django .name будет по-прежнему вызывать метод, даже если у него нет `@property`. Но это особенность Django, и она не применима к Python в целом…

Хотите верьте, хотите нет, но фактически это дает нам успешно проходящий тест и работающую страницу «Мои списки» (рис. 22.1)!

```
$ python manage.py test functional_tests
[...]
Ran 8 tests in 93.819s

OK
```

Но мы же знаем, что получили это обманным путем. Билли-тестировщик смотрит на нас с подозрением. Мы оставили тест невыполненным на одном уровне, в то время как реализовали его зависимости на более низ-

ком уровне. Давайте посмотрим, как все обернется, если бы мы использовали более оптимальную изоляцию теста…

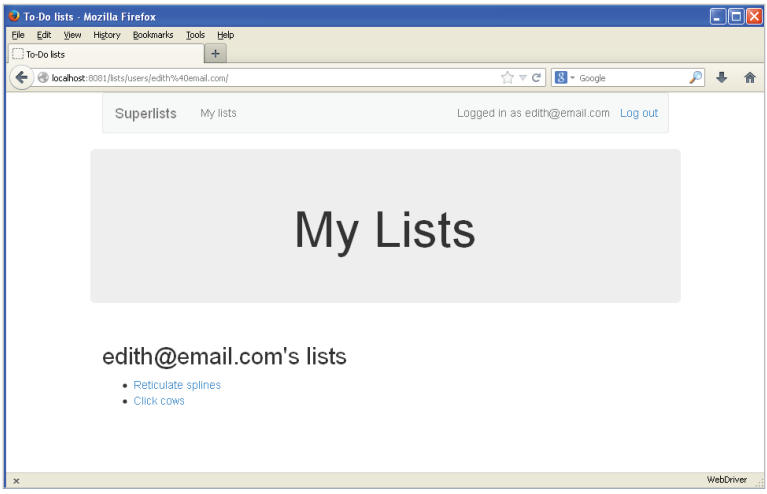


Рис. 22.1. Страница «Мои списки» во всей красе (и как доказательство, что я действительно тестировал в Windows)

Методология TDD на основе подхода «снаружи внутрь»

TDD «снаружи внутрь»

Методология конструирования программного кода на основе тестов, которая начинается с внешних уровней (презентации, графического интерфейса пользователя) и пошагово передвигается «внутрь», через уровни представлений/контроллера, вниз к уровню модели. Его суть состоит в том, чтобы управлять проектированием кода исходя из его применения, которое вы собираетесь внедрить, вместо того чтобы пытаться прогнозировать потребности снизу-вверх.

Фантазийное программирование

Процесс «снаружи внутрь» иногда называют фантазийным программированием или руководимым желанием. На самом деле любой вид TDD в некоторой степени сопряжен с принятием желаемого за действительное. Мы всегда пишем тесты для вещей, которые еще не существуют.

Ловушки подхода «снаружи внутрь»

Подход «снаружи внутрь» – это не чудодейственное средство. Он способствует тому, чтобы мы сосредотачивались на вещах, которые видимы пользователю непосредственно, но он автоматически не напомнит нам о необходимости писать другие критические тесты, которые для пользователя менее очевидны, например обеспечение безопасности. Вам придется помнить о них самим.

Глава 23

Изоляция тестов и «слушание своих тестов»

В предыдущей главе мы оставили неработающий модульный тест на уровне представлений и перешли к написанию других тестов и другого программного кода на уровне моделей, чтобы заставить его пройти успешно.

Нам это сошло с рук, потому что приложение было простым. Но я подчеркиваю, что в более сложном приложении это было бы опасно. Переход к работе на более низких уровнях, когда вы не совсем уверены, что более высокие уровни *действительно* завершены, – рискованная стратегия.

Я благодарен Гэри Бернхарду, который посмотрел черновой вариант предыдущей главы и посоветовал мне подробнее рассказать об изоляции тестов.

Обеспечение изоляции между уровнями действительно сопряжено с бо́льшими усилиями (и бо́льшим количеством ужасных имитаций!), но это также способствует вычленению улучшенной структуры кода, как мы увидим в этой главе.

Пересмотр точки принятия решения: уровень представлений зависит от ненаписанного кода моделей

Давайте пересмотрим точку, в которой мы были в середине предыдущей главы, когда не смогли заставить работать представление `new_list`, потому что списки еще не имели атрибута `.owner`.

По сути, мы проверим старую кодовую базу, чтобы увидеть, как все работало бы, если бы мы использовали более изолированные тесты.

```
$ git checkout -b more-isolation # ветка для данного эксперимента
$ git reset --hard revisit_this_point_with_isolated_tests
```

Вот так выглядят наши неработающие тесты:

lists/tests/test_views.py

```
class NewListTest(TestCase):
    '''тест нового списка'''
    [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        '''тест: владелец списка сохраняется, если
           пользователь аутентифицирован'''
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)
        self.client.post('/lists/new', data={'text': 'new item'})
        list_ = List.objects.first()
        self.assertEqual(list_.owner, user)
```

А так выглядит пробное решение:

lists/views.py

```
def new_list(request):
    '''новый список'''
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

И в этой точке тест представления не срабатывает, потому что у нас еще нет уровня модели:

```
    self.assertEqual(list_.owner, user)
AttributeError: Объект 'List' не имеет атрибута 'owner'
```

Вы не увидите эту ошибку, если не возьмете из хранилища старый код и не восстановите *lists/models.py*. Вам определенно следует это сделать, одна из задач этой главы – посмотреть, сможем ли мы писать тесты для уровня моделей, который еще не существует.

Первая попытка использования имитаций для изоляции

У списков еще нет владельцев, но мы можем позволить тестам уровня представлений притвориться, что они есть, применив несколько имитаций:

lists/tests/test_views.py (ch20l003)

```
from unittest.mock import patch
[...]

    @patch('lists.views.List')  ❶
    @patch('lists.views.ItemForm')  ❷
    def test_list_owner_is_saved_if_user_is_authenticated(
        self, mockItemFormClass, mockListClass  ❸
    ):
        '''тест: владелец списка сохраняется, если
           пользователь аутентифицирован'''
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)

        self.client.post('/lists/new', data={'text': 'new item'})

        mock_list = mockListClass.return_value  ❹
        self.assertEqual(mock_list.owner, user)  ❺
```

❶ Имитируем класс `List`, чтобы иметь возможность получать доступ к любым спискам, которые могли бы быть созданы представлением.

❷ Имитируем `ItemForm`, иначе форма поднимет ошибку, когда мы вызовем `form.save()`, потому что она не может использовать имитирующий mock-объект в качестве внешнего ключа для элемента `Item`, который она хочет создать. Когда вы начинаете имитирование, этот процесс трудно остановить!

❸ Mock-объекты внедряются в аргументы теста в порядке, обратном тому, в котором они объявлены. Тесты с большим количеством имитаций часто имеют эту странную сигнатуру с висячей скобкой `):`. Вы к ней привыкнете!

❹ Экземпляр списка, к которому представление будет иметь доступ, будет возвращаемым значением имитируемого класса `List`.

❺ И мы можем сделать утверждение, что на нем действительно установлен атрибут `.owner`.

Если теперь попытаться выполнить этот тест, он должен пройти.

```
$ python manage.py test lists
[...]
Ran 37 tests in 0.145s

OK
```

Если тест не проходит, удостоверьтесь, что ваш код представлений в *views.py* в точности такой, как я показал, – с `List()`, а не `List.objects.create`.

Использование имитаций действительно привязывает вас к конкретным приемам использования API. Это один из многих компромиссов, связанных с применением mock-объектов.

Использование имитации side_effects для проверки последовательности событий

Проблема этого теста в том, что он по-прежнему допускает написание неправильного кода по ошибке. Представьте, что мы случайно вызываем `save` до того, как присвоим владельца:

lists/views.py

```
    if form.is_valid():
        list_ = List()
        list_.save()
        list_.owner = request.user
        form.save(for_list=list_)
        return redirect(list_)
```

Тест в том виде, как он теперь написан, по-прежнему проходит:

```
OK
```

Поэтому на самом деле нам нужно проверить не только то, что владелец присвоен, но и что он присвоен до того, как мы вызовем `save` на объекте `List`.

Вот как можно протестировать последовательность событий с использованием имитаций: сымитировать функцию и использовать ее в качестве шпиона для проверки состояния мира в момент ее вызова:

lists/tests/test_views.py (ch20l005)

```
    @patch('lists.views.List')
    @patch('lists.views.ItemForm')
    def test_list_owner_is_saved_if_user_is_authenticated(
```

```
        self, mockItemFormClass, mockListClass
    ):
        '''тест: владелец списка сохраняется, если
           пользователь аутентифицирован'''
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)
        mock_list = mockListClass.return_value

        def check_owner_assigned():  ❶
            '''проверить, что владелец назначен'''
            self.assertEqual(mock_list.owner, user)
        mock_list.save.side_effect = check_owner_assigned  ❷

        self.client.post('/lists/new', data={'text': 'new item'})

        mock_list.save.assert_called_once_with()  ❸
```

❶ Определяем функцию, делающую утверждение об элементе, который мы хотим, чтобы произошел в первую очередь: проверка, что владелец списка был установлен.

❷ Присваиваем эту функцию проверки в качестве функции `side_effect` компоненту, который мы хотим проверить, что он произойдет во вторую очередь. Когда представление вызовет имитируемую функцию `save`, оно пройдет через это утверждение. Удостоверяемся, что установили его до того, как фактически вызовем функцию, которую тестируем.

❸ И наконец удостоверяемся, что функция `side_effect` была фактически инициирована, то есть мы сделали `.save()`. Иначе наше утверждение может быть фактически не выполнено никогда.

Две частые ошибки при использовании имитационных побочных эффектов: 1) побочный эффект присваивается слишком поздно, *после* того как вы вызываете тестируемую функцию; 2) не проверяется, что функция с побочным эффектом фактически вызвана. При этом прилагательное «частые» здесь означает, что я несколько раз допускал обе ошибки *при написании этой главы*.

Если в этой точке у вас по-прежнему «поврежденный» ранее означенный код, где мы присваиваем владельца, но вызываем `save` в неправильном порядке, то теперь вы должны увидеть неполадку:

```
FAIL: test_list_owner_is_saved_if_user_is_authenticated
(lists.tests.test_views.NewListTest)
```

```
[...]
  File "/.../superlists/lists/views.py", line 17, in new_list
    list_.save()
[...]
  File "/.../superlists/lists/tests/test_views.py", line 74, in
check_owner_assigned
    self.assertEqual(mock_list.owner, user)
AssertionError: <MagicMock name='List().owner' id='140691452447208'> !=
<User: User object>
```

Обратите внимание, как неполадка происходит, когда мы пытаемся сохранить и затем зайти внутрь функции `side_effect`.

Мы можем снова заставить этот тест пройти успешно вот так:

lists/views.py

```
    if form.is_valid():
        list_ = List()
        list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
...

    OK
```

Надо же, он становится уродливым!

Слушайте свои тесты: уродливые тесты сигнализируют о необходимости рефакторизации

Всегда, когда вы обнаруживаете, что вам приходится писать подобного рода тест и вам трудно с ним работать, скорее всего, ваши тесты пытаются вам что-то сообщить. Девять установочных строк (три строки для имитируемого пользователя, еще четыре – для объекта запроса и три – для функции побочного эффекта) – слишком много.

Этот тест пытается нам сказать, что наше представление делает слишком много работы, занимаясь созданием формы, нового объекта списка и решением, сохранить или нет владельца списка.

Мы уже видели, что представления можно проще и легче для понимания, передав часть работы вниз, в класс формы. Почему представление должно создавать объект списка? Ведь это мог бы сделать метод `ItemForm.save`. И почему представление должно принимать решения относительно того, сохранять `request.user` или нет? И это могла бы делать форма.

Раз мы возлагаем на эту форму больше ответственности, наверное, ей следует дать и новое имя. Мы можем назвать ее `NewListFom`, поскольку оно точнее поясняет, что она делает… Что-то вроде этого?

lists/views.py

```
# пока не вводите этот программный код, он лишь в нашем воображении.

def new_list(request):
    '''новый список'''
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user) # создает сразу List и Item
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

Это было бы круче! Давайте посмотрим, как бы мы добрались до этого состояния с помощью полностью изолированных тестов.

Написание тестов по-новому для представления, которое будет полностью изолировано

Наша первая попытка в написании комплекта тестов привела к высокой *интегрированности* этого представления. Чтобы он проходил успешно, ему требовались полнофункциональный уровень базы данных и уровень форм. Мы начали пытаться делать его более изолированным, а теперь давайте пройдем этот путь полностью.

Держите рядом старый комплект интегрированных тестов в качестве проверки на токсичность

Давайте переименуем старый класс `NewListTest` в `NewListViewIntegratedTest` и бросим попытку использовать тест с имитацией для проверки, что владелец сохраняется, отложив интегрированную версию, на время установив на него декоратор пропуска:

lists/tests/test_views.py (ch20l008)

```
import unittest
[...]

class NewListViewIntegratedTest(TestCase):
```

```
    '''интегрированный тест нового представления списка'''

    def test_can_save_a_POST_request(self):
        '''тест: может сохранить POST-запрос'''
        [...]

    @unittest.skip
    def test_list_owner_is_saved_if_user_is_authenticated(self):
        '''тест: владелец списка сохраняется, если
           пользователь аутентифицирован'''
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)
        self.client.post('/lists/new', data={'text': 'new item'})
        list_ = List.objects.first()
        self.assertEqual(list_.owner, user)
```

Вам незнаком термин «интеграционный тест» (или комплексный тест) и задаетесь вопросом, чем он от отличается от «интегрированного теста»? Перейдите к вставке с определениями в главе 26.

```
$ python manage.py test lists
[...]
Ran 37 tests in 0.139s

OK
```

Новый комплект тестов с полной изоляцией

Давайте начнем с чистого листа и убедимся, что сможем использовать изолированные тесты, чтобы управлять заменой представления `new_list`. Назовем его `new_list2`, создадим его наряду со старым представлением. Когда мы будем готовы, можно его поменять и убедиться, что старые интегрированные тесты по-прежнему проходят.

lists/views.py (ch20l009)

```
def new_list(request):
    '''новый список'''
    [...]

def new_list2(request):
    '''новый список 2'''
    pass
```

Мышление с точки зрения взаимодействующих объектов

Чтобы заново написать тесты в полностью интегрированном виде, нам придется отказаться от старого образа мыслей о тестах как о реальных эффектах представления на базу данных и вместо этого думать о представлении с точки зрения объектов, с которыми оно сотрудничает, и того, как оно с ними взаимодействует.

В новом мире основным взаимодействующим объектом представления будет объект формы. Поэтому мы его имитируем, чтобы иметь возможность полностью управлять им и с помощью фантазийного мышления определять, как мы хотим, чтобы наша форма работала.

lists/tests/test_views.py (ch20l010)

```
from unittest.mock import patch
from django.http import HttpRequest
from lists.views import new_list2
[...]

@patch('lists.views.NewListForm')  ❷
class NewListViewUnitTest(unittest.TestCase):  ❶
    '''модульный тест нового представления списка'''

    def setUp(self):
        '''установка'''
        self.request = HttpRequest()
        self.request.POST['text'] = 'new list item'  ❸

    def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
        '''тест: передаются POST-данные в новую форму списка'''
        new_list2(self.request)
        mockNewListForm.assert_called_once_with(data=self.request.POST)  ❹
```

❶ Класс Django `TestCase` сильно упрощает написание интегрированных тестов. Как способ обеспечить написание «чистых», изолированных модульных тестов мы будем использовать только `unittest.TestCase`.

❷ Мы имитируем класс `NewListForm` (который еще даже не существует). Он будет использоваться во всех тестах, поэтому имитируем его на уровне класса.

❸ Устанавливаем базовый POST-запрос в `setUp`, конструируя запрос вручную, вместо того чтобы использовать чрезмерно интегрированный тестовый клиент Django.

❹ И первое, что мы проверяем в нашем новом представлении: оно инициализирует своего сотрудника, `NewListForm`, с правильным конструктором – данными из запроса.

Он начнется с неполадки, которая говорит о том, что в представлении еще нет формы `NewListForm`.

```
AttributeError: <module 'lists.views' from '/.../superlists/lists/views.py'>
does not have the attribute 'NewListForm'
```

Давайте создадим для нее заготовку:

lists/views.py (ch20l011)

```
from lists.forms import ExistingListItemForm, ItemForm, NewListForm
[...]
```

и

lists/forms.py (ch20l012)

```
class ItemForm(forms.models.ModelForm):
    '''форма для элемента списка'''
    [...]

class NewListForm(object):
    '''форма для нового списка'''
    pass

class ExistingListItemForm(ItemForm):
    '''форма для элемента существующего списка'''

    [...]
```

Мы получаем реальную неполадку:

```
AssertionError: Ожидалось, что объект 'NewListForm' будет вызван один раз.
Вызван 0 раз.
```

И наша реализация будет выглядеть вот так:

lists/views.py (ch20l012-2)

```
def new_list2(request):
    '''новый список 2'''
    NewListForm(data=request.POST)
```

```
$ python manage.py test lists
[...]
Ran 38 tests in 0.143s

OK
```

Продолжаем. Если форма допустима, то мы хотим вызвать определенный на ней метод `save`:

lists/tests/test_views.py (ch20l013)

```
from unittest.mock import patch, Mock
[...]

@patch('lists.views.NewListForm')
class NewListViewUnitTest(unittest.TestCase):
    '''модульный тест нового представления списка'''

    def setUp(self):
        '''установка'''
        self.request = HttpRequest()
        self.request.POST['text'] = 'new list item'
        self.request.user = Mock()

    def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
        '''тест: передаются POST-данные в новую форму списка'''
        new_list2(self.request)
        mockNewListForm.assert_called_once_with(data=self.request.POST)

    def test_saves_form_with_owner_if_form_valid(self, mockNewListForm):
        '''тест: сохраняет форму с владельцем, если форма допустима'''
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = True
        new_list2(self.request)
        mock_form.save.assert_called_once_with(owner=self.request.user)
```

Это приводит нас сюда:

lists/views.py (ch20l014)

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    form.save(owner=request.user)
```

В случае, где форма допустима, мы хотим, чтобы представление возвращало переадресацию, отправляя нас наблюдать объект, который форма только что создала. Поэтому мы имитируем еще одного сотрудника представления – функцию `redirect`:

lists/tests/test_views.py (ch20l015)

```
@patch('lists.views.redirect')  ❶
def test_redirects_to_form_returned_object_if_form_valid(
```

```
    self, mock_redirect, mockNewListForm  ❷
):
    '''тест: переадресует в возвращаемый формой объект,
       если форма допустима'''
    mock_form = mockNewListForm.return_value
    mock_form.is_valid.return_value = True  ❸

    response = new_list2(self.request)

    self.assertEqual(response, mock_redirect.return_value)  ❹
    mock_redirect.assert_called_once_with(mock_form.save.return_value)  ❺
```

❶ Мы имитируем функцию переадресации `redirect`, на этот раз на уровне метода.

❷ Декораторы `patch` применяются начиная с самых внутренних, поэтому новая имитация внедряется перед `mockNewListForm`.

❸ Указываем, что тестируем случай, где форма допустима.

❹ Проверяем, что отклик из представления является результатом функции `redirect`.

❺ И проверяем, что функция переадресации была вызвана объектом, который форма возвращает при выполнении `save`.

Это приводит нас сюда:

lists/views.py (ch20l016)

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    list_ = form.save(owner=request.user)
    return redirect(list_)
```

```
$ python manage.py test lists
[...]
Ran 40 tests in 0.163s

OK
```

И теперь случай с неполадкой: если форма недопустима, мы хотим вывести (в качестве HTML) шаблон домашней страницы:

lists/tests/test_views.py (ch20l017)

```
@patch('lists.views.render')
def test_renders_home_template_with_form_if_form_invalid(
    self, mock_render, mockNewListForm
):
```

```
        '''тест: отображает домашний шаблон с формой, если форма недопустима'''
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = False
        response = new_list2(self.request)

        self.assertEqual(response, mock_render.return_value)

        mock_render.assert_called_once_with(
            self.request, 'home.html', {'form': mock_form}
        )
```

Это дает нам:

```
AssertionError: <HttpResponseRedirect status_code=302, "te[114 chars]%3E"> !=
<MagicMock name='render()' id='140244627467408'>
```

При использовании методов `assert` на объектах-имитациях, подобных `assert_called_once_with`, вдвойне важно убедиться, что вы выполняете тест и видите, что он не срабатывает. Очень легко сделать опечатку в имени функции утверждения и закончить тем, что будет вызван имитируемый метод, который ничего не делает. (Моя опечатка была в написании `asssert_called_once_with` с тремя s – попробуйте сами!)

Мы делаем преднамеренную ошибку, только чтобы удостовериться, что наши тесты исчерпывающие:

lists/views.py (ch20l018)

```
def new_list2(request):
    form = NewListForm(data=request.POST)
    list_ = form.save(owner=request.user)
    if form.is_valid():
        return redirect(list_)
    return render(request, 'home.html', {'form': form})
```

Он проходит, хотя не должен! Тогда еще один тест:

lists/tests/test_views.py (ch20l019)

```
    def test_does_not_save_if_form_invalid(self, mockNewListForm):
        '''тест: не сохраняет, если форма недопустима'''
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = False
        new_list2(self.request)
        self.assertFalse(mock_form.save.called)
```

Который не проходит:

```
    self.assertFalse(mock_form.save.called)
AssertionError: истина не является ложью
```

И мы приходим к нашему аккуратному, небольшому законченному представлению:

```
    def new_list2(request):
        form = NewListForm(data=request.POST)
        if form.is_valid():
            list_ = form.save(owner=request.user)
            return redirect(list_)
        return render(request, 'home.html', {'form': form})
...
```

```
$ python manage.py test lists
[...]
Ran 42 tests in 0.163s

OK
```

Спуск вниз на уровень форм

Итак, мы создали функцию представления, основываясь на фантазийной версии формы под названием `NewListForm`, которая еще даже не существует.

Нам понадобится метод `save` формы для создания нового списка и новый элемент на основе текста из проверенных данных POST-запроса формы. Если бы мы просто использовали ORM, исходный код мог бы выглядеть примерно так:

```
class NewListForm(models.Form):
    '''форма для нового списка'''

    def save(self, owner):
        list_ = List()
        if owner:
            list_.owner = owner
        list_.save()
        item = Item()
        item.list = list_
        item.text = self.cleaned_data['text']
        item.save()
```

Эта реализация зависит от двух классов из уровня модели – `Item` и `List`. Тогда как бы выглядел хорошо изолированный тест?

```
class NewListFormTest(unittest.TestCase):
    '''тест формы для нового списка'''

    @patch('lists.forms.List')  ❶
    @patch('lists.forms.Item')  ❶
    def test_save_creates_new_list_and_item_from_post_data(
        self, mockItem, mockList  ❶
    ):
    '''тест: сохраняет новый список и элемент из POST-данных'''
    mock_item = mockItem.return_value
    mock_list = mockList.return_value
    user = Mock()
    form = NewListForm(data={'text': 'new item text'})
    form.is_valid()  ❷

    def check_item_text_and_list():
        '''проверить текст элемента и список'''
        self.assertEqual(mock_item.text, 'new item text')
        self.assertEqual(mock_item.list, mock_list)
        self.assertTrue(mock_list.save.called)
    mock_item.save.side_effect = check_item_text_and_list  ❸

    form.save(owner=user)

    self.assertTrue(mock_item.save.called)  ❹
```

❶ Имитируем этих двух сотрудников для нашей формы из уровня моделей ниже.

❷ Нам нужно вызвать `is_valid()`, чтобы форма заполнила словарь `.cleaned_data`, где она хранит проверенные данные.

❸ Применяем метод `side_effect`, чтобы удостовериться, что, сохраняя объект нового элемента, мы делаем это с сохраненным списком и с правильным текстом элемента.

❹ Как всегда, перепроверяем, что функция побочного эффекта была фактически вызвана.

Фу, какой уродливый тест!

Продолжайте слушать свои тесты: удаление программного кода ORM из нашего приложения

И снова эти тесты пытаются нам что-то сказать: объектно-реляционный преобразователь Django (ORM) трудно имитировать, и нашему классу формы нужно слишком много знать о том, как он работает. И снова фантазий-

ное программирование, чтобы выяснить, какой более простой API могла бы использовать наша форма? Как насчет чего-то подобного:

```
    def save(self):
        List.create_new(first_item_text=self.cleaned_data['text'])
```

Наше фантазийное мышление подсказывает: а что если у нас есть вспомогательный метод, который будет располагаться на классе `List`[1] и инкапсулировать всю логику сохранения нового объекта `List` и связанного с ним первого элемента?

В качестве альтернативы напишем для этого тест:

lists/tests/test_forms.py (ch20l021)

```
import unittest

from unittest.mock import patch, Mock
from django.test import TestCase
from lists.forms import (
    DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR,
    ExistingListItemForm, ItemForm, NewListForm
)
from lists.models import Item, List
[...]

class NewListFormTest(unittest.TestCase):
    '''тест формы для нового списка'''

    @patch('lists.forms.List.create_new')
    def test_save_creates_new_list_from_post_data_if_user_not_authenticated(
        self, mock_List_create_new
    ):
        '''тест: save создает новый список из POST-данных
           если пользователь не аутентифицирован'''
        user = Mock(is_authenticated=False)
        form = NewListForm(data={'text': 'new item text'})
        form.is_valid()
        form.save(owner=user)
        mock_List_create_new.assert_called_once_with(
            first_item_text='new item text'
        )
```

И раз мы тут, протестируем случай с аутентифицированным пользователем:

[1] Он легко мог бы быть автономной функцией, но его определение на классе модели – прекрасный способ отслеживать место его расположения и дает дополнительную подсказку о том, что он будет делать.

lists/tests/test_forms.py (ch20l022)

```
    @patch('lists.forms.List.create_new')
    def test_save_creates_new_list_with_owner_if_user_authenticated(
        self, mock_List_create_new
    ):
        '''тест: save создает новый список с владельцем,
           если пользователь аутентифицирован'''
        user = Mock(is_authenticated=True)
        form = NewListForm(data={'text': 'new item text'})
        form.is_valid()
        form.save(owner=user)
        mock_List_create_new.assert_called_once_with(
            first_item_text='new item text', owner=user
        )
```

Этот тест намного более читаем. Давайте приступим к реализации новой формы. Начнем с импорта:

lists/forms.py (ch20l023)

```
from lists.models import Item, List
```

Теперь имитация нам говорит создать заготовку для метода `create_new`:

```
AttributeError: У <class 'lists.models.List'> нет атрибута 'create_new'
```

lists/models.py

```
class List(models.Model):

    def get_absolute_url(self):
        '''получить абсолютный url'''
        return reverse('view_list', args=[self.id])

    def create_new():
        '''создать новый'''
        pass
```

И после нескольких шагов мы должны прийти к методу `save` формы:

lists/forms.py (ch20l025)

```
class NewListForm(ItemForm):
    '''форма для нового списка'''

    def save(self, owner):
        if owner.is_authenticated:
            List.create_new(first_item_text=self.cleaned_data['text'],
```

```
owner=owner)
        else:
            List.create_new(first_item_text=self.cleaned_data['text'])
```

И к тестам, которые проходят:

```
$ python manage.py test lists
Ran 44 tests in 0.192s

OK
```

Сокрытие кода ORM позади вспомогательных методов

Одним из методов, появившихся в результате использования изолированных тестов, был «вспомогательный метод ORM».

ORM Django позволяет добиваться цели быстро и с довольно читаемым синтаксисом (разумеется, он намного лучше, чем необработанный SQL). Но некоторые разработчики предпочитают минимизировать объем исходного кода ORM в приложении, удаляя его из уровней форм и представлений.

Такой подход намного упрощает тестирование этих уровней. Но есть и еще одна причина: он вынуждает создавать вспомогательные функции, которые четче выражают предметную логику. Сравните:

```
        list_ = List()
        list_.save()
        item = Item()
        item.list = list_
        item.text = self.cleaned_data['text']
        item.save()
```

С:

```
    List.create_new(first_item_text=self.cleaned_data['text'])
```

Это также применимо к запросам на чтение и запись. Представьте что-то вроде этого:

```
    Book.objects.filter(in_print=True, pub_date__lte=datetime.today())
```

По сравнению со вспомогательным методом:

```
    Book.all_available_books()
```

Создавая вспомогательные функции, мы можем давать им имена, которые выражают то, что происходит с точки зрения предметной области бизнеса. Это делает код четче, а также приносит пользу от хранения всех ORM-вызовов на уровне модели и, следовательно, делает все приложение намного слабее сопряженным.

Наконец, спуск вниз на уровень моделей

На уровне моделей нам больше не нужно писать изолированные тесты – весь смысл этого уровня в интеграции с базой данных, поэтому целесообразно писать интегрированные тесты:

lists/tests/test_models.py (ch20l026)

```
class ListModelTest(TestCase):
   '''тест модели списка'''

    def test_get_absolute_url(self):
        '''тест: получен абсолютный url'''
        list_ = List.objects.create()
        self.assertEqual(list_.get_absolute_url(), f'/lists/{list_.id}/')

    def test_create_new_creates_list_and_first_item(self):
        '''тест: create_new создает список и первый элемент'''
        List.create_new(first_item_text='new item text')
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'new item text')
        new_list = List.objects.first()
        self.assertEqual(new_item.list, new_list)
```

Что дает:

```
TypeError: create_new() получила неожиданный именованный аргумент 'first_
item_text'
```

И это приведет нас к пробной реализации, которая выглядит вот так:

lists/models.py (ch20l027)

```
class List(models.Model):

    def get_absolute_url(self):
        '''получить абсолютный url'''
        return reverse('view_list', args=[self.id])

    @staticmethod
    def create_new(first_item_text):
        list_ = List.objects.create()
        Item.objects.create(text=first_item_text, list=list_)
```

Обратите внимание: нам удалось добраться до уровня моделей, управляя корректной структурой уровней представлений и форм. При этом модель списка по-прежнему не поддерживает наличие владельца!

Теперь протестируем случай, когда список должен иметь владельца, и добавим:

lists/tests/test_models.py (ch20l028)

```
from django.contrib.auth import get_user_model
User = get_user_model()
[...]

    def test_create_new_optionally_saves_owner(self):
        '''тест: create_new необязательно сохраняет владельца'''
        user = User.objects.create()
        List.create_new(first_item_text='new item text', owner=user)
        new_list = List.objects.first()
        self.assertEqual(new_list.owner, user)
```

И пока мы тут, можно написать тесты для атрибута нового владельца:

lists/tests/test_models.py (ch20l029)

```
class ListModelTest(TestCase):
    '''тест модели списка'''
    [...]

    def test_lists_can_have_owners(self):
        '''тест: списки могут иметь владельца'''
        List(owner=User()) # не должно поднять исключение

    def test_list_owner_is_optional(self):
        '''тест: владелец списка необязательный'''
        List().full_clean() # не должно поднять исключение
```

Оба этих теста почти полностью совпадают с теми, которые мы использовали в предыдущей главе. Но я немного переписал их, поэтому они не сохраняют объекты – для этого теста будет достаточным, что они будут в качестве объектов в памяти.

Используйте модельные объекты в памяти (то есть несохраненные) во всех тестах везде, где это возможно. Это ускоряет их выполнение.

Это дает:

```
$ python manage.py test lists
[...]
```

```
ERROR: test_create_new_optionally_saves_owner
TypeError: create_new() получила неожиданный именованный аргумент 'owner'
[...]
ERROR: test_lists_can_have_owners (lists.tests.test_models.ListModelTest)
TypeError: 'owner' не является допустимым именованным аргументом для этой функции
[...]
Ran 48 tests in 0.204s
FAILED (errors=2)
```

Выполняем реализацию точно так же, как в предыдущей главе:

lists/models.py (ch20l030-1)

```
from django.conf import settings
[...]

class List(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, blank=True, null=True)
    [...]
```

Это дает нам типичные ошибки целостности, пока мы не выполним миграцию:

```
django.db.utils.OperationalError: нет такого столбца: lists_list.owner_id
```

Создание миграции приведет всего к трем неполадкам:

```
ERROR: test_create_new_optionally_saves_owner
TypeError: получен неожиданный именованный аргумент 'owner'
[...]
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f5b2380b4e0>>":
"List.owner" должен быть экземпляром "User".
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7f5b237a12e8>>":
"List.owner" должен быть экземпляром "User".
```

Давайте займемся первой из них, которая касается метода `create_new`:

lists/models.py (ch20l030-3)

```
    @staticmethod
    def create_new(first_item_text, owner=None):
        list_ = List.objects.create(owner=owner)
        Item.objects.create(text=first_item_text, list=list_)
```

Назад к представлениям

Два из старых интегрированных теста для уровня представлений не срабатывают. Что же происходит?

```
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x7fbad1cb6c10>>":
"List.owner" должен быть экземпляром "User".
```

А-а, старое представление еще недостаточно различает, что оно делает с владельцами списков:

lists/views.py

```
    if form.is_valid():
        list_ = List()
        list_.owner = request.user
        list_.save()
```

В этой точке мы понимаем, что старый код не подходит для поставленной цели. Исправим его, чтобы все тесты прошли успешно:

lists/views.py (ch20l031)

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        if request.user.is_authenticated:
            list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {“form”: form})

def new_list2(request):
    [...]
```

Одно из преимуществ интегрированных тестов в том, что они помогают отлавливать менее предсказуемые взаимодействия такого рода. Мы забыли написать тест для случая, когда пользователь не аутентифицирован, но поскольку интегрированные тесты используют весь стек сверху вниз, появились ошибки из уровня модели, чтобы сообщить, что мы о чем-то забыли:

```
$ python manage.py test lists
[...]
Ran 48 tests in 0.175s

OK
```

Момент истины (и риски имитации)

Попробуем выключить наше старое представление и активировать новое. Этот обмен можно произвести в *urls.py*:

lists/urls.py

```
[...]
    url(r'^new$', views.new_list2, name='new_list'),
```

Нам также следует удалить `unittest.skip` из класса интегрированного теста, чтобы убедиться, что новый программный код для владельцев списков действительно работает:

lists/tests/test_views.py (ch20l033)

```
class NewListViewIntegratedTest(TestCase):
    '''интегрированный тест нового представления списка'''

    def test_can_save_a_POST_request(self):
        '''тест: может сохранить POST-запрос'''
        [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        '''тест: владелец списка сохраняется,
           если пользователь аутентифицирован'''
        [...]
self.assertEqual(list_.owner, user)
```

Что же произойдет, если выполнить наши тесты? О нет!

```
ERROR: test_list_owner_is_saved_if_user_is_authenticated
[...]
ERROR: test_can_save_a_POST_request
[...]
ERROR: test_redirects_after_POST
(lists.tests.test_views.NewListViewIntegratedTest)
  File "/.../superlists/lists/views.py", line 30, in new_list2
    return redirect(list_)
[...]
TypeError: три ошибки, в том числе ошибка согласования типов: аргумент с
типом 'NoneType' не является итерируемым

FAILED (errors=3)
```

Важный урок, который надо извлечь из изоляции тестов: интеграция, возможно, поможет вычленять хорошую структуру для отдельных слоев, но она автоматически не проверит допустимость интеграции *между* уровнями.

Здесь произошло следующее: представление ожидало, что форма вернет элемент списка:

lists/views.py

```
    list_ = form.save(owner=request.user)
    return redirect(list_)
```

Но мы забыли добавить, чтобы она возвращала что угодно:

```
def save(self, owner):
    if owner.is_authenticated:
        List.create_new(first_item_text=self.cleaned_data['text'], owner=owner)
    else:
        List.create_new(first_item_text=self.cleaned_data['text'])
```

Точка зрения о взаимодействиях между уровнями как о контрактах

Даже если бы мы писали исключительно изолированные модульные тесты, функциональные тесты все равно подхватили бы этот конкретный промах. Однако в идеале мы хотели бы, чтобы цикл обратной связи был быстрее: как только приложение начнет расти, выполнение функциональных тестов будет занимать несколько минут или даже несколько часов. Есть ли способ предупредить такого рода проблемы?

В методологическом отношении этот способ заключается в том, чтобы подумать о взаимодействии между уровнями с точки зрения контрактов. Всякий раз, когда мы имитируем поведение одного уровня, мы должны брать на заметку, что теперь существует неявный контракт между уровнями и что имитация на одном уровне должна, вероятно, транслироваться в тест уровнем ниже.

Вот часть контракта, который мы пропустили:

lists/tests/test_views.py

```
    @patch('lists.views.redirect')
    def test_redirects_to_form_returned_object_if_form_valid(
        self, mock_redirect, mockNewListForm
    ):
        '''тест: переадресует в возвращаемый формой объект,
           если форма допустима'''
        mock_form = mockNewListForm.return_value
        mock_form.is_valid.return_value = True

        response = new_list2(self.request)

        self.assertEqual(response, mock_redirect.return_value)
```

```
        mock_redirect.assert_called_once_with(mock_form.save.return_value)  ❶
```

❶ Имитируемая функция `form.save` возвращает объект, который, по нашим ожиданиям, представление сможет использовать.

Идентификация неявных контрактов

Стоит пересмотреть каждый тест в `NewListViewUnitTest` и разглядеть, что именно каждая имитация говорит о неявном контракте:

lists/tests/test_views.py

```
def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
    '''тест: передаются POST-данные в новую форму списка'''
    [...]
    mockNewListForm.assert_called_once_with(data=self.request.POST)  ❶

def test_saves_form_with_owner_if_form_valid(self, mockNewListForm):
    '''тест: сохраняет форму с владельцем, если форма допустима'''
    mock_form = mockNewListForm.return_value
    mock_form.is_valid.return_value = True  ❷
    new_list2(self.request)
    mock_form.save.assert_called_once_with(owner=self.request.user)  ❸

def test_does_not_save_if_form_invalid(self, mockNewListForm):
    '''тест: не сохраняет, если форма недопустима'''
    [...]
    mock_form.is_valid.return_value = False  ❷
    [...]

@patch('lists.views.redirect')
def test_redirects_to_form_returned_object_if_form_valid(
    self, mock_redirect, mockNewListForm
):
    '''тест: переадресует в возвращаемый формой объект,
       если форма допустима'''
    [...]
    mock_redirect.assert_called_once_with(mock_form.save.return_value)  ❹

@patch('lists.views.render')
def test_renders_home_template_with_form_if_form_invalid(
    '''тест: отображает домашний шаблон с формой, если форма недопустима'''
    [...]
```

❶ Необходимо инициализировать нашу форму путем передачи ей POST-запроса в качестве данных.

❷ Она должна иметь функцию `is_valid()`, которая возвращает, соответственно, `True` или `False`, опираясь на входные данные.

❸ Форма должна иметь метод `.save`, который будет принимать пользователя запроса `request.user`, который может быть зарегистрированным или нет, и обрабатывать его соответствующим образом.

❹ Метод `.save` формы должен возвращать новый объект списка, к которому наше представление переадресует пользователя.

Если просмотреть на наши тесты для формы, мы увидим, что явным образом тестируется лишь элемент 3. На элементах 1 и 2 мы были удачливы – это заданные по умолчанию свойства Django `ModelForm`, они покрываются нашими тестами для родительского класса `ItemForm`.

Но пункту 4 контракта удалось проскользнуть сквозь сеть.

При выполнении TDD снаружи внутрь, используя изолированные тесты, необходимо отслеживать неявно принятые допущения каждого теста относительно контракта, который следующий уровень должен реализовать, и помнить о необходимости протестировать каждый из них позже. Для этого можно воспользоваться блокнотом или создать тест-заготовку с `self.fail`.

Исправление недосмотра

Давайте добавим новый тест, который проверяет, что наша форма возвращает новый сохраненный список:

lists/tests/test_forms.py (ch20l038-1)

```
@patch('lists.forms.List.create_new')
def test_save_returns_new_list_object(self, mock_List_create_new):
    '''тест: save возвращает новый объект списка'''
    user = Mock(is_authenticated=True)
    form = NewListForm(data={'text': 'new item text'})
    form.is_valid()
    response = form.save(owner=user)
    self.assertEqual(response, mock_List_create_new.return_value)
```

Это хороший пример: у нас есть неявный контракт с `List.create_new`; мы хотим, чтобы он вернул новый объект списка. Для этого добавим тест-заготовку.

lists/tests/test_models.py (ch20l038-2)

```
class ListModelTest(TestCase):
    [...]

    def test_create_returns_new_list_object(self):
        '''тест: create возвращает новый объект списка'''
        self.fail()
```

Итак, у нас есть одна тестовая неполадка, которая говорит исправить сохранение формы:

```
AssertionError: None != <MagicMock name='create_new()'
id='139802647565536'>
FAILED (failures=2, errors=3)
```

Вот так:

lists/forms.py (ch20l039-1)

```
class NewListForm(ItemForm):

    def save(self, owner):
        if owner.is_authenticated:
            return List.create_new(first_item_text=self.cleaned_data['text'],\
                                   owner=owner)
        else:
            return List.create_new(first_item_text=self.cleaned_data['text'])
```

Это начало; теперь следует взглянуть на тест-заготовку:

```
[...]
FAIL: test_create_returns_new_list_object
    self.fail()
AssertionError: None

FAILED (failures=1, errors=3)
```

Конкретизируем его:

lists/tests/test_models.py (ch20l039-2)

```
    def test_create_returns_new_list_object(self):
        '''тест: create возвращает новый объект списка'''
        returned = List.create_new(first_item_text='new item text')
        new_list = List.objects.first()
        self.assertEqual(returned, new_list)
...

    AssertionError: None != <List: List object>
```

И добавляем возвращаемое значение:

lists/models.py (ch20l039-3)

```
    @staticmethod
    def create_new(first_item_text, owner=None):
        list_ = List.objects.create(owner=owner)
        Item.objects.create(text=first_item_text, list=list_)
        return list_
```

И это приводит нас к комплекту из полностью проходящих тестов:

```
$ python manage.py test lists
[...]
Ran 50 tests in 0.169s

OK
```

Еще один тест

Это был программный код для сохранения владельцев списков, управляемый тестами вплоть до самого низа и работающий. Однако функциональный тест пока не вполне проходит:

```
$ python manage.py test functional_tests.test_my_lists
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: Reticulate splines
```

Причина в том, что нам предстоит реализовать еще одно последнее свойство – атрибут `.name` на объектах списка. И снова мы можем подхватить тест и программный код из предыдущей главы:

lists/tests/test_models.py (ch20l040)

```
def test_list_name_is_first_item_text(self):
    '''тест: имя списка является текстом первого элемента'''
    list_ = List.objects.create()
    Item.objects.create(list=list_, text='first item')
    Item.objects.create(list=list_, text='second item')
    self.assertEqual(list_.name, 'first item')
```

(Поскольку это тест уровня модели, все будет нормально, если использовать ORM. Наверняка вы могли бы написать этот тест с использованием имитаций, но в этом нет необходимости.)

lists/models.py (ch20l041)

```
    @property
    def name(self):
        return self.item_set.first().text
```

И это приводит нас к ФТ, который проходит!

```
$ python manage.py test functional_tests.test_my_lists
Ran 1 test in 21.428s

OK
```

Наведение порядка: что следует убрать из комплекта интегрированных тестов

Теперь все работает, и мы можем удалить некоторые избыточные тесты и решить, какие из старых интегрированных тестов оставить.

Удаление избыточного кода на уровне форм

Мы можем избавиться от теста для старого метода `save`, определенного на `ItemForm`:

lists/tests/test_forms.py

```
--- a/lists/tests/test_forms.py
+++ b/lists/tests/test_forms.py
@@ -23,14 +23,6 @@ class ItemFormTest(TestCase):

         self.assertEqual(form.errors['text'], [EMPTY_ITEM_ERROR])

-    def test_form_save_handles_saving_to_a_list(self):
-        list_ = List.objects.create()
-        form = ItemForm(data={'text': 'do me'})
-        new_item = form.save(for_list=list_)
-        self.assertEqual(new_item, Item.objects.first())
-        self.assertEqual(new_item.text, 'do me')
-        self.assertEqual(new_item.list, list_)
-
```

Кроме того, в нашем фактическом коде мы можем избавиться от двух избыточных методов `save` в *forms.py*:

lists/forms.py

```
--- a/lists/forms.py
+++ b/lists/forms.py
@@ -22,11 +22,6 @@ class ItemForm(forms.models.ModelForm):

         self.fields['text'].error_messages['required'] = EMPTY_ITEM_ERROR

-    def save(self, for_list):
```

```
-        self.instance.list = for_list
-        return super().save()
-
-

 class NewListForm(ItemForm):

@@ -52,8 +47,3 @@ class ExistingListItemForm(ItemForm):

             e.error_dict = {'text': [DUPLICATE_ITEM_ERROR]}
             self._update_errors(e)
-
-
-    def save(self):
-        return forms.models.ModelForm.save(self)
-
```

Удаление старой реализации представления

Сейчас мы можем полностью удалить старое представление `new_list` и переименовать `new_list2` в `new_list`:

lists/tests/test_views.py

```
-from lists.views import new_list, new_list2
+from lists.views import new_list

 class HomePageTest(TestCase):
@@ -75,7 +75,7 @@ class NewListViewIntegratedTest(TestCase):
         request = HttpRequest()
         request.user = User.objects.create(email='a@b.com')
         request.POST['text'] = 'new list item'
-        new_list2(request)
+        new_list(request)
         list_ = List.objects.first()
         self.assertEqual(list_.owner, request.user)

@@ -91,21 +91,21 @@ class NewListViewUnitTest(unittest.TestCase):

     def test_passes_POST_data_to_NewListForm(self, mockNewListForm):
-        new_list2(self.request)
+        new_list(self.request)

[.. еще несколько]
```

lists/urls.py

```
--- a/lists/urls.py
+++ b/lists/urls.py
@@ -3,7 +3,7 @@ from django.conf.urls import url
 from lists import views

 urlpatterns = [
-    url(r'^new$', views.new_list2, name='new_list'),
+    url(r'^new$', views.new_list, name='new_list'),
     url(r'^(\d+)/$', views.view_list, name='view_list'),
     url(r'^users/(.+)/$', views.my_lists, name='my_lists'),
]
```

lists/views.py (ch20l047)

```
def new_list(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user)
        [...]
```

И выполнить быструю проверку, что все тесты по-прежнему проходят:

```
OK
```

Удаление избыточного кода на уровне форм

Наконец, мы должны решить, что оставить из нашего комплекта интегрированных тестов (если вообще что-то оставить).

Один из вариантов – отбросить их все и решить, что функциональные тесты засекут любые проблемы интеграции. Это вполне допустимо.

С другой стороны, мы видели, как интегрированные тесты обнаруживают небольшие ошибки в интеграции уровней. Мы могли бы оставить парочку тестов под рукой для проверки на исправность, чтобы получить более быстрый цикл обратной связи.

Как насчет этих трех:

lists/tests/test_views.py (ch20l048)

```
class NewListViewIntegratedTest(TestCase):

    def test_can_save_a_POST_request(self):
        '''тест: может сохранить POST-запрос'''
        self.client.post('/lists/new', data={'text': 'A new list item'})
        self.assertEqual(Item.objects.count(), 1)
```

```
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, 'A new list item')

    def test_for_invalid_input_doesnt_save_but_shows_errors(self):
        '''тест: недопустимый ввод не сохраняется, но показывает ошибки'''
        response = self.client.post('/lists/new', data={'text': ''})
        self.assertEqual(List.objects.count(), 0)
        self.assertContains(response, escape(EMPTY_ITEM_ERROR))

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        '''тест: владелец списка сохраняется, если
           пользователь аутентифицирован'''
        user = User.objects.create(email='a@b.com')
        self.client.force_login(user)
        self.client.post('/lists/new', data={'text': 'new item'})
        list_ = List.objects.first()
        self.assertEqual(list_.owner, user)
```

Если вы вообще собираетесь сохранить какие-либо тесты промежуточного уровня, то вот эти три мне нравятся, потому что они, похоже, делают бо́льшую часть работы по интеграции: тестируют полный стек от запроса вниз вплоть до фактической базы данных; охватывают три самых важных варианта использования нашего представления.

Выводы: когда писать изолированные тесты, а когда – интегрированные

Инструменты тестирования Django помогают быстро собирать интегрированные тесты. Исполнитель тестов с готовностью создает быструю версию вашей базы данных в оперативной памяти и обнуляет ее, когда вы находитесь между тестами. Класс `TestCase` и тестовый клиент упрощают тестирование ваших представлений начиная с проверки, были ли модифицированы объекты базы данных, подтверждая, что ваши URL-преобразования работают, и инспектируя результаты генерирования шаблонов (в качестве HTML). Это позволяет вам легко приступить к тестированию и получать хорошее покрытие тестами по всему стеку.

С другой стороны, такого рода интегрированные тесты не обязательно имеют все преимущества, которые призваны принести строгое модульное тестирование и TDD на основе подхода «снаружи внутрь» с точки зрения структуры кода.

Глядя на пример этой главы, сравните программный код, который был до этого, с тем, который получился после:

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List()
        if not isinstance(request.user, AnonymousUser):
            list_.owner = request.user
        list_.save()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})

def new_list(request):
    form = NewListForm(data=request.POST)
    if form.is_valid():
        list_ = form.save(owner=request.user)
        return redirect(list_)
    return render(request, 'home.html', {'form': form})
```

Если бы мы не потрудились спуститься вниз по маршруту изоляции, то могли бы мы подумать о рефакторизации функции представления? В первом черновом варианте книги я этого точно не сделал. И мне хотелось бы думать, что сделал бы это в реальной ситуации, но сложно сказать что-то конкретное. Однако написание изолированных тестов действительно способствует пониманию того, где в вашем коде лежит вычислительная сложность.

Пусть вычислительная сложность будет вашим гидом

Я бы сказал, что точка, в которой изолированные тесты начинают приобретать значение, связана с вычислительной сложностью. Пример в этой книге чрезвычайно прост, и поэтому до сих пор это практически стоило того. Даже в примере этой главы я могу убедить себя, что в действительности не было необходимости писать изолированные тесты.

Но как только приложение приобретает чуть бо́льшую сложность – если оно начнет наращивать какие-то новые уровни между представлениями и моделями, если вы обнаружите, что пишете вспомогательные методы или собственные классы, то вы, скорее всего, получите преимущества от написания более изолированных тестов.

Следует ли делать оба типа тестов?

У нас уже есть комплект функциональных тестов, которые выдают нам сообщения, если мы допускаем какие-либо ошибки в интеграции различных частей программного кода. Написание изолированных тестов помогает вычленять более оптимальную структуру программного кода и в мельчайших деталях проверять его допустимость. Будет ли средний уровень интеграционных тестов служить какой-либо дополнительной цели?

Считаю, что потенциально это возможно, если они обеспечат более быстрый цикл обратной связи и помогут точнее определить, от каких проблем интеграции вы страдаете. Их обратная трассировка может предоставлять более оптимальную отладочную информацию, чем, например, та, которую дает функциональный тест.

Кроме того, могут даже возникнуть основания для их компоновки как отдельного комплекта тестов: можно иметь один комплект быстрых, изолированных модульных тестов, в котором даже не используется `manage.py`, потому что им не требуется какая-то очистка и демонтаж базы данных, предоставляемые исполнителем тестов Django; затем используемый Django промежуточный уровень и, наконец, уровень функциональных тестов, который обменивается с промежуточным сервером. Это имеет смысл, если каждый уровень предоставляет инкрементные преимущества.

Решение вы принимаете по своему усмотрению. Надеюсь, прочитав эту главу, вы почувствовали, о каких компромиссах идет речь.

Вперед!

Мы вполне довольны новой версией тестов, поэтому передаем их в ветку master:

```
$ git add .
$ git commit -m "Добавлены владельцы списков через формы. Более
изолированные тесты"
$ git checkout master
$ git checkout -b master-noforms-noisolation-bak # optional backup
$ git checkout master
$ git reset --hard more-isolation # перенастроить master на нашу ветку.
```

На выполнение этих ФТ требуется раздражающе много времени. Интересно, можно ли как-то это исправить?

Об аргументах «за» и «против» разных типов тестов, и программный код отсоединения ORM

Функциональные тесты

- Обеспечивают наилучшую гарантию того, что ваше приложение действительно работает правильно с точки зрения пользователя.
- Но имеют более медленный цикл обратной связи.
- И не всегда помогают писать чистый код.

Интегрированные тесты (зависят от, например, ORM или тестового клиента Django)

- Пишутся быстро.
- Их легко понять.
- Предупреждают о любых проблемах интеграции.
- Но не всегда могут выявлять хорошую структуру кода (тут все зависит от вас).
- Обычно медленнее изолированных тестов.

Изолированные (с использованием имитаций) тесты

- Сопряжены с самой тяжелой работой.
- Их труднее читать и понять.
- Но они лучше всего обеспечивают достижение более оптимальной структуры кода.
- Выполняются быстрее всех.

Отсоединение приложения от кода, связанного с ORM

Одним из последствий попытки написать изолированные тесты стало то, что мы были вынуждены удалить весь код, связанный с ORM, из представлений и форм путем его сокрытия позади вспомогательных функций или методов. Это может быть выгодно с точки зрения отсоединения приложения от ORM и просто потому, что это делает ваш программный код более читаемым. Как и во всем остальном, вы самостоятельно принимаете решение, стоят ли дополнительные усилия достигнутого результата в конкретных условиях.

Глава 24

Непрерывная интеграция

По мере того как наш сайт растет, требуется все больше времени на выполнение всех функциональных тестов. Если это продолжится, мы можем перестать об этом беспокоиться.

Чтобы не позволить этому произойти, можно автоматизировать выполнение функциональных тестов, установив сервер непрерывной интеграции (Continuous Integration), или CI-сервер. Благодаря этому в ежедневном процессе разработки можно просто выполнять ФТ, с которым мы работаем в настоящее время, и опираться на CI-сервер, который будет выполнять все тесты автоматически и сообщать, не нарушили ли мы что-либо случайным образом. Модульные тесты останутся настолько быстрыми, что мы сможем выполнять их каждые несколько секунд.

В последнее время предпочтительным CI-сервером считается сервер под названием Jenkins. Он немного Java-ориентирован, немного сбойный, немного уродливый. Но это именно тот сервер, который используют все, и он располагает огромной экосистемой плагинов. Давайте его настроим и приведем в рабочее состояние.

Инсталляция сервера Jenkins

Существует несколько CI-служб с возможностью хостинга, которые предоставляют готовый к работе сервер Jenkins. Я наткнулся на Sauce Labs, Travis, Circle-CI, ShiningPanda, и по-видимому, их еще больше. Но я буду исходить из того, что мы все инсталлируем на сервере, который контролируем.

Не следует устанавливать Jenkins на том же сервере, что и промежуточный или производственный серверы. Помимо всего прочего, Jenkins может понадобиться для перезагрузки промежуточного сервера!

Установим последнюю версию из официального репозитория apt сервера Jenkins, потому что версия, принятая в Ubuntu по умолчанию, до сих

пор имеет несколько раздражающих дефектов, связанных с поддержкой локальной кодировки и Юникода. И она также по умолчанию не настраивается на прослушивание общедоступного Интернета:

```
root@server:$ wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key |\
     apt-key add -
root@server:$ echo deb http://pkg.jenkins.io/debian-stable binary/ | tee \
    /etc/apt/sources.list.d/jenkins.list
root@server:$ apt-get update
root@server:$ apt-get install jenkins
```

(Инструкции взяты с сайта Jenkins (https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+on+Ubuntu)

Пока мы тут, установим несколько других зависимостей:

```
root@server:$ apt-get install firefox python3-venv xvfb
# и чтобы собрать fabric3:
root@server:$ apt-get install build-essential libssl-dev libffi-dev
```

А также скачаем, разархивируем и установим geckodriver (во время написания главы он имел версию v0.17, но при чтении этого материала вы можете заменить последней версией).

```
root@server:$ wget https://github.com/mozilla/geckodriver/releases\
/download/v0.17.0/geckodriver-v0.17.0-linux64.tar.gz
root@server:$ tar -xvzf geckodriver-v0.17.0-linux64.tar.gz
root@server:$ mv geckodriver /usr/local/bin
root@server:$ geckodriver --version
geckodriver 0.17.0
```

Увеличение файла подкачки

Jenkins требует довольно много памяти, и если вы его выполняете на малой виртуальной машине (VM) объемом менее двух гигабайт RAM, он вероятно, будет убит ошибкой нехватки памяти (OOM), если только вы не увеличите файл подкачки.

```
$ fallocate -l 4G /swapfile
$ mkswap /swapfile
$ chmod 600 /swapfile
$ swapon /swapfile
```

Этого будет достаточно.

Конфигурирование сервера Jenkins

Теперь вы готовы зайти на Jenkins по URL/IP вашего сервера на порту 8080 и увидите нечто, как на рис. 24.1.

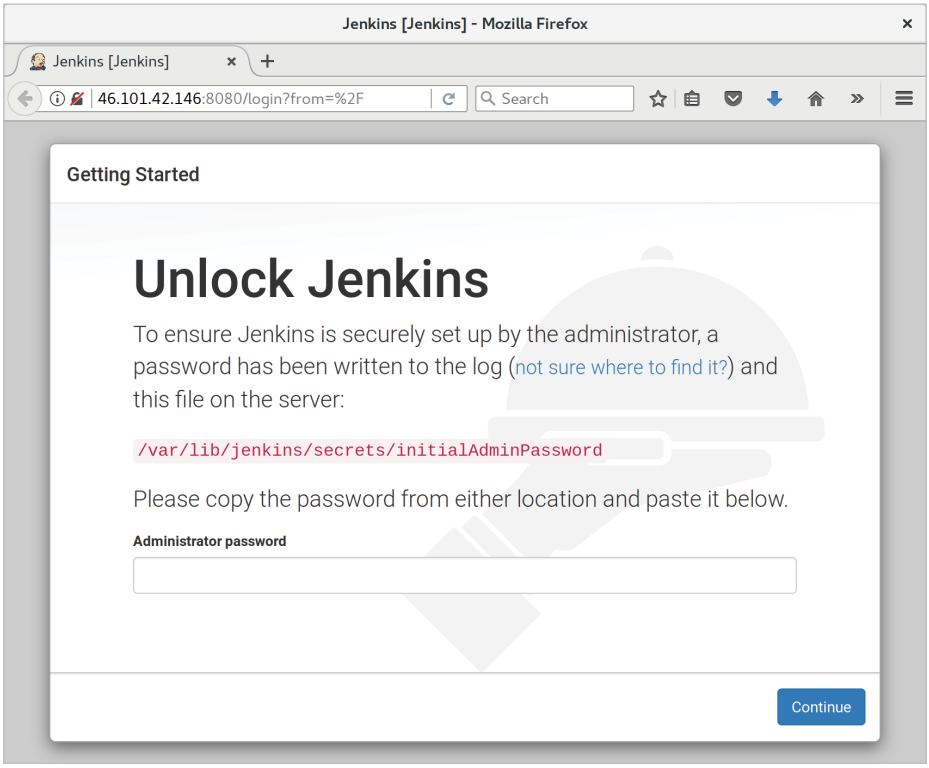


Рис. 24.1. Экран разблокировки Jenkins

Первоначальная разблокировка

При первом запуске экран разблокировки предлагает нам прочитать файл из диска для разблокирования сервера. Я перешел на терминал и набрал следующее:

```
root@server$ cat /var/lib/jenkins/secrets/initialAdminPassword
```

Набор плагинов на первое время

Затем нам предложат плагинов на выбор. Для начала их будет достаточно. (Будучи компьютерными фанатами с чувством собственного достоинства, мы испытаем инстинктивное желание сразу же нажать **Customize** (Индивидуализировать настройки). Именно это я сделал в первый раз. Но, как оказалось, экран не дает то, что мы хотим. Не переживайте, позже мы добавим еще несколько плагинов.)

Конфигурирование пользователя-администратора

Затем мы устанавливаем имя пользователя и пароль для входа на сервер Jenkins (см. рис. 24.2):

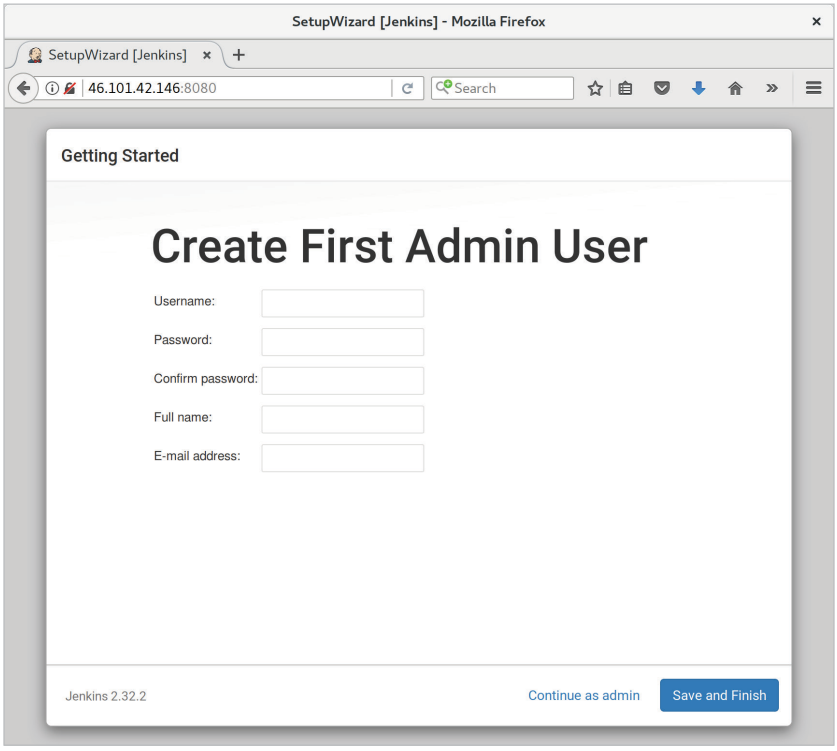


Рис. 24.2. Конфигурирование пользователя – администратора Jenkins

И когда мы окажемся в системе, то увидим экран приветствия (рис. 24.3):

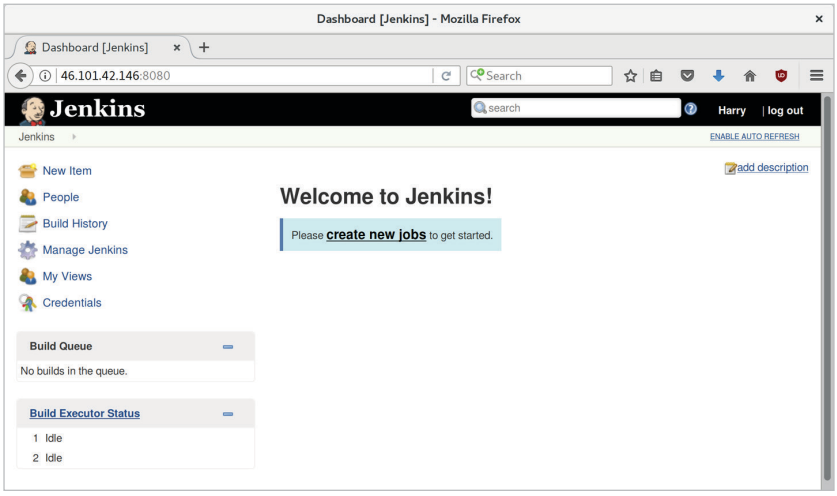


Рис. 24.3. Оригинально, дворецкий

Добавление плагинов

Пройдите по ссылкам **Manage Jenkins → Manage Plugins → Available** (Управление Jenkins → Управление плагинами → Имеющиеся в наличии)

Нам нужны плагины для:

- ShiningPanda;
- Xvfb.

И нажмите **Install** (Инсталлировать) (рис. 24.4).

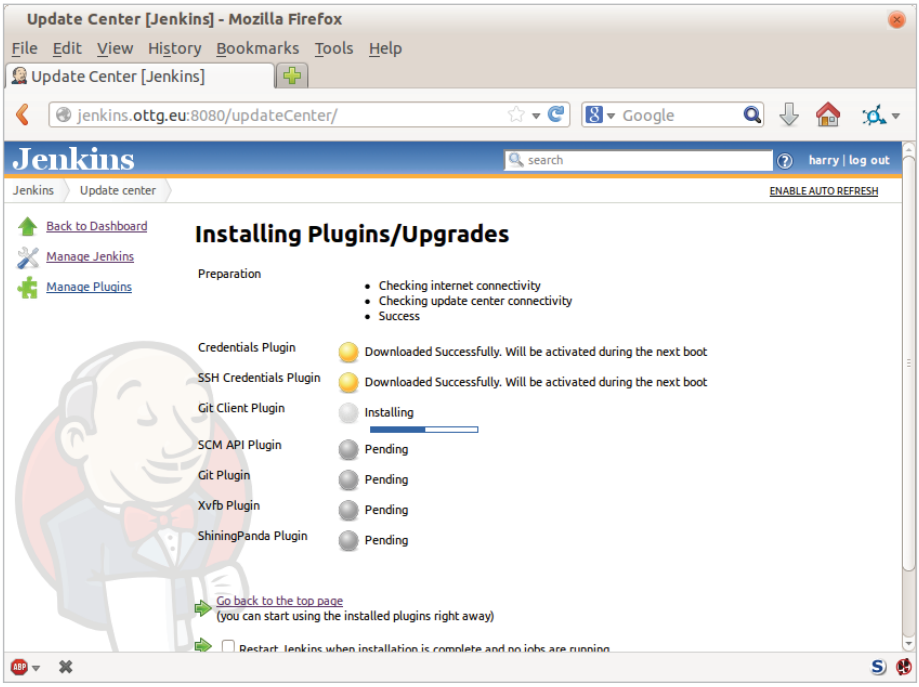


Рис. 24.4. Установка плагинов...

Указание серверу Jenkins, где искать Python 3 и Xvfb

Нам нужно сказать плагину ShiningPanda, где инсталлирован Python 3 (обычно это */usr/bin/python3*, но можно свериться командой `which python3`):

- **Manage Jenkins → Global Tool Configuration** (Управление Jenkins → Конфигурирование глобального инструментария).
- **Python → Python installations → Add Python** (Python → Инсталлированные версии Python → Добавить версию Python) (см. рис. 24.5, сообщение с предупреждением можно спокойно проигнорировать).
- **Xvfb installation → Add Xvfb installation** (Инсталлированные версии Xvfb → Добавить версию Xvfb); введите */usr/bin* в качестве инсталляционного каталога.

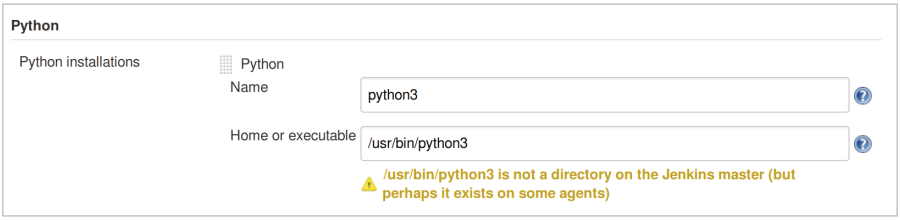


Рис. 24.5. И где я оставил этот Python?

Завершение с HTTPS

Чтобы завершить обеспечение работы экземпляра Jenkins, нужно установить HTTPS, назначить, чтобы nginx HTTPS использовал самоподписанный сертификат и запросы через прокси из порта 443 в порт 8080. Затем вы даже можете блокировать порт 8080 в брандмауэре. Сейчас я не буду вдаваться в подробности, но ниже и далее в сноске дам несколько ссылок на инструкции, которые мне показались полезными[1]:

- Официальное руководство по инсталляции Jenkins в Ubuntu;
- Как создавать самоподписанный сертификат SSL;
- Как переадресовывать HTTP в HTTPS.

Настройка проекта

Теперь, когда Jenkins установлен в базовой конфигурации, давайте установим наш проект.

- Нажмите кнопку **New Item** (Новый элемент).
- В качестве имени проекта наберите *Superlists*, затем выберите тип проекта **Freestyle project** и нажмите **OK.**
- Добавьте репозиторий Git, как на рис. 24.6.

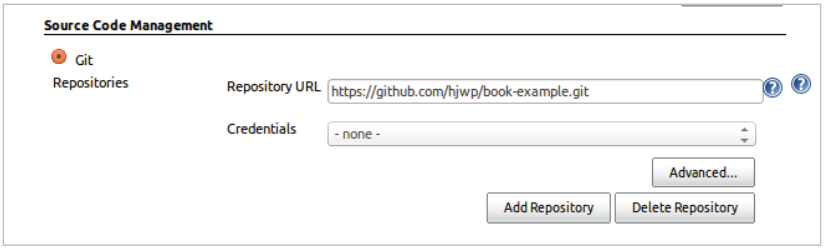


Рис. 24.6. Возьмите его в Git

- Установите опрос через каждый час (рис. 24.7) (прочтите текст справки внизу – там перечислены другие варианты инициации сборок).

[1] См.,соответственно,https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+on+Ubuntu, https://www.digitalocean.com/community/tutorials/how-to-create-an-ssl-certificate-on-nginx-for-ubuntu-14-04, http://serverfault.com/questions/250476/how-to-force-or-redirect-to-ssl-in-nginx%23424016

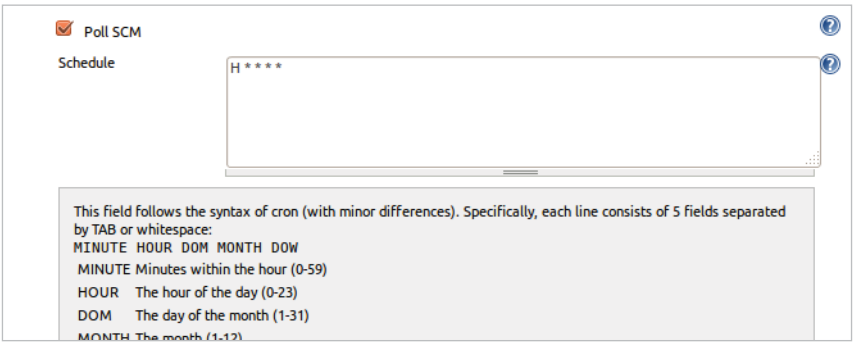


Рис. 24.7. Опрос Github по поводу изменений

- Выполните тесты внутри виртуальной среды Python 3 virtualenv.
- Выполните отдельно модульные тесты и функциональные тесты (рис. 24.8).

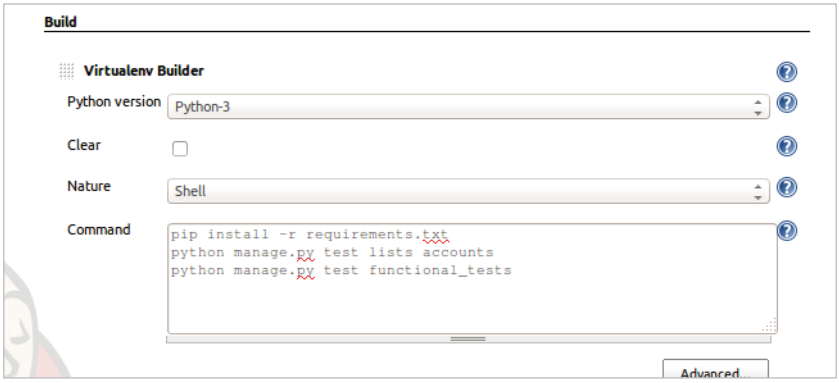


Рис. 24.8. Шаги сборки Virtualenv

Первая сборка!

Нажмите **Build Now!** (Выполнить сборку!), затем перейдите и посмотрите консольный вывод. Вы увидите что-то вроде этого:

```
Started by user harry
Building in workspace /var/lib/jenkins/jobs/Superlists/workspace
Fetching changes from the remote Git repository
Fetching upstream changes from https://github.com/hjwp/book-example.git
Checking out Revision d515acebf7e173f165ce713b30295a4a6ee17c07 (origin/master)
[workspace] $ /bin/sh -xe /tmp/shiningpanda7260707941304155464.sh
+ pip install -r requirements.txt
Requirement already satisfied (use --upgrade to upgrade): Django==1.11 in
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/
python3.3/site-packages
```

```
(from -r requirements.txt (line 1))

Requirement already satisfied (use --upgrade to upgrade): gunicorn==17.5 in
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/
python3.3/site-packages
(from -r requirements.txt (line 3))
Downloading/unpacking requests==2.0.0 (from -r requirements.txt (line 4))
  Running setup.py egg_info for package requests

Installing collected packages: requests
  Running setup.py install for requests

Successfully installed requests
Cleaning up...
+ python manage.py test lists accounts
.....................................................................
 ----------------------------------------------------------------------
Ran 67 tests in 0.429s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...
+ python manage.py test functional_tests
EEEEEE
======================================================================
ERROR: functional_tests.test_layout_and_styling (unittest.loader._FailedTest)
----------------------------------------------------------------------
ImportError: Failed to import test module: functional_tests.test_layout_
and_styling
[...]
ImportError: No module named 'selenium'

Ran 6 tests in 0.001s

FAILED (errors=6)

Build step 'Virtualenv Builder' marked build as failure
```

Ага, в нашей virtualenv нужен Selenium.

Давайте добавим инсталляцию Selenium в шаги ручной сборки:

```
pip install -r requirements.txt
python manage.py test accounts lists
pip install selenium
python manage.py test functional_tests
```

Некоторые разработчики любят использовать файл *test-requirements.txt* для перечисления пакетов, которые необходимы для тестов, но не для главного приложения.

И снова нажмите **Build Now**!

Далее произойдет одно из двух. Либо в консольном выводе вы увидите несколько сообщений об ошибках, как вот это:

```
    self.browser = webdriver.Firefox()
[...]
selenium.common.exceptions.WebDriverException: Message: 'The browser appears
to have exited before we could connect. The output was: b"\\n(process:19757):
GLib-CRITICAL **: g_slice_set_config: assertion \'sys_page_size == 0\'
failed\\nError: no display specified\\n"'
[...]
selenium.common.exceptions.WebDriverException: Message: connection refused
```

Либо, возможно, ваша сборка просто тупо зависнет (у меня это произошло по крайней мере один раз). Причина в том, что Firefox не может запуститься, потому что у него нет дисплея, в котором он будет выполняться.

Установка виртуального дисплея, чтобы ФТ выполнялись бездисплейно

Как можно увидеть из обратной трассировки, Firefox не способен запуститься, потому что сервер не имеет дисплея.

Есть два способа решить эту проблему. Первый – перейти на использование бездисплейного браузера, например PhantomJS или SlimerJS. Эти инструменты определенно имеют свое место: с одной стороны, они быстрее, но у них также есть и недостатки. Прежде всего, они не являются реальными веб-браузерами, поэтому вы не можете быть уверены, что отловите все причуды и странные формы поведения фактических браузеров, которыми пользуются люди. Второй состоит в том, что они могут вести себя по-другому внутри Selenium и часто требуют от переписать некоторые куски исходного кода ФТ.

Я бы рассматривал использование бездисплейных браузеров исключительно, как инструмент для разработки, чтобы ускорить выполнение функциональных тестов на машине разработчика, в то время как в тестах на CI-сервере используются фактические браузеры.

Альтернативой является установка виртуального дисплея: мы заставляем сервер притвориться, что к нему подключен экран и Firefox отлично работает. Для этого есть несколько инструментов; мы будем использовать один из них – Xvfb (X Virtual Framebuffer)[2], потому что его легко установить и использовать и он имеет удобный плагин Jenkins (теперь вы понимаете, зачем мы установили его).

Возвращаемся к нашему проекту и снова нажимаем **Configure** (Сконфигурировать), затем находим раздел **Build Environment** (Среда сборки). Задействование виртуального дисплея выполняется проставлением галочки напротив надписи **Start Xvfb before the build, and shut it down after** (Запустить Xvfb перед сборкой и завершить его работу после), как на рис. 24.9.

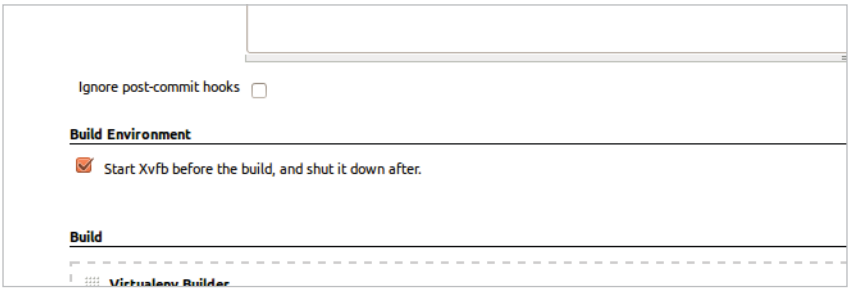


Рис. 24.9. Иногда конфигурирование дается легко

Теперь сборка проходит намного лучше:

```
[...]
Xvfb starting$ /usr/bin/Xvfb :2 -screen 0 1024x768x24 -fbdir
/var/lib/jenkins/2013-11-04_03-27-2215100124277394709 28xvfb
[...]
+ python manage.py test lists accounts
.....................................................................
 ---------------------------------------------------------------------
Ran 63 tests in 0.410s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...
+ pip install selenium
Requirement already satisfied (use --upgrade to upgrade): selenium in
/var/lib/jenkins/shiningpanda/jobs/ddc1aed1/virtualenvs/d41d8cd9/lib/
python3.5/site-packages
Cleaning up...

+ python manage.py test functional_tests
```

[2] Обратитесь к pyvirtualdisplay (https://pypi.python.org/pypi/PyVirtualDisplay) как способу контроля за виртуальными дисплеями из Python.

```
......F.
======================================================================
FAIL: test_can_start_a_list_for_one_user
(functional_tests.test_simple_list_creation.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_simple_list_creation.py",
line 43, in test_can_start_a_list_for_one_user
    self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
  File "/.../superlists/functional_tests/base.py", line 51, in
wait_for_row_in_list_table
    raise e
  File "/.../superlists/functional_tests/base.py", line 47, in
wait_for_row_in_list_table
    self.assertIn(row_text, [row.text for row in rows])
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
 ----------------------------------------------------------------------
Ran 8 tests in 89.275s

FAILED (errors=1)
Creating test database for alias 'default'...
[{'secure': False, 'domain': 'localhost', 'name': 'sessionid', 'expiry':
1920011311, 'path': '/', 'value': 'a8d8bbde33nreq6gihw8a7r1cc8bf02k'}]
Destroying test database for alias 'default'...
Build step 'Virtualenv Builder' marked build as failure
Xvfb stopping
Finished: FAILURE
```

Уже близко! Чтобы устранить эту неполадку, нам будут нужны снимки экрана.

Эта ошибка произошла из-за производительности моего экземпляра Jenkins. Вы можете наблюдать другую ошибку или никакую вообще. В любом случае пригодятся приведенные ниже инструменты для взятия снимков экрана и работы с ситуациями состязания. Продолжайте читать!

Взятие снимков экрана

Чтобы быть в состоянии устранить неожиданные неполадки, которые происходят на удаленном ПК, было бы неплохо видеть изображение экрана

во время неполадки и, возможно, также дамп HTML-страницы. Мы можем сделать это с помощью некоторой индивидуализированной логики в классе ФТ `tearDown`. Нам придется выполнить небольшую интроспекцию внутренних составляющих модуля `unittest`, приватного атрибута под названием `_outcomeForDoCleanups`, но это будет работать:

functional_tests/base.py (ch21l006)

```
import os
from datetime import datetime
[...]

SCREEN_DUMP_LOCATION = os.path.join(
    os.path.dirname(os.path.abspath(__file__)), 'screendumps'
)
[...]

    def tearDown(self):
        '''демонтаж'''
        if self._test_has_failed():
            if not os.path.exists(SCREEN_DUMP_LOCATION):
                os.makedirs(SCREEN_DUMP_LOCATION)
            for ix, handle in enumerate(self.browser.window_handles):
                self._windowid = ix
                self.browser.switch_to_window(handle)
                self.take_screenshot()
                self.dump_html()
        self.browser.quit()
        super().tearDown()

    def _test_has_failed(self):
        '''тест не сработал'''
        # слегка туманно, но не смог найти способа получше!
        return any(error for (method, error) in self._outcome.errors)
```

Сначала создаем каталог для снимков экрана. Затем выполняем итерации по всем открытым вкладкам и страницам браузера, используем несколько методов Selenium, `get_screenshot_as_file` и `browser.page_source` для наших дампов изображений и HTML:

functional_tests/base.py (ch21l007)

```
    def take_screenshot(self):
        '''взять снимок экрана'''
        filename = self._get_filename() + '.png'
        print('screenshotting to', filename)
```

```
        self.browser.get_screenshot_as_file(filename)

    def dump_html(self):
        '''выгрузить html'''
        filename = self._get_filename() + '.html'
        print('dumping page HTML to', filename)
        with open(filename, 'w') as f:
        f.write(self.browser.page_source)
```

И наконец, вот способ сгенерировать уникальный идентификатор имени файла, который включает название теста и его класса, а также метку времени:

functional_tests/base.py (ch21l008)

```
    def _get_filename(self):
        '''получить имя файла'''
        timestamp = datetime.now().isoformat().replace(':', '.')[:19]
        return '{folder}/{classname}.{method}-window{windowid}-{timestamp}'.\
            format(
                folder=SCREEN_DUMP_LOCATION,
                classname=self.__class__.__name__,
                method=self._testMethodName,
                windowid=self._windowid,
                timestamp=timestamp
            )
```

Вы можете сначала протестировать это локально, путем преднамеренного повреждения одного из тестов, например при помощи `self.fail()`, и вы увидите что-то вроде этого:

```
[...]
screenshotting to /.../superlists/functional_tests/screendumps/
MyListsTest.test
_logged_in_users_lists_are_saved_as_my_lists-window0-2014-03-09T11.19.12.png
dumping page HTML to /.../superlists/functional_tests/screendumps/MyListsTest.t
est_logged_in_users_lists_are_saved_as_my_lists-window0-[...]
```

Отмените `self.fail()`, затем зафиксируйте и внесите локальные изменения в репозиторий:

```
$ git diff # изменения в base.py
$ echo «functional_tests/screendumps» >> .gitignore
$ git commit -am "Добавлен скриншот на неполадке в исполнителе ФТ"
$ git push
```

Повторно выполнив сборку на сервере Jenkins, мы увидим что-то вроде этого:

```
screenshotting to /var/lib/jenkins/jobs/Superlists/.../functional_tests/
screendumps/LoginTest.test_login_with_persona-window0-2014-01-22T17.45.12.png
dumping page HTML to /var/lib/jenkins/jobs/Superlists/.../functional_tests/
screendumps/LoginTest.test_login_with_persona-window0-2014-01-22T17.45.12.html
```

Мы можем посетить эти ссылки в рабочей области – папке, в которой сервер Jenkins хранит исходный код и выполняет тесты, как на рис. 24.10.

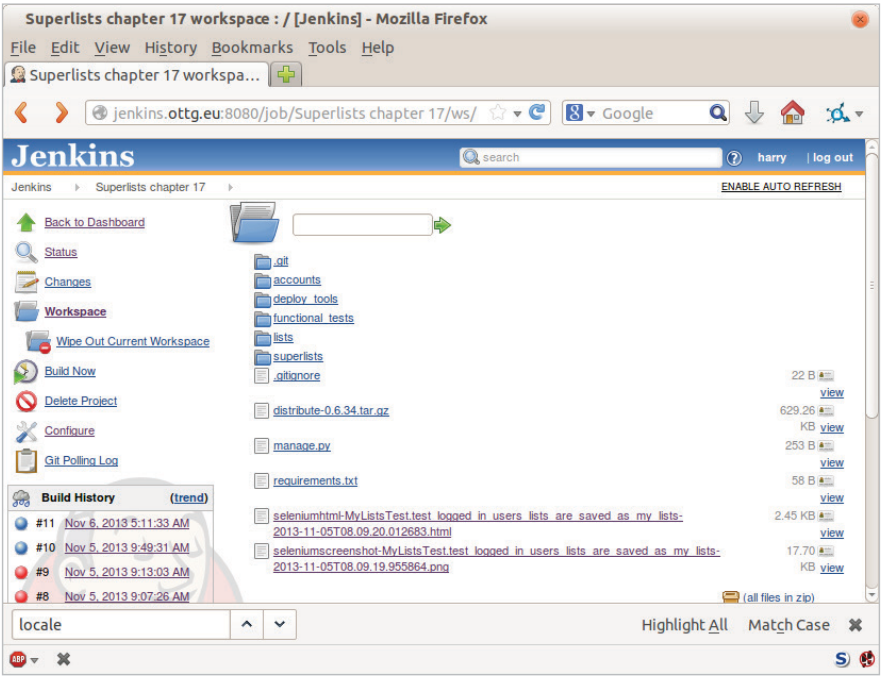


Рис. 24.10. Посещение рабочей области проекта

И затем мы смотрим на снимок экрана, как показано на рис. 24.11.

Если сомневаетесь – встряхните тайм-аут!

Хм. Никаких очевидных зацепок. Как говорится в старой пословице, если сомневаешься – встряхни компас.

functional_tests/base.py

```
MAX_WAIT = 20
```

Затем можно повторно выполнить сборку на Jenkins с помощью Build now! и убедиться, что теперь она работает, как на рис. 24.12.

Jenkins отмечает успешные сборки синим цветом, а не зеленым, что немного сбивает с толку. Но посмотрите на солнце, выглядывающее из-за облаков, – это ободряет! Это индикатор соотношения скользящего среднего значения успешных сборок к неуспешным. Дело пошло!

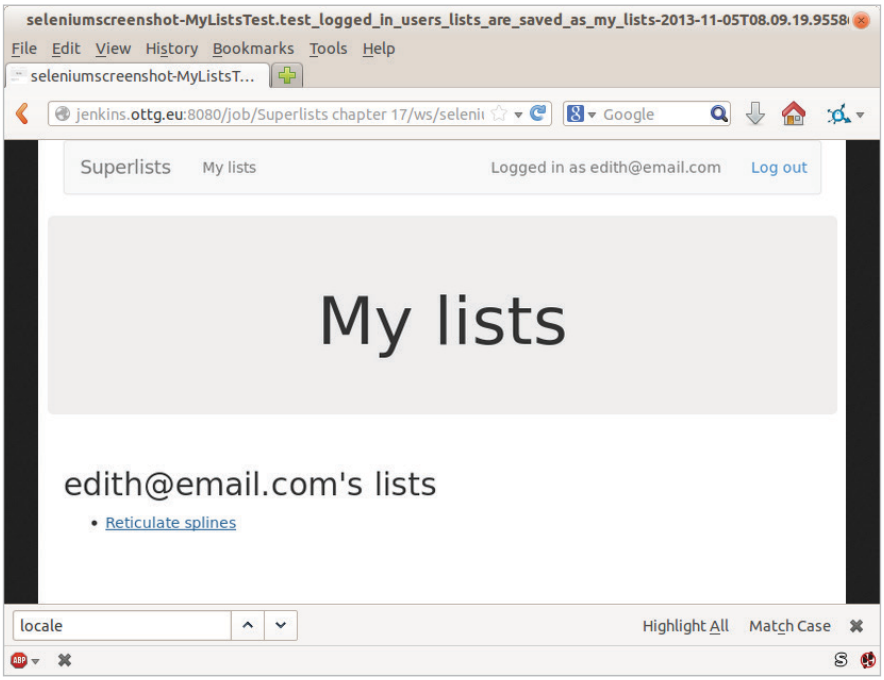


Рис. 24.11. Снимок экрана выглядит нормально

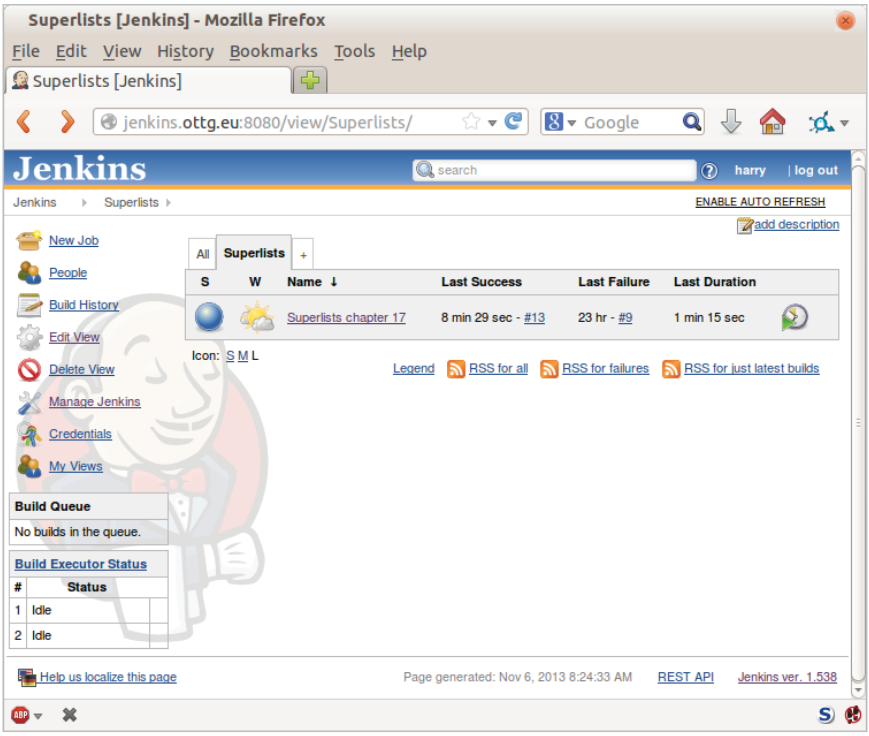


Рис. 24.12. Перспективы становятся радужнее

Выполнение тестов JavaScript QUnit на Jenkins вместе с PhantomJS

Есть ряд тестов, о которых мы почти забыли, – тесты на JavaScript. Сейчас нашим исполнителем тестов является фактический веб-браузер. Чтобы Jenkins их выполнял, нужен консольный исполнитель тестов. Хороший шанс для PhantomJS.

Установка node

Хватит притворяться, что мы не в игре с JavaScript. Мы занимаемся веб-разработкой. Это значит, что мы работаем с JavaScript, а это в свою очередь значит, что мы в конечном итоге будем использовать на наших компьютерах node.js. Так устроена жизнь.

Следуйте инструкциям на странице загрузки node.js[3]. Там имеются установщики для Windows и Mac, а также репозитории для популярных дистрибутивов Linux[4].

Как только у нас есть node, мы можем установить phantom:

```
root@server $ npm install -g phantomjs # -g означает "в масштабе системы".
```

Затем извлекаем исполнитель тестов QUnit/PhantomJS из удаленного репозитория. Их несколько (я даже написал элементарный исполнитель тестов, чтобы иметь возможность протестировать распечатки QUnit из этой книги), но, пожалуй, самый лучший – исполнитель тестов, ссылка на который указана на странице плагинов QUnit[5]. Во время написания настоящей главы его репозиторий находился по адресу https://github.com/jonkemp/qunit-phantomjs-runner. Вам потребуется всего один файл *runner.js*.

В итоге у вас будет:

```
$ tree lists/static/tests/
lists/static/tests/
├── qunit-2.0.1.css
├── qunit-2.0.1.js
├── runner.js
└── tests.html

0 directories, 4 files
```

Давайте испытаем его:

```
$ phantomjs lists/static/tests/runner.js lists/static/tests/tests.html
Took 24ms to run 2 tests. 2 passed, 0 failed.
```

[3] См. http://nodejs.org/download/

[4] Убедитесь, что вы получили последнюю версию. В Ubuntu используйте PPA, а не стандартный пакет, принятый по умолчанию.

[5] См. http://qunitjs.com/plugins/

Только чтобы убедиться, давайте преднамеренно что-нибудь нарушим:

lists/static/list.js (ch21l019)

```
$('input[name="text"]').on('keypress', function () {
  // $('.has-error').hide();
});
```

Разумеется:

```
$ phantomjs lists/static/tests/runner.js lists/static/tests/tests.html

Test failed: errors should be hidden on keypress
    Failed assertion: expected: false, but was: true
file:///.../superlists/lists/static/tests/tests.html:27:15

Took 27ms to run 2 tests. 1 passed, 1 failed.
```

Все в порядке! Теперь отменим преднамеренную ошибку, зафиксируем и внесем исполнитель в репозиторий, а затем добавим его в нашу сборку Jenkins:

```
$ git checkout lists/static/list.js
$ git add lists/static/tests/runner.js
$ git commit -m "Добавлен исполнитель тестов phantomjs для тестов на javascript"
$ git push
```

Добавление шагов сборки в Jenkins

Снова отредактируйте конфигурацию проекта и добавьте шаг для каждого набора тестов на JavaScript, как на рис. 24.13.

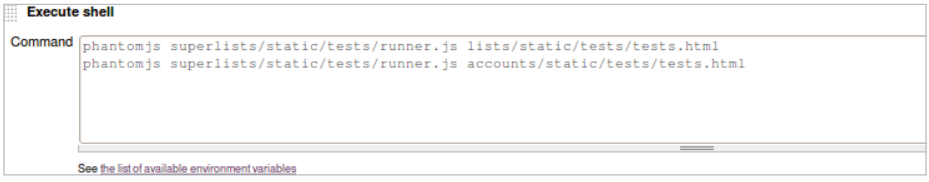


Рис. 24.13. Добавьте шаг сборки для модульных тестов на JavaScript

Вам также понадобится инсталлировать на сервере PhantomJS:

```
root@server:$ add-apt-repository -y ppa:chris-lea/node.js
root@server:$ apt-get update
root@server:$ apt-get install nodejs
root@server:$ npm install -g phantomjs
```

Вуаля! Полная сборка CI со всеми нашими тестами!

```
Started by user harry
Building in workspace /var/lib/jenkins/jobs/Superlists/workspace
Fetching changes from the remote Git repository
Fetching upstream changes from https://github.com/hjwp/book-example.git
Checking out Revision 936a484038194b289312ff62f10d24e6a054fb29 (origin/chapter_1
Xvfb starting$ /usr/bin/Xvfb :1 -screen 0 1024x768x24 -fbdir /var/lib/jenkins/20
[workspace] $ /bin/sh -xe /tmp/shiningpanda7092102504259037999.sh

+ pip install -r requirements.txt
[...]

+ python manage.py test lists
.................................
 ----------------------------------------------------------------------
Ran 43 tests in 0.229s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...

+ python manage.py test accounts
..................
 ----------------------------------------------------------------------
Ran 18 tests in 0.078s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...

[workspace] $ /bin/sh -xe /tmp/hudson2967478575201471277.sh
+ phantomjs lists/static/tests/runner.js lists/static/tests/tests.html
Took 32ms to run 2 tests. 2 passed, 0 failed.
+ phantomjs lists/static/tests/runner.js accounts/static/tests/tests.html
Took 47ms to run 11 tests. 11 passed, 0 failed.

[workspace] $ /bin/sh -xe /tmp/shiningpanda7526089957247195819.sh
+ pip install selenium
Requirement already satisfied (use --upgrade to upgrade): selenium in /var/lib/

Cleaning up...
[workspace] $ /bin/sh -xe /tmp/shiningpanda2420240268202055029.sh
+ python manage.py test functional_tests
.........
 ----------------------------------------------------------------------
Ran 8 tests in 76.804s

OK
```

Круто осознавать, что, независимо от степени моей лени, которая появляется, когда речь заходит о выполнении полного комплекта тестов на моей собственной машине, CI-сервер все равно схватит меня за руку. Еще один из агентов Билли-тестировщика, который наблюдает за нами из киберпространства…

Больше возможностей с CI-сервером

Я лишь слегка прикоснулся к тому, что можно делать с CI-серверами и сервером Jenkins. Например, его можно сделать намного умнее в том, как он отслеживает ваш репозиторий на предмет новых фиксаций.

Но, пожалуй, еще интереснее, что вы можете использовать CI-сервер для автоматизации тестов на промежуточном этапе, а также нормальных функциональных тестов. Если все ФТ проходят, можно добавить шаг сборки, который разворачивает программный код на промежуточном сервере и затем относительно него повторно выполняет все ФТ, автоматизируя еще один этап и гарантируя, что промежуточный сервер автоматически обновляется самой последней версией программного кода.

Некоторые разработчики используют CI-сервер даже в качестве средства развертывания своих производственных релизов!

Лучшие приемы использования CI и Selenium

Установите CI-сервер для вашего проекта как можно раньше

Как только на выполнение функциональных тестов начнет уходить больше нескольких секунд, вы станете избегать их выполнения всех вместе. Переложите эту работа на CI-сервер, тем самым вы обеспечите, что все тесты будут где-то выполняться.

Настройте снимки экрана и дампы HTML для неполадок

Процесс отладки в тестах проще, если вы знаете, как выглядит страница при неполадке. Это особенно полезно для отладки неполадок непрерывной интеграции, но для тестов также очень полезно то, что вы работаете локально.

Будьте готовы увеличить свои тайм-ауты

CI-сервер не может быть таким же быстрым, как ваш ноутбук, особенно если он находится под нагрузкой, одновременно выполняя многочисленные тесты. Будьте еще более щедрыми с вашими тайм-аутами, чтобы свести к минимуму риск возникновения внезапных неполадок.

Изучите организацию совместной работы CI-сервера и промежуточного сервера

Тесты, которые используют `LiveServerTestCase`, очень хорошо подходят для серверов разработки, но подлинную гарантию дает выполнение тестов относительно реального сервера. Лучше изучите возможность развертывания CI-сервера на промежуточном и выполнения функциональных тестов относительно него. Это имеет дополнительное преимущество: так вы тестируете сценарии автоматизированного развертывания.

Глава 25

Социально зачимый кусок, шаблон проектирования «Страница» и упражнение для читателя

Разве шутки о том, что нынче все должно быть социальным, не начали обрастать бородой? В любом случае теперь все должно быть списками, A/B-протестированными на основе больших данных и с большой кликабельностью, из 10 предметов, о которых вот этот вдохновляющий наставник сказал, что они заставят вас поменять свое мнение по поводу ля-ля тополя… Списками – вдохновляющими или любыми другими – нередко обмениваются. Давайте предоставим пользователям возможность делиться своими списками с другими пользователями.

По пути мы усовершенствуем наши ФТ, начав реализовывать то, что называется шаблоном проектирования «Страничный объект» (Page Object).

Я не буду в явном виде демонстрировать, что нужно делать, а позволю вам самостоятельно написать модульные тесты и прикладной код приложения. Не волнуйтесь, вы не будете брошены на произвол судьбы! Я дам общую схему шагов, а также несколько полезных советов и подсказок.

ФТ с многочисленными пользователями и addCleanup

Приступим. Для этого ФТ нам понадобятся два пользователя:

functional_tests/test_sharing.py (ch22l001)

```
from selenium import webdriver
from .base import FunctionalTest

def quit_if_possible(browser):
```

```
    try: browser.quit()
    except: pass

class SharingTest(FunctionalTest):
    '''тест обмена данными'''

    def test_can_share_a_list_with_another_user(self):
        '''тест: можно обмениваться списком с еще одним пользователем'''
        # Эдит является зарегистрированным пользователем
        self.create_pre_authenticated_session('edith@example.com')
        edith_browser = self.browser
        self.addCleanup(lambda: quit_if_possible(edith_browser))

        # Ее друг Анцифер тоже зависает на сайте списков
        oni_browser = webdriver.Firefox()
        self.addCleanup(lambda: quit_if_possible(oni_browser))
        self.browser = oni_browser
        self.create_pre_authenticated_session('oniciferous@example.com')

        # Эдит открывает домашнюю страницу и начинает новый список
        self.browser = edith_browser
        self.browser.get(self.live_server_url)
        self.add_list_item('Get help')

        # Она замечает опцию "Поделиться этим списком"
        share_box = self.browser.find_element_by_css_selector(
            'input[name=”sharee”]'
        )
        self.assertEqual(
            share_box.get_attribute('placeholder'),
            'your-friend@example.com'
        )
```

В этом разделе стоит обратить внимание на интересную особенность. Это функция `addCleanup`, описание которой можно найти в документации по ссылке[1]. Ее можно применять в качестве альтернативы функции `tearDown` для очистки ресурсов, используемых во время теста. Она наиболее полезна, когда ресурс выделяется по ходу выполнения теста и поэтому вам не приходится тратить время в `tearDown`, придумывая, что делать или не делать во время очистки.

Функция `addCleanup` выполняется после `tearDown`, и именно поэтому нам нужен блок `try/except` для `quit_if_possible` (для выхода, если надо). Когда любой из браузеров, `edith_browser` или `oni_browser`, также присваивает-

[1] См. https://docs.python.org/3/library/unittest.html%23unittest.TestCase.addCleanup

ся свойству `self.browser` в точке, в которой тест заканчивается, функция `tearDown` уже будет завершена.

Нам также нужно переместить функцию создания предварительно аутентифицированного сеанса `create_pre_authenticated_session` из *test_my_lists.py* в *base.py*.

Ладно, давайте убедимся, что это работает:

```
$ python manage.py test functional_tests.test_sharing
[...]
Traceback (most recent call last):
  File "/.../superlists/functional_tests/test_sharing.py", line 31, in
test_can_share_a_list_with_another_user
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: input[name="sharee"]
```

Великолепно! Кажется, он успешно прошел создание двух пользовательских сеансов и добрался до ожидаемой неполадки – на странице нет поля ввода для адреса электронной почты человека, с которым пользователь будет обмениваться списками.

Давайте сделаем фиксацию в этой точке, потому что у нас по крайней мере есть заготовка ФТ, полезная модификация функции `create_pre_authenticated_session`, и мы собираемся предпринять небольшую рефакторизацию ФТ:

```
$ git add functional_tests
$ git commit -m "Новый ФТ для обмена, перемещено создание сеансов в базу"
```

Страничный шаблон проектирования

Прежде чем идти дальше, хочу показать альтернативный метод сокращения дублирования кода в ФТ, который назвается «Страничные объекты» (Page Objects)[2].

Мы уже сконструировали несколько вспомогательных методов для ФТ, включая `add_list_item`, который мы здесь применили, но, если просто продолжить добавлять все больше и больше методов, он сильно переполнится. Я имел опыт работы с базовым классом ФТ, который состоял более чем из 1500 строк кода. Надо сказать, он показался мне довольно громоздким.

Страничные объекты являются альтернативой, которая поощряет нас хранить всю информацию и вспомогательные методы о различных типах страниц на нашем сайте в одном месте. Давайте посмотрим, как это могло

[2] См. http://www.seleniumhq.org/docs/06_test_design_considerations.jsp%23page-object-design-pattern

бы выглядеть для нашего сайта, начиная с класса, который будет представлять страницу любого списка:

functional_tests/list_page.py

```
from selenium.webdriver.common.keys import Keys
from .base import wait

class ListPage(object):
    '''страница списка'''

    def __init__(self, test):
        self.test = test  ❶

    def get_table_rows(self):  ❸
        '''получить строки таблицы'''
        return self.test.browser.find_elements_by_css_selector('#id_list_table tr')

    @wait
    def wait_for_row_in_list_table(self, item_text, item_number):  ❷
        row_text = '{}: {}'.format(item_number, item_text)
        rows = self.get_table_rows()
        self.test.assertIn(row_text, [row.text for row in rows])

    def get_item_input_box(self):  ❷
        '''получить поле ввода для элемента'''
        return self.test.browser.find_element_by_id('id_text')

    def add_list_item(self, item_text):  ❷
        '''добавить элемент списка'''
        new_item_no = len(self.get_table_rows()) + 1
        self.get_item_input_box().send_keys(item_text)
        self.get_item_input_box().send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table(item_text, new_item_no)
        return self  ❹
```

❶ Он инициализируется объектом, который представляет текущий тест. Это дает нам способность делать утверждения, получать доступ к экземпляру браузера через `self.test.browser` и применять функцию `self.test.wait_for`.

❷ Я скопировал несколько существующих вспомогательных методов из *base.py* и чуть-чуть их подправил…

❸ Например, они используют этот новый метод.

❹ Возвращение свойства `self` – это просто удобство. Оно позволяет выстраивать методы в цепочку[3], что мы сейчас увидим в действии.

[3] См. https://en.wikipedia.org/wiki/Method_chaining

functional_tests/test_sharing.py (ch22l004)

```
from .list_page import ListPage
[...]

        # Эдит открывает домашнюю страницу и начинает новый список
        self.browser = edith_browser
        list_page = ListPage(self).add_list_item('Get help')
```

Давайте продолжим переписывать наш тест, используя страничный объект всякий раз, когда мы хотим получать доступ к элементам из страницы списков:

functional_tests/test_sharing.py (ch22l008)

```
        # Она замечает опцию "Поделиться этим списком"
        share_box = list_page.get_share_box()
        self.assertEqual(
            share_box.get_attribute('placeholder'),
            'your-friend@example.com'
        )

        # Она делится своим списком.
        # Страница обновляется и сообщает, что
          теперь страница используется совместо с Анцифером:
        list_page.share_list_with('oniciferous@example.com')
```

В `ListPage` мы добавим следующие три функции:

functional_tests/list_page.py (ch22l009)

```
    def get_share_box(self):
        '''получить поле для обмена списками'''
        return self.test.browser.find_element_by_css_selector(
            'input[name="sharee"]'
        )

    def get_shared_with_list(self):
        '''получить список от того, кто им делится'''
        return self.test.browser.find_elements_by_css_selector(
            '.list-sharee'
        )

    def share_list_with(self, email):
        '''поделиться списком с'''
        self.get_share_box().send_keys(email)
        self.get_share_box().send_keys(Keys.ENTER)
        self.test.wait_for(lambda: self.test.assertIn(
```

```
            email,
            [item.text for item in self.get_shared_with_list()]
        ))
```

В основе страничного шаблона проектирования Page pattern лежит идея, что он должен получать всю информацию об определенной странице на вашем сайте таким образом, чтобы, если позже вы захотите внести в эту страницу изменения (даже простые исправления в ее HTML-компоновке), у вас будет единственное место, куда можно обратиться, чтобы внести корректировку в функциональные тесты, а не копаться в десятках ФТ.

Следующий шаг – рефакторизация ФТ во всех остальных тестах. Я не буду их здесь показывать. Вы можете попрактиковаться самостоятельно, чтобы почувствовать, как выглядят компромиссы между подходом DRY и удобочитаемостью теста…

Расширение ФТ до второго пользователя и страница «Мои списки»

Давайте конкретизируем, какой мы хотим иметь историю пользователя, который делится информацией. Эдит на своей странице увидела, что теперь ее список «предоставлен Анциферу для совместного пользования». Мы можем иметь ситуацию, когда Анци заходит на сайт и на своей странице видит список «Мои списки» – возможно, в разделе «Списки для совместного пользования»:

functional_tests/test_sharing.py (ch22l010)

```
from .my_lists_page import MyListsPage
[...]

        list_page.share_list_with('oniciferous@example.com')

        # Анцифер переходит на страницу списков в своем браузере
        self.browser = oni_browser
        MyListsPage(self).go_to_my_lists_page()

        # Он видит на ней список Эдит!
        self.browser.find_element_by_link_text('Get help').click()
```

Это означает еще одну функцию в нашем классе `MyListsPage`:

functional_tests/my_lists_page.py (ch22l011)

```
class MyListsPage(object):

    def __init__(self, test):
```

```
        self.test = test

    def go_to_my_lists_page(self):
        '''перейти на мою страницу списков'''
        self.test.browser.get(self.test.live_server_url)
        self.test.browser.find_element_by_link_text('My lists').click()
        self.test.wait_for(lambda: self.test.assertEqual(
        self.test.browser.find_element_by_tag_name('h1').text,
            'My Lists'
        ))
        return self
```

Опять-таки, эту функцию неплохо было бы перенести в *test_my_lists.py*, возможно, вместе собъектом `MyListsPage`.

Между делом Анцифер может также что-то добавить в список:

functional_tests/test_sharing.py (ch22l012)

```
    # На странице, которую Анцифер видит, говорится, что это список Эдит
    self.wait_for(lambda: self.assertEqual(
        list_page.get_list_owner(),
        'edith@example.com'
    ))

    # Он добавляет элемент в список
    list_page.add_list_item('Hi Edith!')

    # Когда Эдит обновляет страницу, она видит дополнение Анцифера
    self.browser = edith_browser
    self.browser.refresh()
    list_page.wait_for_row_in_list_table('Hi Edith!', 2)
```

Это еще одно дополнение к объекту `ListPage`:

functional_tests/list_page.py (ch22l013)

```
class ListPage(object):
    [...]

    def get_list_owner(self):
        '''получить владельца списка'''
        return self.test.browser.find_element_by_id('id_list_owner').text
```

Уже давно пора выполнить ФТ и проверить, все ли работает!

```
$ python manage.py test functional_tests.test_sharing

    share_box = list_page.get_share_box()
```

```
    [...]
selenium.common.exceptions.NoSuchElementException: Message: Невозможно
локализовать элемент: input[name="sharee"]
```

Это ожидаемая неполадка; у нас нет поля ввода для адресов электронной почты людей, с которыми пользователи будут делиться. Давайте выполним фиксацию:

```
$ git add functional_tests
$ git commit -m "Созданы страничные объекты для страниц списков, использовать
в ФТ с обменом списками»
```

Упражнение для читателя

Я действительно не до конца понимал, что делал, пока не завершил упражнение для читателя в главе 21.

— Йейн Г. (читатель)

Нет ничего, что цементировало бы обучение лучше, чем снятие учебных колес и получение чего-то работающего пешим ходом. Поэтому я надеюсь, что это вы попробуете сами.

Вот общая схема, что можно предпринять:

1. Прежде всего нам будет нужен новый раздел в *list.html*, форма с полем ввода для адреса электронной почты. Это продвинет ФТ на один шаг вперед.
2. Далее потребуется представление для формы, куда отправлять информацию. Начните с определения URL-адреса в шаблоне, что-то вроде *lists/<id_списка>/share*.
3. Затем наступает очередь первого модульного теста. Вполне может хватить представления-заготовки. Мы хотим, чтобы представление отвечало на POST-запросы, и оно должно откликаться переадресацией на страницу списка, поэтому тест мог бы вызваться как-то так: `ShareListTest.test_post_redirects_to_lists_page` (тест: POST-запрос переадресуется на страницу списка).
4. Мы выстраиваем наше представление-заготовку как двустрочную функцию, которая находит список и к нему переадресует.
5. Затем можно написать новый модульный тест, который создает пользователя и список, выполняет POST-запрос с его адресом электронной почты и проверяет, что пользователь добавлен в `list_.shared_with.all()` (что-то похожее на применение ORM к «Моим спискам»). Упомянутый атрибут `shared_with` еще не будет существовать, так как мы движемся снаружи внутрь.

6. Таким образом, прежде чем мы сможем привести этот тест к успешному прохождению, мы должны спуститься на уровень модели. Следующий тест в *test_models.py* проверяет, что список имеет метод `shared_with.add`, который можно вызывать с адресом электронной почты пользователя, и затем проверить результат вызова `shared_with.all()`, который впоследствии будет содержать этого пользователя.
7. Затем потребуется поле «многие-ко-многим» `ManyToManyField`. Вы, вероятно, увидите сообщение об ошибке, связанной с конфликтом имен `related_name`. Решение которой есть в документации Django.
8. Оно потребует выполнить миграцию базы данных.
9. Это приведет тесты модели к успешному прохождению. Вернитесь на уровень выше, чтобы зафиксировать тест представления.
10. Вы можете обнаружить, что представление для переадресации не работает – не отправляет допустимый POST-запрос. Выберите одно из двух: проигнорировать недопустимый ввод или скорректировать тест, чтобы отправлялся допустимый POST-запрос.
11. Затем вернитесь к уровню шаблона. На странице «Мои списки» нам нужен элемент `<ul>` с циклом `for` по спискам, предоставленным для общего пользования. На странице списков мы также хотим показать, кто может пользоваться конкретным списком и кто является его владельцем. Обратитесь к ФТ по поводу использования правильных классов и идентификаторов. Если хотите, по каждому из них вы тоже можете создать короткие модульные тесты.
12. Запуск сайта командой `runserver` поможет вам сгладить какие-то дефекты, а также донастроить макет и эстетику. Если вы используете приватный сеанс браузера, вы сможете зарегистрировать многочисленных пользователей.

К концу работы у вас получится что-то подобное (рис. 25.1):

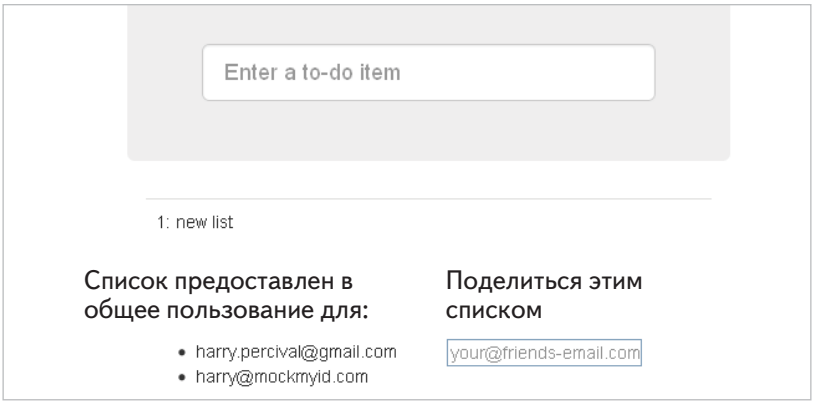


Рис. 25.1. Обмен списками

Страничный шаблон проектирования и реальное упражнение для читателя

Применяйте к своим функциональным тестам принцип DRY

Как только ваш комплект ФТ начинает расти, вы заметите, что различные тесты используют похожие части графического интерфейса. Постарайтесь избегать констант, таких как идентификаторы HTML или классы определенных элементов графического интерфейса, которые дублируются между вашими ФТ.

Страничный шаблон проектирования

Перемещение вспомогательных методов в базовый класс `FunctionalTest` делает его громоздким. Используйте отдельные страничные объекты Page, чтобы в них содержать всю логику работы с определенными частями сайта.

Упражнение для читателя

Надеюсь, вы действительно попробовали его выполнить! Попытайтесь следовать принципу «снаружи внутрь» и временами работайте вручную, если зашли в тупик. Разумеется, реальным упражнением для читателя будет применение методологии TDD к своему следующему проекту. Надеюсь, вы получите удовольствие от работы!

В следующей главе мы завершим обсуждение тестирования «лучших приемов».

Глава 26

Быстрые тесты, медленные тесты и горячий пол

База данных – это игра в горячий пол![1]
— Кэйси Кинси

Вплоть до главы 23 почти все модульные тесты, приведенные в этой книге, нужно было называть *интегрированными*, потому что они либо опираются на базу данных, либо используют тестовый клиент Django, делающий слишком много волшебных вещей на межплатформенном уровне, которые находятся между запросами, откликами и функциями представлений данных.

Существует мнение, что истинный модульный тест всегда должен быть изолированным, потому что он предназначен для тестирования конкретной единицы программного обеспечения.

Некоторые ветераны TDD говорят, что вместо написания интегрированных тестов следует всегда стремиться писать «чистые» изолированные модульные тесты, если есть возможность. Это одна из непрекращающихся (иногда оживленных) дискуссий в сообществе разработчиков, занимающихся тестированием программного обеспечения.

Являясь всего лишь неотесанным сорванцом, я только наполовину погрузился во все тонкости данного спора. Но в этой главе попытаюсь поговорить о том, почему люди настроены столь решительно, и постараюсь дать некоторое представление о том, когда можно обойтись без неразберихи, связанной с интегрированными тестами (которых, признаюсь, я делаю очень много!), и когда стоит стремиться к более чистым модульным тестам.

1 Горячий пол, или раскаленная лава, – это миленькая классическая детская игра, в которой нельзя касаться пола, иначе вы мертвы. – Прим. перев.

Терминология: различные типы тестов

Изолированные тесты (чистые модульные тесты) против интегрированных

Основная цель модульного теста – проверять правильность логики приложения. Изолированный тест проверяет строго один фрагмент кода, его успешность или неуспешность не зависит от какого-то другого внешнего кода. Это то, что я называю чистым модульным тестом: тест для единственной функции, например, написанный так, что только эта конкретная функция может заставить его не сработать. Если функция зависит от другой системы и нарушение этой системы повреждает наш тест, значит, мы имеем интегрированный тест. Такая система может быть внешней, например база данных, но она также может быть и еще одной функцией, которой мы не управляем. В любом случае, если повреждение системы делает наш тест нерабочим, значит, он не изолирован должным образом и не является чистым модульным тестом. Это не всегда плохо, но может означать, что тест выполняет сразу две задачи.

Интеграционные тесты

Интеграционный (комплексный) тест проверяет, что программный код, которым вы управляете, правильно интегрирован в некоторую внешнюю систему, которой вы не управляете. Обычно интеграционные тесты также являются интегрированными.

Системные тесты

Если интеграционный тест проверяет интеграцию в одну внешнюю систему, то системный проверяет интеграцию многочисленных систем в ваше приложение. Например, проверка организации налаженного взаимодействия между базой данных, статическими файлами и конфигурацией сервера.

Функциональные тесты и приемочные тесты

Приемочный тест (или приемочное испытание) предназначается для тестирования того, работает ли наша система с точки зрения пользователя («принял бы пользователь такое ее поведение?»). Очень трудно написать приемочный тест, который не является полностековым, сквозным (от начала до конца). Функциональные тесты, которые мы использовали, играли роль как приемочных, так и системных тестов.

Если вы простите меня за претенциозную философскую терминологию, я собираюсь придерживаться гегельянской диалектической структуры:

- *Тезис*: аргумент в пользу чистых модульных тестов – они быстрые.
- *Антитезис*: некоторые риски связаны с наивным подходом на основе чистого модульного тестирования.
- *Синтез*: обсуждение лучших приемов, к примеру метода «Порты и адаптеры» или «Функциональное ядро, императивная оболочка», и того, что мы хотим от наших тестов.

Тезис: модульные тесты сверхбыстры и к тому же хороши

Один из аргументов, которые вы часто слышите в пользу модульных тестов, состоит в том, что они намного быстрее. Не думаю, что быстродействие на самом деле является первоочередным преимуществом модульных тестов, но эту тему стоит исследовать.

Более быстрые тесты означают более быструю разработку

При прочих равных условиях: чем быстрее выполняется ваш модульный тест, тем лучше. И в меньшей степени: чем быстрее выполняются все ваши тесты, тем лучше.

В этой книге я в общих чертах обрисовал цикл методологии TDD, выполняемый по схеме «тест/программный код». Вы начали получать представление о потоке операций TDD, о том, как вы носитесь туда-сюда между написанием крошечных фрагментов кода и выполнением тестов. И в итоге вы приходите к тому, что выполняете модульные тесты несколько раз в минуту и функциональные тесты – несколько раз в день.

Поэтому (на очень элементарном уровне) чем больше времени требуется на их выполнение, тем больше времени вы тратите на ожидание, когда завершатся ваши тесты, и это сильно замедляет процесс разработки. Но есть еще кое-что…

Священное состояние потока

Если рассуждать с социологической точки зрения, мы, программисты, имеем свою собственную культуру и в некотором роде собственную племенную религию. В ней есть много общин, таких как культ TDD, в котором вы теперь инициированы. Существуют последователи редактора vi и еретики emacs. Но вот в чем мы едины, так это в одной конкретной духовной практике, нашей собственной трансцендентальной медитации, которой является священное состояние потока. То чувство чистого сосредоточия, концентрации, где часы проходят, как никакое время вообще, где исходный код естественным образом вытекает из-под наших пальцев, где проблемы коварны лишь настолько, чтобы быть интересными, но не настолько трудны, чтобы они одерживали над нами верх…

Нет абсолютно никакой надежды на достижение потока, если вы тратите свое время на ожидание, когда медленный комплект тестов выполнится. Все, что дольше нескольких секунд, заставляет ваше внимание блуждать, вы контекстно переключаетесь, и состояние потока исчезает. И тогда состояние потока – просто хрупкая мечта. Как только оно пропало, требуется по крайней мере минут 15, чтобы зажить снова.

Медленные тесты не выполняются часто, что приводит к плохому коду

Если тесты медленные и нарушают вашу концентрацию, то опасность кроется в том, что вы начнете избегать их выполнения, а это может привести к проникновению дефектов. Или же вы будете чувствовать робость перед рефакторизацией кода, поскольку мы знаем, что любая рефакторизация будет означать необходимость ждать столетия, пока все тесты не завершатся. В любом случае результатом будет плохой код.

Теперь у нас все в порядке, но интегрированные тесты со временем становятся медленнее

Вы можете подумать: «Да ладно, наш комплект тестов содержит большое количество интегрированных тестов – более 50, и нужно всего 0,2 секунды на их выполнение».

Однако не забывайте, что у нас очень простое приложение. Как только оно становится сложнее, а база данных наращивает все больше и больше таблиц и столбцов, интегрированные тесты будут все медленнее и медленнее. А обнуление базы данных при помощи Django между каждым тестом будет занимать все больше и больше времени.

Не верьте мне

Гэри Бернхард, человек с намного большим опытом тестирования, чем у меня, красноречиво изложил эти аргументы в интервью «Быстрый тест, медленный тест»[2]. Советую прочесть.

И модульные тесты управляют хорошей структурой кода

Но, пожалуй, важнее всего вышеперечисленного – помнить урок из главы 23. Опыт написания хороших изолированных модульных тестов помогает вычленять более оптимальную структуру программного кода, заставляя нас идентифицировать зависимости и побуждая следовать раздельной архитектуре путем, которым не позволяют идти интегрированные тесты.

Проблемы с «чистыми» модульными тестами

Все это сопровождается огромным «но». Написание изолированных объединенных тестов таит собственные опасности, особенно для тех, кто, подобно мне, еще не является профессионалом в методологии TDD.

Изолированные тесты труднее читать и писать

Мысленно вернитесь к первому изолированному модульному тесту, который мы написали. Разве он не был уродлив? Надо признать, что все

[2] См. https://www.youtube.com/watch?v=RAxiiRPHS9k

улучшилось, когда мы реструктурировали код в формы. Но вообразите, что было бы, если бы мы выполнили рефакторизацию? Мы бы остались с довольно нечитаемым тестом в кодовой базе. Но и финальная версия тестов, к которой мы в итоге пришли, содержит несколько довольно заумных фрагментов.

Изолированные тесты не тестируют интеграцию автоматически

Изолированные тесты тестируют только рассматриваемую единицу кода. Они не тестируют интеграцию между единицами программного кода.

Это общеизвестная проблема, и есть способы ее смягчения. Но, как мы видели, это смягчение требует достаточно сложной работы программиста: вам приходится все время отслеживать интерфейсы между единицами кода, идентифицировать неявный контракт, который каждая единица должна соблюдать, и для этих контрактов вам нужно писать тесты, впрочем, как и для внутренней функциональности единицы программного кода.

Модульные тесты редко отлавливают неожиданные дефекты

Модульные тесты помогут обнаружить ошибки неучтенных единиц кода и логическую путаницу (один из дефектов, который мы вносим все время, поэтому в некотором роде ожидаем таких ошибок). Но они не предупреждают о более неожиданных дефектах. Модульные тесты не напомнят, что вы забыли создать миграцию базы данных. Не скажут, когда межплатформенный уровень выполняет умное экранирование символов-мнемоник в HTML, которое влияет на вывод данных в качестве HTML… Это что-то вроде «неизвестных не известных» Дональда Рамсфельда?[3]

Тесты с имитациями могут стать близко привязанными к реализации

Наконец, тесты с имитациями могут стать очень плотно привязанными к реализации. Если для создания объектов вы решите использовать `List.objects.create()`, но ваши имитации ожидают использования `List()` и `.save()`, то вы получите неработающие тесты, несмотря на то что фактический эффект кода будет одним и тем же. Если вы не проявите осторожность, это начнет работать против одного из предполагаемых преимуществ наличия тестов, которые должны способствовать реструктуризации программного кода (рефакторизации). Когда вы захотите поменять вну-

[3] См. https://en.wikipedia.org/wiki/There_are_known_knowns – *Прим. перев.*

тренний API, может оказаться, что вам нужно вносить изменения в десятки тестов с имитациями и тестов соблюдения контрактов.

Обратите внимание: это может стать еще большей проблемой, если вы имеете дело с API, которым не управляете. Возможно, вы помните извилистый путь, который мы преодолели, чтобы протестировать форму, имитируя два класса модели Django и используя функцию `side_effect` для проверки состояния мира. Если вы пишете код, который полностью находится под вашим контролем, скорее всего, вы будете конструировать внутренние API так, чтобы они были чище и требовали меньше зигзагов во время тестирования.

Но все эти проблемы могут быть преодолены

Защитники изоляции сейчас скажут, мол, все эти вещи можно облегчить – просто нужно лучше писать изолированные тесты и помнить про священное состояние потока. Священное состояние потока!

Так, где же мы находимся?

Синтез: что мы хотим от всех наших тестов?

Давайте поразмышляем о том, какие преимущества должны предоставлять тесты. Для чего вообще мы их пишем?

Правильность

Мы хотим, чтобы наше приложение было свободно от дефектов – низкоуровневых логических ошибок (например, ошибок неучтенных единиц кода) и высокоуровневых дефектов. То есть программное обеспечение в конечном счете должно показывать то, что нужно нашим пользователям. Мы хотим выяснить, вносим ли мы вообще какую-то регрессию, которая нарушает то, что раньше работало. И мы хотим это узнать это до того, как пользователи заметят, что что-то не работает. Мы ждем, что тесты скажут, что приложение работает правильно.

Чистый код, удобный в сопровождении

Мы хотим, чтобы наш код соблюдал правила YAGNI и DRY. Нам нужен код, который ясно выражает свои намерения, который разбит на разумные компоненты, имеющие четко определенную ответственность, и легко понятен. Мы ожидаем, что наши тесты вселят в нас уверенность выполнять рефакторизацию нашего приложения на постоянной основе, чтобы мы никогда не боялись улучшить его структуру. Также мы хотели бы, чтобы они активно помогали нам найти правильную структуру кода.

Продуктивный поток операций

Наконец, мы хотим, чтобы наши тесты помогали задействовать быстрый и продуктивный поток операций. Чтобы они помогали избавиться от части напряжения в процессе разработки, защищали нас от глупых ошибок. Мы хотим, чтобы они помогали нам оставаться в состоянии потока не только потому, что мы им наслаждаемся, но и потому, что это очень продуктивно. Мы хотим, чтобы наши тесты давали обратную связь о нашей работе как можно быстрее, чтобы мы могли испытывать новые идеи и быстро их развивать. И мы не хотим ощущать, что тесты являются скорее помехой, чем помощью, когда дело доходит до развития кодовой базы.

Оценивайте свои тесты относительно преимуществ, которые вы хотите от них получить

Не думаю, что есть какие-то универсальные правила о том, сколько тестов необходимо написать и какой должен быть корректный баланс между функциональными, интегрированными и изолированными тестами. Обстоятельства разнятся от проекта к проекту. Но, размышляя обо всех тестах и задаваясь вопросом, предоставляют ли они преимущества, которых вы от них ждете, можно принять некоторые решения.

Таблица 26.1. Каким образом разные типы тестов помогают достигать целевые задачи?

Целевая задача	Некоторые соображения
Правильность	* Есть ли у меня достаточно функциональных тестов, чтобы убедиться, что мое приложение работает с точки зрения пользователя? * Тестирую ли я все граничные случаи полностью? По всей видимости, эта работа подходит для низкоуровневых, изолированных тестов. * Есть ли у меня тесты, которые как следует проверяют, подходят ли все компоненты приложения друг другу? С этим смогут справиться несколько интегрированных тестов или же достаточно функциональных тестов?
Чистый и удобный в сопровождении программный код	* Позволяют ли мои тесты выполнять рефакторизацию кода без боязни и часто? * Помогают ли тесты вычленять приемлемую структуру кода? Если у меня много интегрированных тестов и несколько изолированных, есть ли в моем приложении какие-либо части, где дополнительные усилия, потраченные на написание новых изолированных тестов, дадут более оптимальную обратную связь о структуре моего кода?

Целевая задача	Некоторые соображения
Продуктивный поток операций	* Являются ли циклы обратной связи такими быстрыми, как я и хотел? Когда я получаю предупреждения о дефектах? Есть ли какой-то прием, чтобы это происходило раньше? * Если у меня много высокоуровневых функциональных тестов, выполнение которых занимает много времени, и я провожу всю ночь в ожидании обратной связи о случайных регрессиях, есть ли способ писать более быстрые, возможно, интегрированные тесты, которые будут предоставлять мне более быструю обратную связь? * Могу ли я выполнить подмножество полного комплекта тестов, когда мне это нужно? * Провожу ли я слишком много времени в ожидании завершения работы тестов, и, следовательно, меньше времени – в продуктивном состоянии потока?

Архитектурные решения

Существуют также некоторые архитектурные решения, которые помогают получить максимум из вашего комплекта тестов и избежать некоторых недостатков изолированных тестов.

Главным образом они связаны с попыткой идентифицировать границы вашей системы – точки, в которых код взаимодействует с внешними системами (базы данных, файловая система, Интернет или пользовательский интерфейс) и попыткой отделить их от профильной бизнес-логики вашего приложения.

Порты и адаптеры/шестиугольная/чистая архитектура

Интегрированные тесты проявляют свою ключевую полезность на границах системы. В этих точках негативные аспекты изоляции тестов и имитаций проявляются сильнее всего, потому что именно на границах вы, скорее всего, будете раздосадованы, если окажется, что ваши тесты плотно соединены с реализацией, или захотите убедиться, что все интегрировано как надо.

С другой стороны, код в *ядре* нашего приложения касается исключительно нашей предметной области бизнеса и бизнес-правил. Он находится полностью под нашим контролем и испытывает меньшую потребность в интегрированных тестах, поскольку мы его весь контролируем и понимаем.

Таким образом, один из способов добиться желаемого – попытаться минимизировать объем программного кода, который должен иметь дело с границами. Затем мы тестируем нашу профильную бизнес-логику с помощью изолированных тестов и тестируем точки интеграции с помощью интегрированных тестов.

Стив Фримэн и Нэт Прайс в своей книге «*Наращивание объектно-ориентированных ПО под руководством тестов*» (Growing Object-Oriented Software, Guided By Tests) называют такой подход «Порты и адаптеры» (см. рис. 26.1).

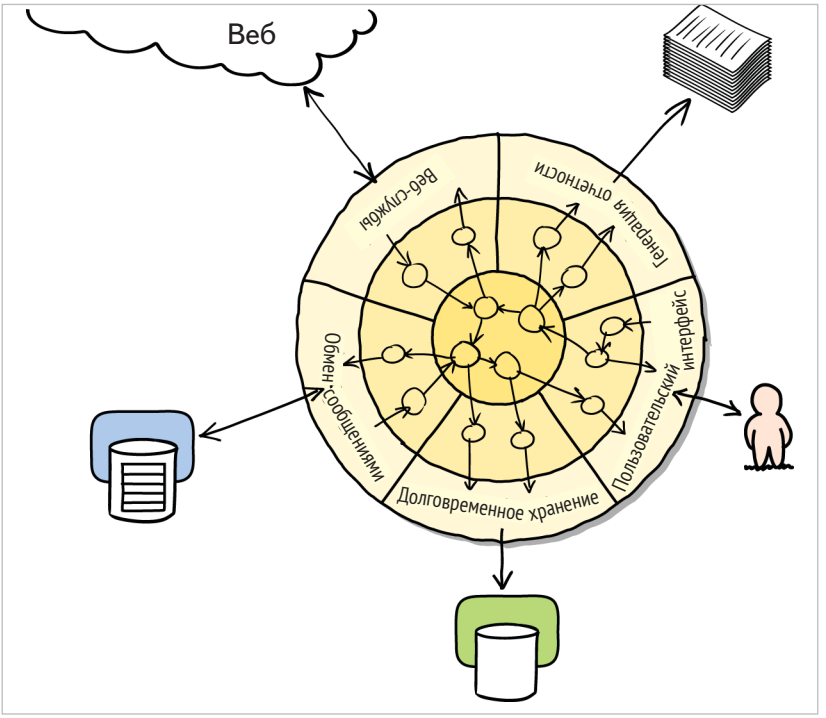


Рис. 26.1. Порты и адаптеры (диаграмма Нэта Прайса)

Фактически мы начали двигаться в сторону архитектуры портов и адаптеров в главе 23, когда обнаружили, что написание изолированных модульных тестов способствует выталкиванию из главного приложения программного кода, связанного с ORM, и его сокрытию во вспомогательных функциях из уровня модели.

Такой шаблон проектирования иногда называют чистой или шестиугольной архитектурой. Более подробную информацию вы можете найти в разделе дополнительных материалов для чтения в конце главы.

Функциональное ядро, императивная оболочка

Гэри Бернхард развивает эту идею и рекомендует архитектуру, которую он называет функциональным ядром, императивной оболочкой, в рамках которой «оболочка» приложения, место, где происходит взаимодействие с границами, повинуется парадигме императивного программирования и может тестироваться интегрированными тестами, приемочными тестами или даже (дух захватывает!) не тестироваться вообще, если оно остается

достаточно минимальным. Но ядро приложения пишется, подчиняясь парадигме функционального программирования (в комплекте с сопутствующим отсутствием побочных эффектов), которое фактически допускает полностью изолированные чистые модульные тесты, абсолютно без имитаций.

Обратитесь к презентации Гэри под заголовком «Границы»[4], чтобы получить дополнительную информацию об этом подходе.

Заключение

Я попытался сделать обзор некоторых более продвинутых соображений, относящихся к процессу TDD. Освоение этих тем приходит с годами практической деятельности, следовательно, я не обладаю достаточной квалификацией вести разговор об этих вещах. Поэтому я настоятельно рекомендую вам принимать все, что я сказал, с некоторым недоверием, попробовать выяснить, что работает именно для вас, и – самое главное – ознакомиться с мнениями настоящих экспертов!

Вот куда можно направить свои стопы за дополнительными материалами для чтения.

Дополнительные материалы для чтения

Быстрый тест, медленный тест и границы

Беседы Гэри Бернхарда на конференции Pycon 2012 и 2013. Также стоит посмотреть его экранные демонстрации (см., соответственно, https://www.youtube.com/watch?v=RAxiiRPHS9k и http://www.destroyallsoftware.com/).

Порты и адаптеры

Стив Фримэн и Нэт Прайс написали об этом методе упомянутую выше книгу (см. http://vimeo.com/83960706). Вы попадете на неплохое обсуждение указанной идеи в беседе по ссылке http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html. Смотрите также описание чистой архитектуры дядей Бобом и статью «Алистер Кокберн вводит термин «шестиугольная архитектура» (см. http://alistair.cockburn.us/Hexagonal+architecture).

Горячий пол

Незабываемое предупреждение Кэйси Кинси о том, что следует избегать базы данных всегда, когда это возможно (см. https://www.youtube.com/watch?v=bsmFVb8guMU).

Инвертирование пирамиды

[4] См. https://www.youtube.com/watch?v=e0Yal8elnZk

Идея, что проекты в конечном счете показывают слишком большое соотношением медленных, высокоуровневых тестов к модульным тестам и является визуальной метафорой для усилий с целью инвертировать это соотношение (см. http://watirmelon.com/tag/testing-pyramid/).

Интегрированные тесты являются жульничеством

Знаменитую тираду Дж. Б. Рейнсбергера о том, как интегрированные тесты разрушат вашу жизнь, можно найти по ссылке http://blog.thecodewhisperer.com/2010/10/16/integrated-tests-are-a-scam/. Видеопрезентацию можно найти по ссылке http://www.jbrains.ca/permalink/using-integration-tests-mindfully-a-case-study (имеется два видеоролика, но ни один не может похвастаться совершенной кинематографией). Гляньте пару ответных постов, особенно это выступление в защиту приемочных тестов (которые я называю функциональными) http://www.infoq.com/presentations/integration-tests-scam и http://vimeo.com/80533536 и анализ того, как медленные тесты убивают производительность http://www.jbrains.ca/permalink/part-2-some-hidden-costs-of-integration-tests.

Wiki-справочник по тестированию Test-Double (Тестовый дублер)

Онлайн-ресурс Джастина Серлза – великолепный источник определений и аргументов «за» и «против» тестирования. Серлз приходит к собственным выводам относительно верного способа действий: wiki-справочник по тестированию (см. https://github.com/testdouble/contributing-tests/wiki/Test-Driven-Development).

Прагматическое представление

Мартин Фаулер (автор книги о рефакторизации) представляет обоснованно сбалансированный, прагматический подход (см. http://martinfowler.com/bliki/UnitTest.html).

О достижении правильного баланса между разными типами тестов

Будьте прагматичными

Тратить кучу времени на размышления о том, какие виды тестов писать, – отличный способ отклониться от истины. Лучше начать с написания любого типа тестов, который придет вам в голову первым, и изменить его позже, если потребуется. Учитесь на практике.

Сосредоточьтесь на том, что именно вы хотите получить от тестов

Ваши целевые задачи – правильность, хорошая структура кода и быстрые циклы обратной связи. Разные типы тестов в разной мере помогут достичь каждую из этих задач. Таблица 26.1 содержит несколько хороших вопросов, на которые вы должны ответить самому себе.

Архитектура имеет значение

Архитектура в известной мере диктует типы тестов, которые вам нужны. Чем точнее вы сможете отделить бизнес-логику от внешних зависимостей и чем более модульным будет программный код, тем ближе вы станете к корректному балансу модульных, интеграционных (комплексных) и сквозных тестов.

Повинуйтесь Билли-тестировщику!

Назад к Билли-тестировщику.

Слышу стон и ваш возглас: «Гарри! Билли-тестировщик перестал быть смешным уже глав 17 назад!». Потерпите, я собираюсь использовать этот персонаж, чтобы сделать серьезное заявление.

Тестировать очень тяжело

Мне кажется, причина, почему фраза «Повинуйся Билли-тестировщику» зацепила меня с самого начала, когда я ее увидел, в том, что она действительно отражает сложность тестирования. Причем процесс не настолько труден сам по себе, насколько трудно его придерживаться и продолжать.

Всегда есть подспудное желание для простоты срезать углы и пропустить несколько тестов. И вдвойне сложнее в психологическом плане, потому что награда за труды уж очень удалена от точки, в которой вы вкладываете свои усилия. Тест, на написание которого вы тратите время в данный момент, не приносит вознаграждение сразу же, он окажет помощь намного позже – возможно, несколько месяцев спустя, когда спасет вас от внесения дефекта при рефакторизации либо отловит регрессию, когда вы будете обновлять зависимости. Или же отплатит вам так, что его ценность будет сложно измерить, побуждая вас писать более структурированный код. Тем не менее вы убеждаете себя, что способны написать его столь же изящно и без тестов.

Я сам начал соскальзывать, когда писал тестовую инфраструктуру для этой книги[1]. Будучи довольно сложным животным, эта инфраструктура имеет собственные тесты, но я срезал несколько углов. В результате покрытие тестами получилось не вполне совершенным, и теперь я сожалею об этом, потому что она оказалась довольно громоздкой и уродливой (налетайте, я выложил ее в открытый доступ, так что можете показывать на меня пальцем и поднять на смех).

Держите свои сборки CI-сервера на зеленом уровне

Еще одна область, которая требует по-настоящему трудоемкой работы, – непрерывная интеграция. В главе 24 вы видели, что странные и непредсказуемые дефекты иногда происходят на CI-сервере. Когда вы на них смотрите и думаете, мол, это же прекрасно работает на моей машине, есть сильное искушение просто проигнорировать их… Но если вы не будете осторожны, то станете толерантными к неработающему комплекту тестов на CI-сервере и довольно скоро ваша сборка CI-сервера станет практичес-

[1] См. https://github.com/hjwp/Book-TDD-Web-Dev-Python/tree/master/tests

ки бесполезной. Не попадайтесь в эту ловушку. Будьте настойчивы, и вы найдете причину, почему ваш тест не работает, и способ ее устранить, сделав тест детерминированным и снова зеленым.

Гордитесь своими тестами так же, как своим программным кодом

Перестаньте думать о тестах как о несущественном дополнении к реальному коду. Воспринимайте их частью готового изделия, которое вы создаете, частью, которая должна полироваться столь же тщательно, должна быть столь же эстетичной и приятной на вид, и поставкой которой можно по праву гордиться…

Делайте это, потому что так говорит Билли-тестировщик. А также потому, что вознаграждение будет стоить того, даже если оно не будет немедленным. И просто из чувства долга или профессионализма, или навязчивого состояния, или чистой неуступчивости. Потому, что практиковаться всегда полезно. В конце концов, потому, что так разрабатывать программное обеспечение даже интересней.

Не забудьте дать на чай персоналу заведения

Этой книги не было бы без поддержки со стороны издательства, замечательного O'Reilly Media. Если вы читаете бесплатное издание онлайн, надеюсь, вы рассмотрите возможность покупки реальной копии… Если же она не нужна вам, то, может быть, в качестве подарка для друга?

Не пропадайте!

Надеюсь, вы получили удовольствие о чтения этой книги. Обязательно со мной свяжитесь и поделитесь своими мыслями!

Гарри

- https://twitter.com/hjwp
- obeythetestinggoat@gmail.com

Приложение A

PythonAnywhere

Эта книга исходит из предположения, что вы работаете с Python и программируете на своем собственном компьютере. Разумеется, сегодня это не единственный способ программировать на Python, вы можете использовать онлайн-платформу, подобную PythonAnywhere (где, к слову сказать, я работаю).

На PythonAnywhere можно выполнить все, что излагается в книге, но с несколькими поправками и изменениями. Вам придется установить веб-приложение вместо тестового сервера; использовать Xvfb для выполнения функциональных тестов; как только вы доберетесь до глав о развертывании, вам потребуется обновиться до платной учетной записи. Так что все возможно, но будет проще, если вы станете работать на собственном ПК.

С учетом этого, если вы по-прежнему нацелены попробовать, вот несколько подробностей о том, что необходимо сделать.

Если вы этого еще не сделали, необходимо зарегистрироваться в PythonAnywhere. Это может быть бесплатная учетная запись.

Затем на странице консолей запустите *консоль Bash*. Здесь мы выполняем бо́льшую часть работы.

Выполнение сеансов Firefox Selenium при помощи Xvfb

Первое, на что стоит обратить внимание: PythonAnywhere – исключительно консольная среда, в ней нет дисплея, чтобы открыть Firefox. Но мы можем использовать виртуальный дисплей.

В главе 1, когда мы пишем самый первый тест, вы обнаружите, что не все работает так, как ожидалось. Первый тест выглядит так, и вы можете его вполне нормально набрать, используя редактор PythonAnywhere:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://localhost:8000')
assert 'Django' in browser.title
```

Но когда вы попытаетесь его выполнить (в *консоли Bash*), получите ошибку:

```
(superlists)$ python functional_tests.py
Traceback (most recent call last):
File "tests.py", line 3, in <module>
browser = webdriver.Firefox()
[...]
selenium.common.exceptions.WebDriverException: Message: 'Браузер похоже
завершил работу до того, как мы смогли установить связь. В результате:
Ошибка: не указан дисплей\n'
```

Ввиду того, что PythonAnywhere прикреплен к более старой версии Firefox, нам практически не требуется Geckodrive, но нам действительно нужно перейти назад на Selenium 2 вместо Selenium 3:

```
(superlists) $ pip install "selenium<3"
Collecting selenium<3
Installing collected packages: selenium
  Found existing installation: selenium 3.4.3
    Uninstalling selenium-3.4.3:
        Successfully uninstalled selenium-3.4.3
Successfully installed selenium-2.53.6
```

Теперь мы сталкиваемся со второй проблемой:

```
(superlists)$ python functional_tests.py
Traceback (most recent call last):
File "tests.py", line 3, in <module>
browser = webdriver.Firefox()
[...]
selenium.common.exceptions.WebDriverException: Message: The browser
appears to have exited before we could connect. If you specified a log_file
in the FirefoxBinary constructor, check it for details.
```

Firefox не запускается, потому что нет дисплея, на котором ему выполняться, потому что PythonAnywhere – это серверная среда. Выходом является использование браузера *Xvfb*, название которого расшифровывается как X Virtual Framebuffer, или X виртуальный кадровый буфер. Он запустит виртуальный дисплей, который Firefox может использовать, даже если нет реального дисплея.

Команда `xvfb-run` выполнит следующую команду в Xvfb. Ее применение даст ожидаемую неполадку:

```
(superlists)$ xvfb-run python functional_tests.py
Traceback (most recent call last):
```

```
File "tests.py", line 11, in <module>
assert 'Django' in browser.title
AssertionError
```

Урок заключается в том, чтобы использовать команду `xvfb-run` всякий раз, когда требуется выполнить функциональные тесты.

Настройка Django как веб-приложение PythonAnywhere

Сразу после этого мы устанавливаем Django, используя команду `django-admin.py startproject`. Но вместо применения команды `manage.py runserver` для запуска локального сервера разработки мы установим наш сайт как реальное веб-приложение PythonAnywhere.

Перейдите к вкладке **Web** и нажмите кнопку, чтобы добавить новое веб-приложение. Выберите **Manual configuration** (Ручная конфигурация) и затем Python 3.6.

На следующем экране введите имя вашей виртуальной среды virtualenv (superlists), и когда вы его предоставите, оно должно автозаполниться до */home/yourusername/.virtualenvs/superlists*.

Наконец, перейдите по ссылке, чтобы *отредактировать ваш файл wsgi* и найти и раскомментировать раздел, связанный с Django. Нажмите **Save** (Сохранить) и затем **Reload** (Перезагрузить), чтобы обновить веб-приложение.

С этого момента вместо выполнения тестового сервера из консоли на `localhost:8000` вы можете использовать реальный URL-адрес вашего веб-приложения PythonAnywhere:

```
browser.get('http://my-username.pythonanywhere.com')
```

Следует нажимать на **Reload** (Перезагрузить) всякий раз, когда вы вносите изменения в программный код, чтобы обновлять сайт.

Это должно работать лучше[1]. Вам придется соблюдать эту схему: указывать функциональные тесты на версию сайта для PythonAnywhere и нажимать на Reload перед каждым выполнением ФТ вплоть до главы 7, когда мы перейдем на использование `LiveServerTestCase` и `self.live_server_url`.

[1] Вы *могли бы* выполнять сервер разработки Django из консоли, но проблема в том, что консоли PythonAnywhere не всегда выполняются на том же сервере, поэтому нет гарантии, что консоль, в которой вы выполняете свои тесты, будет той же, в которой работает сервер. Плюс, когда он выполняется в консоли, нет простого способа визуально проинспектировать сайт.

Очистка папки /tmp

Selenium и Xvfb часто оставляют много мусора в папке */tmp*, особенно когда они закрыты некорректно (вот почему ранее я включал блок `try/finally`).

Остается так много ненужных файлов, что они могут превысить квоту хранилища. Поэтому время от времени наводите порядок в */tmp*:

```
$ rm -rf /tmp/*
```

Снимки экрана

В главе 5 я предлагаю использовать оператор `time.sleep` для приостановки выполнения ФТ, чтобы посмотреть, что именно браузер Selenium показывает на экране. Мы не можем это сделать в PythonAnywhere, потому что браузер выполняется в режиме виртуального дисплея. Но вы можете осмотреть живой сайт либо верить мне на слово.

Лучший способ визуально контролировать тесты, которые выполняются в виртуальном дисплее, – использовать снимки экрана. Если вам интересно, обратитесь к главе 24 – там приводится небольшой пример.

Глава о развертывании

Когда вы перейдете к главе 9, у вас будет выбор продолжить использовать PythonAnywhere либо узнать, как создать реальный сервер. Я рекомендую последнее, так вы получите максимум из возможного.

Если вы действительно хотите придерживаться PythonAnywhere, что на самом деле неправда, можете подписаться на второй аккаунт PythonAnywhere и использовать его как промежуточный сайт. Или же добавьте в свою существующую учетную запись второй домен. Но тогда большинство инструкций в данной главе не будет иметь смысла (в PythonAnywhere нет необходимости в nginx или gunicorn или доменных сокетах).

Так или иначе, в этой точке вам, вероятно, потребуется платная учетная запись:

- Если вы хотите выполнять свой промежуточный сайт на домене, не связанном с PythonAnywhere.
- Если вы хотите иметь возможность выполнять функциональные тесты относительно домена, не связанного с PythonAnywhere (потому что его не будет в нашем белом списке).
- Если вы хотите выполнять fabric относительно учетной записи PythonAnywhere (потому что вам нужен SSH), когда доберетесь до главы 11.

Если вы захотите пойти обманным путем, попробуйте выполнять функциональные тесты в промежуточном режиме относительно вашего существующего веб-приложения и просто пропустите все, что связано с fabric, хотя, если вы спросите меня, это большая отмазка. Вы же всегда можете обновить учетную запись, а затем сразу же отменить и потребовать возмещения согласно 30-дневной гарантии ;)

Если вы используете PythonAnywhere, чтобы работать, следя за изложением в книге, то я хотел бы знать, как у вас дела! Отправьте мне электронное письмо на obeythetes-tinggoat@gmail.com.

Приложение B

Представления на основе классов в Django

Этот дополнительный материал является продолжением главы 15, в которой мы реализовали формы Django для валидации и рефакторизовали наши представления. К концу главы наши представления по-прежнему использовали функции.

Однако сверкающей новизной игрушкой в мире Django являются представления, основанные на классах. В этом дополнении мы выполним рефакторизацию нашего приложения, чтобы использовать их вместо функций представления. Говоря конкретнее, попытаемся использовать обобщенные представления на основе классов.

Обобщенные представления на основе классов

Существует разница между представлениями на основе классов и *обобщенными* представлениями на основе классов. Представления на основе классов – это просто еще один способ определения функций представления. Они делают немного предположений относительно того, что ваши представления будут делать, и обеспечивают главное преимущество перед функциями представления – могут подразделяться на подклассы. Это достигается, пожалуй, за счет меньшей читаемости, чем традиционные представления на основе функций. Основной вариант применения представлений на основе *простых* классов – это когда у вас есть несколько представлений, которые повторно используют ту же самую логику. Мы хотим повиноваться принципу DRY. В случае с представлениями на основе функций вы будете применять вспомогательные функции или декораторы. В теории использование структуры с разделением на классы может дать более изящное решение.

Обобщенные представления на основе классов – это представления на основе таких классов, которые пытаются обеспечить готовые решения для распространенных случаев, таких как выборка объекта из базы данных и передача его в шаблон, выборка списка объектов, сохранение введенных

пользователем данных из POST-запроса при помощи ModelForm и т. д. Они кажутся похожими на наши варианты использования, но, как мы скоро увидим, дьявол кроется в деталях.

В этом месте я должен отметить, что не очень много пользовался обоими видами представлений на основе классов. Я определенно вижу в них смысл и много вариантов их применения в приложениях Django, куда обобщенные представления на основе классов отлично вписались бы. Однако как только ваш вариант использования слегка выходит за пределы элементарных вещей (к примеру, как только у вас более одной модели, которые вы хотите использовать), я нахожу, что использование представлений на основе классов может (опять-таки не утверждаю) привести к коду, который намного труднее читать, чем классическую функцию представления.

И тем не менее, ввиду того, что мы вынуждены использовать несколько вариантов индивидуализации представлений на основе классов, их реализация в этом случае может научить нас тому, как они работают и как можно их подвергнуть модульному тестированию.

Надеюсь, что модульные тесты, которые мы используем для представлений на основе функций, будут работать так же хорошо для представлений на основе классов. Посмотрим, как мы в этом преуспеем.

Домашняя страница как FormView

Наша домашняя страница просто показывает форму в шаблоне:

lists/views.py

```
def home_page(request):
    return render(request, 'home.html', {'form': ItemForm()})
```

Если просмотреть варианты, предлагаемые программной инфраструктурой Django[1], можно найти обобщенное представление под названием `FormView`. Давайте посмотрим, как оно работает:

lists/views.py (ch31l001)

```
from django.views.generic import FormView
[...]

class HomePageView(FormView):
    template_name = 'home.html'
    form_class = ItemForm
```

Мы сообщаем ему, какой шаблон и какую форму хотим использовать. Затем нам просто нужно обновить *urls.py*, заменив строку, в которой раньше было `lists.views.home_page`:

[1] См. https://docs.djangoproject.com/en/1.11/ref/class-based-views/

```
[...]
urlpatterns = [
    url(r'^$', list_views.HomePageView.as_view(), name='home'),
    url(r'^lists/', include(list_urls)),
]
```

И все тесты подтверждаются! Это было просто...

```
$ python manage.py test lists
[...]
Ran 34 tests in 0.119s

OK

$ python manage.py test functional_tests
[...]
Ran 5 tests in 15.160s

OK
```

Пока неплохо. Мы заменили однострочную функцию представления на класс, состоящий из двух строк, и он по-прежнему читаем. Самое время выполнить фиксацию...

Использование form_valid для индивидуализации CreateView

Теперь попробуем изменить представление, которое мы используем для создания совершенно нового списка, в настоящее время используется функция `new_list`. Вот как это выглядит теперь:

lists/views.py

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
    else:
        return render(request, 'home.html', {"form": form})
```

Просматривая возможные обобщенные представления на основе классов, вероятно, мы захотим применить `CreateView`, и мы знаем, что используем класс `ItemForm`, так что давайте посмотрим, как мы с ними поладим и помогут ли нам тесты:

lists/views.py (ch31l003)

```
from django.views.generic import FormView, CreateView
[...]

class NewListView(CreateView):
    form_class = ItemForm

def new_list(request):
    [...]
```

Я оставлю старую функцию представления в *views.py*, чтобы можно было копировать код оттуда. Его можно будет удалить, как только все заработает. Это безопасно при условии, что мы переключим преобразования URL-адресов на:

lists/urls.py (ch31l004)

```
[...]
urlpatterns = [
    url(r'^new$', views.NewListView.as_view(), name='new_list'),
    url(r'^(\d+)/$', views.view_list, name='view_list'),
]
```

В результате выполнения тестов мы получаем шесть ошибок:

```
$ python manage.py test lists
[...]
ERROR: test_can_save_a_POST_request (lists.tests.test_views.NewListTest)
TypeError: save() missing 1 required positional argument: 'for_list'

ERROR: test_for_invalid_input_passes_form_to_template
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'

ERROR: test_for_invalid_input_renders_home_template
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
'get_template_names()'

ERROR: test_invalid_list_items_arent_saved (lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin requires
either a definition of 'template_name' or an implementation of
```

```
'get_template_names()'

ERROR: test_redirects_after_POST (lists.tests.test_views.NewListTest)
TypeError: save() missing 1 required positional argument: 'for_list'

ERROR: test_validation_errors_are_shown_on_home_page
(lists.tests.test_views.NewListTest)
django.core.exceptions.ImproperlyConfigured: TemplateResponseMixin требует
либо определить 'template_name', либо реализовать 'get_template_names()'
FAILED (errors=6)
```

Давайте начнем с третьей. Может, просто добавить шаблон?

lists/views.py (ch31l005)

```
class NewListView(CreateView):
    form_class = ItemForm
    template_name = 'home.html'
```

Остается всего две неполадки: мы видим, что они происходят в функции `form_valid` обобщенного представления, и это одна из тех, которые можно переопределить для обеспечения индивидуализированного поведения в обобщенном представлении на основе классов. Как следует из ее имени, она выполняется, когда представление обнаружило допустимую форму. Мы можем просто скопировать часть кода из старой функции представления, которая раньше располагалась после `if form.is_valid():`:

lists/views.py (ch31l006)

```
class NewListView(CreateView):
    template_name = 'home.html'
    form_class = ItemForm

    def form_valid(self, form):
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
```

Это приводит к полному прохождению тестов!

```
$ python manage.py test lists
Ran 34 tests in 0.119s

OK

$ python manage.py test functional_tests
Ran 5 tests in 15.157s

OK
```

И мы даже *смогли* сэкономить еще две строки в попытке повиноваться принципу DRY, воспользовавшись одним из основных преимуществ представлений на основе классов – наследованием!

lists/views.py (ch31l007)

```
class NewListView(CreateView, HomePageView):

    def form_valid(self, form):
        list_ = List.objects.create()
        form.save(for_list=list_)
        return redirect(list_)
```

И все тесты по-прежнему проходят успешно:

```
OK
```

Это на самом деле не является хорошей объектноориентированной практикой. Наследование подразумевает отношение классификации «род – вид», и, наверное, бессмысленно говорить, что наше новое представление списка является видом представления домашней страницы… Так что, пожалуй, лучше всего этого не делать.

С учетом или без учета последнего шага, как это соотносится со старой версией? Я бы сказал, что неплохо. Мы экономим на стереотипном коде, при этом представление по-прежнему довольно ясное. Пока я бы сказал, что у нас один матч в пользу обобщенных представлений на основе классов и одна ничья.

Более сложное представление для обработки просмотра и добавления к списку

Мне потребовалось *несколько* попыток. И хотя тесты сообщили о том, что я все исправил, они не особо помогли выяснить шаги, как к этому прийти. Пришлось методом тыка копаться в функциях вроде `get_context_data`, `get_fom_kwargs` и т. д.

Правда, это помогло мне осознать смысл такого большого количества отдельных тестов, каждый из которых тестирует один аспект. В результате я вернулся и переписал несколько мест в главах 10–12.

Тесты нами руководят, но ненадолго

Вот как дела могут пойти дальше. Начните с мысли о том, что нам нужно представление `DetaHView`, которое показывало бы подробную информацию об объекте:

lists/views.py (ch31l009)

```
from django.views.generic import FormView, CreateView, DetailView
[...]

class ViewAndAddToList(DetailView):
    model = List
```

И его подключения в *urls.py*:

lists/urls.py (ch31l010)

```
    url(r'^(\d+)/$', views.ViewAndAddToList.as_view(), name='view_list'),
```

Это дает:

```
[...]
AttributeError: Обобщенное представление ViewAndAddToList с подробной
информацией должно вызываться либо с объектом pk, либо транслитом URL-адреса.

FAILED (failures=5, errors=6)
```

Не совсем очевидно, но, немного погуглив, я понял, что в регулярном выражении мне нужно было применить именованную группу:

lists/urls.py (ch31l011)

```
@@ -3,6 +3,6 @@ from lists import views

 urlpatterns = [
     url(r'^new$', views.NewListView.as_view(), name='new_list'),
-    url(r'^(\d+)/$', views.view_list, name='view_list'),
+    url(r'^(?P<pk>\d+)/$', views.ViewAndAddToList.as_view(), name='view_list')
 ]
```

Одна из следующего набора ошибок оказалась довольно полезной:

```
[...]
django.template.exceptions.TemplateDoesNotExist: lists/list_detail.html

FAILED (failures=5, errors=6)
```

Она легко разрешается:

lists/views.py (ch31l012)

```
class ViewAndAddToList(DetailView):
    model = List
    template_name = 'list.html'
```

Это приводит всего к пяти неполадкам и двум ошибкам:

```
[...]
ERROR: test_displays_item_form (lists.tests.test_views.ListViewTest)
KeyError: 'form'

FAILED (failures=5, errors=2)
```

Пока не останется только метод тыка

Наше представление не только показывает подробную информацию об объекте, но и позволяет создавать новые. Пусть это будут `DetailView` *и* `CreateView`, и, возможно, добавим `form_class`:

lists/views.py (ch31l013)

```
class ViewAndAddToList(DetailView, CreateView):
    model = List
    template_name = 'list.html'
    form_class = ExistingListItemForm
```

Но это дает очень много ошибок:

```
[...]
TypeError: в __init__() отсутствует 1 необходимый позиционный аргумент:
'for_list'
```

И ошибка `KeyError: 'form'` все там же!

В этой точке ошибки перестали как-то помогать, и уже нельзя было понять, что делать дальше. Мне пришлось обратиться к методу тыка. Тем не менее тесты по крайней мере давали знать, когда мои действия улучшали или ухудшали ситуацию.

Мои первые попытки использовать `get_form_kwargs` не совсем сработали, но я обнаружил, что могу использовать `get_form`:

lists/views.py (ch31l014)

```
def get_form(self):
    self.object = self.get_object()
    return self.form_class(for_list=self.object, data=self.request.POST)
```

Однако это будет работать, только если я по ходу также присвою значение свойству `self.object` в качестве побочного эффекта, что немного огорчает. И все же это приводит всего к трем ошибкам, но мы все еще не достигли цели!

```
django.core.exceptions.ImproperlyConfigured: No URL to redirect to. Either
provide a url or define a get_absolute_url method on the Model.
```

На прежние рельсы

И для этой заключительной неполадки тесты снова становятся полезными. Метод `get_absolute_url` довольно просто определяется на классе `Item`, в результате чего элементы указывают на страницу своего родительского списка:

lists/models.py (ch31l015)

```
class Item(models.Model):
[...]

    def get_absolute_url(self):
        return reverse('view_list', args=[self.list.id])
```

Это ваш окончательный ответ?

В итоге мы приходим к классу представления, который выглядит вот так:

lists/views.py

```
class ViewAndAddToList(DetailView, CreateView):
    model = List
    template_name = 'list.html'
    form_class = ExistingListItemForm

    def get_form(self):
        self.object = self.get_object()
        return self.form_class(for_list=self.object, data=self.request.POST)
```

Сравните старую верию с новой

Давайте взглянем на старую версию?

lists/views.py

```
def view_list(request, list_id):
    list_ = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=list_)
    if request.method == 'POST':
        form = ExistingListItemForm(for_list=list_, data=request.POST)
        if form.is_valid():
            form.save()
            return redirect(list_)
    return render(request, 'list.html', {'list': list_, "form": form})
```

Что ж, количество строк кода сократилось с девяти до семи. И все же я считаю, что версию на основе функций немного легче понять, потому что

в ней чуть меньше волшебства, а как утверждает Дзэн языка Python, явное лучше, чем неявное. В смысле… `SingleObjectMixin`? Что это за?.. И что еще обиднее, все это разваливается, если мы не присваиваем значение свойству `self.object` внутри `get_form`? Фу.

И все же я предполагаю, что что-то из этого каждый видит по-своему.

Лучшие приемы модульного тестирования обобщенных представлений на основе классов

Работая с этим, я чувствовал, что мои модульные тесты иногда были чересчур высокоуровневыми. Это неудивительно, поскольку тесты представлений с участием тестового клиента Django, по-видимому, целесообразно называть интегрированными.

Они сообщали об улучшении или ухудшении положения дел в результате моих действий, но они не всегда давали достаточно подсказок, как именно все исправить.

Временами я задавался вопросом, имеется ли какая-то дистанция в тесте, которая будет ближе к реализации, что-то вроде этого:

lists/tests/test_views.py

```
def test_cbv_gets_correct_object(self):
    our_list = List.objects.create()
    view = ViewAndAddToList()
    view.kwargs = dict(pk=our_list.id)
    self.assertEqual(view.get_object(), our_list)
```

Но это требует глубокого понимания внутреннего устройства обобщенных представлений Django на основе классов, чтобы верно сформировать тестовые условия для этих видов тестов. В итоге вы по-прежнему приходите к путанице из-за сложной иерархии наследования.

Для личного пользования: многочисленные изолированные тесты представлений с единственными утверждениями

Единственное, что я определенно вынес из этого дополнительного материала, это то, что много коротких модульных тестов представлений намного полезнее, чем немного тестов с описательной серией утверждений.

Рассмотрим вот этот монолитный тест:

lists/tests/test_views.py

```
def test_validation_errors_sent_back_to_home_page_template(self):
    response = self.client.post('/lists/new', data={'text': ''})
    self.assertEqual(List.objects.all().count(), 0)
    self.assertEqual(Item.objects.all().count(), 0)
```

```
    self.assertTemplateUsed(response, 'home.html')
    expected_error = escape("You can't have an empty list item")
    self.assertContains(response, expected_error)
```

Он определенно менее полезен, чем три отдельных теста, как вот эти:

lists/tests/test_views.py

```
def test_invalid_input_means_nothing_saved_to_db(self):
    self.post_invalid_input()
    self.assertEqual(List.objects.all().count(), 0)
    self.assertEqual(Item.objects.all().count(), 0)

def test_invalid_input_renders_list_template(self):
    response = self.post_invalid_input()
    self.assertTemplateUsed(response, 'list.html')

def test_invalid_input_renders_form_with_errors(self):
    response = self.post_invalid_input()
    self.assertIsinstance(response.context['form'], ExistingListItemForm)
    self.assertContains(response, escape(empty_list_error))
```

Причина в том, что в первом случае ранняя неполадка означает, что не все утверждения будут проверены. Если бы представление случайным образом сохраняло в базу данных по событию недопустимого POST-запроса, то вы получили бы раннюю неполадку и не узнали бы, использовало ли это представление правильный шаблон или отображение формы, или нет. Вторая формулировка намного упрощает отсев того, что именно работало.

Уроки, извлеченные из обобщенных представлений на основе классов

Обобщенные представления на основе классов могут делать что угодно

Не всегда понятно, что происходит, но зато с обобщенными представлениями на основе классов можно делать практически все.

Модульные тесты с единственным утверждением способствуют рефакторизации

Когда каждый модульный тест обеспечивает отдельное указание на то, что работает, а что нет, то намного проще настроить реализацию наших представлений для использования этой существенно отличающейся парадигмы.

Приложение C

Обеспечение работы серверной среды при помощи Ansible

Для автоматизации процесса развертывания новых версий исходного кода на наших серверах мы использовали Fabric. Однако обеспечение работы нового сервера и обновление файлов конфигурации Nginx и Gunicorn выполнялось в ручном режиме.

Этот вид задания все чаще передается средствам, которые называются инструментами управления конфигурацией или непрерывного развертывания. Популярные инструменты Chef и Puppet были первыми в этой области, а в мире Python – Salt и Ansible.

Из всех перечисленных проще всего начинать с Ansible. Для его работы требуется два файла:

```
pip2 install --user ansible # к сожалению, Python 2
```

Инвентарный файл в `deploy_tools/inventory.ansible` определяет, относительно каких серверов мы можем работать:

deploy_tools/inventory.ansible

```
[live]
superlists.ottg.eu ansible_become=yes ansible_ssh_user=elspeth

[staging]
superlists-staging.ottg.eu ansible_become=yes ansible_ssh_user=elspeth

[local]
localhost ansible_ssh_user=root ansible_ssh_port=6666 ansible_host=127.0.0.1
```

(Локальная запись `local` – это просто пример; в моем случае – виртуальная машина Virtualbox с настройками для перенаправления портов 22 и 80.)

Установка системных пакетов и Nginx

Далее сценарий (или план игры) Ansible, который определяет, что делать на сервере. В нем используется синтаксис YAML:

deploy_tools/provision.ansible.yaml

```
---

- hosts: all

  vars:
      host: "{{ inventory_hostname }}"

  tasks:
    - name: Deadsnakes PPA to get Python 3.6
      apt_repository:
        repo='ppa:fkrull/deadsnakes'
    - name: make sure required packages are installed
      apt: pkg=nginx,git,python3.6,python3.6-venv state=present

    - name: allow long hostnames in nginx
      lineinfile:
        dest=/etc/nginx/nginx.conf
        regexp='(\s+)#? ?server_names_hash_bucket_size'
        backrefs=yes
        line='\1server_names_hash_bucket_size 64;'

    - name: add nginx config to sites-available
      template: src=./nginx.conf.j2 dest=/etc/nginx/sites-available/{{ host }}
      notify:
          - restart nginx

    - name: add symlink in nginx sites-enabled
      file:
          src=/etc/nginx/sites-available/{{ host }}
          dest=/etc/nginx/sites-enabled/{{ host }}
          state=link
      notify:
          - restart nginx
```

Переменная `inventory_hostname` – это доменное имя сервера, относительно которого мы работаем. Я использую раздел `vars` для его переименования в host просто ради удобства.

В этом разделе мы устанавливаем необходимое нам программное обеспечение, используя команду `apt`, поправляем конфигурацию Nginx, чтобы разрешить длинные имена хостов, используя для этого функционал замены в регулярных выражениях, и затем пишем файл конфигурации Nginx, используя шаблон. Это модифицированная версия файла шаблона, который мы сохранили в *deploy_tools/nginx.template.conf* в главе 9. Но теперь он

использует специфический шаблонный синтаксис – Jinja2, который на самом деле во многом походит на шаблонный синтаксис Django:

deploy_tools/nginx.conf.j2

```
server {
    listen 80;
    server_name {{ host }};

    location /static {
        alias /home/{{ ansible_ssh_user }}/sites/{{ host }}/static;
    }

    location / {
        proxy_set_header Host {{ host }};
        proxy_pass http://unix:/tmp/{{ host }}.socket;
    }
}
```

Конфигурирование Gunicorn и использование обработчиков для перезапуска служб

Вот вторая половина нашего сценария:

deploy_tools/provision.ansible.yaml

```
    - name: write gunicorn service script
      template:
          src=./gunicorn.service.j2
          dest=/etc/systemd/system/gunicorn-{{ host }}.service
       notify:
          - restart gunicorn

handlers:
  - name: restart nginx
    service: name=nginx state=restarted

  - name: restart gunicorn
    systemd:
        name=gunicorn-{{ host }}
        daemon_reload=yes
        enabled=yes
        state=restarted
```

Мы еще раз используем шаблон для конфигурирования Gunicorn:

deploy_tools/gunicorn.service.j2

```
[Unit]
Description=Gunicorn server for {{ host }}

[Service]
User={{ ansible_ssh_user }}
WorkingDirectory=/home/{{ ansible_ssh_user }}/sites/{{ host }}/source
Restart=on-failure
ExecStart=/home/{{ ansible_ssh_user }}/sites/{{ host }}/virtualenv/bin/gunicorn \
    --bind unix:/tmp/{{ host }}.socket \
    --access-logfile ../access.log \
    --error-logfile ../error.log \
    superlists.wsgi:application

[Install]
WantedBy=multi-user.target
```

И теперь у нас два обработчика для перезапуска Nginx и Gunicorn. Ansible достаточно умен, поэтому если он видит, что все многочисленные шаги делают вызов одинаковых обработчиков, он ожидает последнего, прежде чем его вызвать.

Вот, собственно, и все! Команда, которая все это заводит, такая:

```
ansible-playbook -i inventory.ansible provision.ansible.yaml --limit=staging
--ask-become-pass
```

Более подробную информацию смотрите в документации по Ansible[1].

Что делать дальше

Я только что дал немного продегустировать, что можно делать с Ansible. Но чем больше вы автоматизируете развертывания, тем больше уверенности в них получите. Вот еще несколько нюансов, на которые стоит обратить внимание.

Переместите развертывание из Fabric в Ansible

Мы видели, что Ansible помогает с некоторыми аспектами настройки, но он также может в значительной степени выполнить за нас весь процесс развертывания. Попробуйте расширить сценарий, чтобы сделать все, что мы в настоящее время делаем в сценарии развертывания с использованием Fabric, включая, если нужно, уведомление о перезапусках.

[1] См. https://docs.ansible.com/

Используйте Vagrant для запуска локальной виртуальной машины

Выполнение тестов относительно промежуточного сайта вселяет в нас окончательную уверенность, что все будет работать, когда мы перейдем в производственную среду, но мы также можем задействовать виртуальную машину (VM) на нашей локальной машине.

Скачайте Vagrant и Virtualbox и убедитесь, что Vagrant смог выполнить сборку сервера разработки на вашем собственном ПК, используя наш сценарий Ansible для развертывания программного кода. Перепишите исполнитель ФТ, чтобы иметь возможность тестировать относительно локальной VM.

Наличие файла конфигурации Vagrant особенно полезно при работе в команде – он помогает новым разработчикам запускать серверы, которые выглядят в точности как ваш.

Приложение D

Тестирование миграций базы данных

В Django и в его предшественнике South миграции существуют уже целую вечность, поэтому, как правило, нет необходимости тестировать миграции базы данных. Однако так уж сложилось, что мы вносим опасный тип миграции, который налагает новое ограничение целостности на наши данные. Когда я в самом начале выполнял сценарий миграции относительно промежуточного сервера, я обнаружил ошибку.

На более крупных проектах с уязвимыми данными может понадобиться дополнительная уверенность, которую дает тестирование миграций в безопасной среде перед их применением к производственным данным. Так что этот игрушечный пример, будем надеяться, станет полезной репетицией.

Еще одна типичная причина, почему необходимо тестировать миграции, – это скорость. Миграции часто связаны с временем простоя (временем работы вхолостую), и иногда, когда они применяются к очень большим наборам данных, их выполнение может занимать некоторое время. И неплохо бы знать заранее его продолжительность.

Попытка развертывания на промежуточном сервере

Вот что произошло со мной, когда я в самом начале попытался развернуть наши новые валидационные ограничения в главе 17:

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
[...]
Running migrations:
  Applying lists.0005_list_item_unique_together...Traceback (most recent call
last):
  File "/usr/local/lib/python3.6/dist-packages/django/db/backends/utils.py",
line 61, in execute
    return self.cursor.execute(sql, params)
```

```
  File "/usr/local/lib/python3.6/dist-packages/django/db/backends/sqlite3/base.py",
line 475, in execute
    return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: columns list_id, text are not unique
[...]
```

А произошло то, что некоторые существующие данные в базе данных нарушили ограничение целостности, поэтому БД жаловалась, когда я пытался их применить.

Чтобы справиться с этим видом проблем, потребуется выполнить сборку миграции данных. Давайте сначала установим локальную среду, относительно которой будем тестировать.

Выполнение тестовой миграции локально

Мы будем использовать копию реальной базы данных, относительно которой будем тестировать нашу миграцию.

Будьте очень и очень осторожны, когда используете реальные данные в целях тестирования. К примеру, они могут содержать реальные адреса электронной почты клиентов – вы же не хотите, чтобы куча тестовых электронных писем была случайно отправлена им. Спросите у меня, уж я-то знаю.

Ввод проблематичных данных

На своем реальном сайте создайте список из нескольких двойных элементов, как показано на рис. D.1.

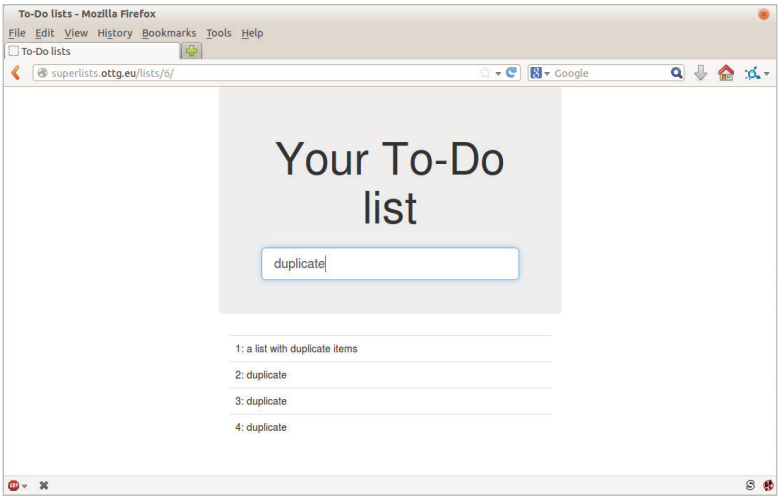


Рис. D.1. Список с двойными элементами

Копирование данных тестирования из реального сайта

Скопируйте базу данных из реального сайта:

```
$ scp elspeth@superlists.ottg.eu:\
/home/elspeth/sites/superlists.ottg.eu/database/db.sqlite3 .
$ mv ../database/db.sqlite3 ../database/db.sqlite3.bak
$ mv db.sqlite3 ../database/db.sqlite3
```

Подтверждение ошибки

Теперь у нас есть локальная база данных, которая не была подвергнута миграции и содержит некоторые проблематичные данные. Если мы попытаемся выполнить команду `migrate`, то увидим ошибку:

```
$ python manage.py migrate --migrate
python manage.py migrate
Operations to perform:
[...]
Running migrations:
[...]
  Applying lists.0005_list_item_unique_together...Traceback (most recent call
last):
[...]
    return Database.Cursor.execute(self, query, params)
sqlite3.IntegrityError: columns list_id, text are not unique
```

Вставка миграции данных

Миграции данных[1] – это особый тип миграции, которая вместо изменения схемы модифицирует данные в БД. Нам нужно создать такую миграцию, которая будет выполняться до того, как мы применим правило ограничения целостности, чтобы профилактически удалить любые копии. Вот как можно это сделать:

```
$ git rm lists/migrations/0005_list_item_unique_together.py
$ python manage.py makemigrations lists --empty
Migrations for 'lists':
  0005_auto_20140414_2325.py:
$ mv lists/migrations/0005_*.py lists/migrations/0005_remove_duplicates.py
```

Обратитесь к документации Django по миграциям данных[2], чтобы получить более подробную информацию. Однако вот тут показано, как добавить некоторые инструкции, чтобы изменить существующие данные:

[1] См. https://docs.djangoproject.com/en/1.11/topics/migrations/#data-migrations

[2] См. https://docs.djangoproject.com/en/dev/topics/migrations/%23data-migrations

lists/migrations/0005_remove_duplicates.py

```
# encoding: utf8
from django.db import models, migrations

def find_dupes(apps, schema_editor):
    List = apps.get_model("lists", "List")
    for list_ in List.objects.all():
        items = list_.item_set.all()
        texts = set()
        for ix, item in enumerate(items):
            if item.text in texts:
                item.text = '{} ({})'.format(item.text, ix)
                item.save()
            texts.add(item.text)

class Migration(migrations.Migration):

    dependencies = [
        ('lists', '0004_item_list'),
    ]

    operations = [
        migrations.RunPython(find_dupes),
    ]
```

Повторное создание старой миграции

Мы повторно создаем старую миграцию при помощи команды `makemigrations`, которая гарантирует, что эта шестая миграция имеет явную зависимость от `0005`, то есть от миграции данных:

```
$ python manage.py makemigrations
Migrations for 'lists':
  0006_auto_20140415_0018.py:
    - Alter unique_together for item (1 constraints)
$ mv lists/migrations/0006_* lists/migrations/0006_unique_together.py
```

Совместное тестирование новых миграций

Теперь мы готовы выполнить наш тест относительно реальных данных:

```
$ cd deploy_tools
$ fab deploy:host=elspeth@superlists-staging.ottg.eu
[...]
```

Нам также потребуется перезапустить реальное задание Gunicorn:

```
elspeth@server:$ sudo systemctl restart gunicorn-superlists.ottg.eu
```

И теперь можно выполнить ФТ относительно промежуточного сервера:

```
$ STAGING_SERVER=superlists-staging.ottg.eu python manage.py test
functional_tests
[...]
....
 ---------------------------------------------------------------------
Ran 4 tests in 17.308s

OK
```

Кажется, все в порядке! Давайте сделаем это относительно реального сервера:

```
$ fab deploy --host=superlists.ottg.eu
[superlists.ottg.eu] Executing task 'deploy'
[...]
```

И на этом все. `git add lists/migrations`, `git commit` и т. д.

Выводы

Это упражнение в первую очередь было ориентировано на создание миграции данных и ее тестирование относительно некоторых реальных данных. Конечно, это лишь капля в море возможного тестирования, которое вы могли бы выполнить для миграции. Вы можете представить создание автоматизированных тестов для проверки, что все ваши данные сохранены, сравнивающих содержимое базы данных до и после. Вы могли бы написать отдельные модульные тесты для вспомогательных функций в миграции данных. Вы могли бы потратить больше времени на измерение продолжительности миграции и поэкспериментировать с приемами ее сокращения – к примеру, путем разбивки миграции на шаги из большего или меньшего числа компонентов.

Помните, что все это будет относительно редким случаем. На моем опыте я не чувствовал необходимости тестировать 99% миграций, с которыми я работал. Но если в своем проекте вы когда-нибудь почувствуете такую необходимость, я надеюсь, что здесь вы нашли несколько указаний, с которых можно будет начать.

О тестировании миграций базы данных

Опасайтесь миграций, которые вносят ограничения

99% миграций проходят без помех, но опасайтесь любых ситуаций, таких как эта, где вы вносите новое ограничение на столбцы, которые уже существуют.

Тестируйте миграции на скорость

Когда ваш проект становится большим, следует подумать о тестировании количества времени, которое будут занимать миграции. Миграции базы данных обычно связаны с временем простоя, поскольку в зависимости от базы данных операция обновления схемы может заблокировать таблицу, с которой она работает, пока не завершится. Неплохо использовать ваш промежуточный сайт, чтобы засечь, сколько времени займет процесс миграции.

Будьте чрезвычайно осторожны при использовании дампа производственных данных

Для этого вам понадобится заполнить вашу базу данных промежуточного сайта объемом данных, который соразмерен вашим производственным данным. Пояснения, как это сделать, выходят за пределы настоящей книги, однако я скажу одно: если вы испытываете желание просто взять дамп своей производственной базы данных и загрузить его в промежуточную базу данных, нужно быть очень осторожными. Производственные данные содержат подробную информацию о реальных потребителях, и я лично отвечал за случайную рассылку несколько сотен неправильных счетов, после того как автоматизированный процесс на моем промежуточном сервере начал обрабатывать скопированные производственные данные, которые я только что загрузил. Вечерок не задался.

Приложение E

Разработка на основе поведения (BDD)

Надо сказать, я не работал с методологией BDD (от англ. Behaviour-Driven Development) достаточно подробно, так что не могу утверждать, что обладаю опытом в этой сфере. Но мне понравилось то, что я видел, поэтому решил предложить вам ураганный тур по данной теме. В этом дополнительном материале мы возьмем часть тестов, которые написали в «нормальном» ФТ, и конвертируем их для использования инструментов BDD.

Что такое BDD?

Строго говоря, BDD – это методология, а не комплект инструментов. Это подход к тестированию приложения путем тестирования поведения, которое оно будет демонстрировать пользователю (статья в Википедии[1] содержит вполне хороший обзор). Поэтому до определенной степени функциональные тесты на основе Selenium, которые я показал в книге, *можно* назвать BDD.

Этот термин тесно связан с набором инструментов для применения BDD, прежде всего, с Gherkin (переводится как «Корнишон»)[2] – человекочитаемым предметно-ориентированным языком (DSL) для написания функциональных (приемочных) тестов. Gherkin вышел из мира Ruby, где он связан с исполнителем тестов Cucumber[3].

В мире Python имеется пара эквивалентных инструментов для выполнения тестов: Lettuce и Behave[4]. В момент написания данного материала только Behave был совместим с Python 3, им-то мы воспользуемся. Также применим плагин behave-django[5].

---

1 См. https://en.wikipedia.org/wiki/Behavior-driven_development

2 См. https://github.com/cucumber/cucumber/wiki/Gherkin

3 См. http://cukes.info/

4 См., соответственно, http://lettuce.it/ и http://pythonhosted.org/behave/

5 См. https://pythonhosted.org/behave-django/

Получение исходного кода для этих примеров

Я собираюсь использовать пример из главы 22. У нас есть элементарный сайт списков неотложных дел, и мы хотим добавить новую опцию: возможность для зарегистрированных пользователей просматривать в одном месте списки, которые они создали. До сего момента все списки практически анонимные.

Если вы следовали указаниям из книги, можете пропустить вплоть до программного кода, соответствующего этой точке. Если вы хотите получить его из моего репозитория, обратитесь к ветке для главы 17 (https://github.com/hjwp/book-example/tree/chapter_17).

Базовые служебные операции

Мы создаем каталог для компонентов BDD (так называемых features), добавляем каталог *steps* (вскоре мы узнаем, что они означают!) и заготовку для первого компонента:

```
$ mkdir -p features/steps
$ touch features/my_lists.feature
$ touch features/steps/my_lists.py
$ tree features
features
├── my_lists.feature
└── steps
    └── my_lists.py
```

Устанавливаем behave-django и добавляют его в *settings.py*:

```
$ pip install behave-django
```

superlists/settings.py

```
--- a/superlists/settings.py
+++ b/superlists/settings.py
@@ -40,6 +40,7 @@ INSTALLED_APPS = [
     'lists',
     'accounts',
     'functional_tests',
+    'behave_django',
]
```

И затем выполняем `python manage.py behave` для проверки исправности:

```
$ python manage.py behave
Creating test database for alias 'default'...
```

```
0 features passed, 0 failed, 0 skipped
0 scenarios passed, 0 failed, 0 skipped
0 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.000s
Destroying test database for alias 'default'...
```

Написание ФТ как компонента при помощи синтаксиса языка Gherkin

До настоящего момента мы писали ФТ, используя человекочитаемые комментарии, которые описывают новый функциональный компонент с точки зрения истории пользователя с вкраплениями программного кода для Selenium, необходимого для выполнения каждого шага в истории.

BDD проводит между ними различие: мы пишем человекочитаемую историю с помощью человекочитаемого (иногда несколько неуклюжего) языка Gherkin, и она называется компонентом описания (feature). Позже мы преобразуем каждую строку на Gherkin в функцию, которая содержит код для Selenium, необходимый для реализации этого шага.

Вот как выглядит компонент для нашей новой страницы «Мои списки»:

features/my_lists.feature

```
Feature: My Lists
    As a logged-in user
    I want to be able to see all my lists in one page
    So that I can find them all after I've written them

    Scenario: Create two lists and see them on the My Lists page

        Given I am a logged-in user

        When I create a list with first item "Reticulate Splines"
            And I add an item "Immanentize Eschaton"
            And I create a list with first item "Buy milk"

        Then I will see a link to "My lists"

        When I click the link to "My lists"
        Then I will see a link to "Reticulate Splines"
        And I will see a link to "Buy milk"

        When I click the link to "Reticulate Splines"
        Then I will be on the "Reticulate Splines" list page
```

As-a/I want to/So that

Вверху вы видите утверждение As-a/I want to/So that (В качестве/я хочу/с тем чтобы). Оно необязательное и не имеет исполнимого эквивалента. Это просто слегка формализованный способ определения аспектов «Кто и почему?» истории пользователя, ненавязчиво побуждающий команду разработчиков думать об обоснованиях для каждого компонента.

Given/When/Then

Given/When/Then (При условии/Когда/Тогда) – реальное ядро BDD-теста. Эта триединая формула соответствует образцу «установить/осуществить/подтвердить», который мы видели в модульных тестах, и представляет фазу настроек/предположений, фазу осуществления/действия и последующую фазу утверждения/наблюдения. Более подробную информацию можно найти в wiki-справочнике по Cucumber[6].

Не всегда идеальная подгонка!

Как видите, бывает нелегко запихнуть историю пользователя ровно в три шага! Для расширения одного шага можно использовать утверждение `And`. Я добавил многочисленные шаги `When` и последующие `Then`, чтобы проиллюстрировать дальнейшие аспекты нашей страницы «Мои списки».

Программирование шаговых функций

Теперь мы создадим эквивалент для нашего компонента на языке Gherkin, который будет представлен шаговыми функциями, то есть функциями, реализующими шаги в программном коде практически.

Генерация заготовок шагов

Когда мы выполняем команду `behave`, она с готовностью сообщает обо всех шагах, которые мы должны реализовать:

```
$ python manage.py behave
Feature: My Lists # features/my_lists.feature:1
  As a logged-in user
  I want to be able to see all my lists in one page
  So that I can find them all after I've written them
  Scenario: Create two lists and see them on the My Lists page #
features/my_lists.feature:6
    Given I am a logged-in user # None
    Given I am a logged-in user # None
    When I create a list with first item "Reticulate Splines" # None
```

[6] См. https://github.com/cucumber/cucumber/wiki/Given-When-Then

```
    And I add an item "Immanentize Eschaton" # None
    And I create a list with first item "Buy milk" # None
    Then I will see a link to "My lists" # None
    When I click the link to "My lists" # None
    Then I will see a link to "Reticulate Splines" # None
    And I will see a link to "Buy milk" # None
    When I click the link to "Reticulate Splines" # None
    Then I will be on the "Reticulate Splines" list page # None

Failing scenarios:
  features/my_lists.feature:6 Create two lists and see them on the My Lists
page

0 features passed, 1 failed, 0 skipped
0 scenarios passed, 1 failed, 0 skipped
0 steps passed, 0 failed, 0 skipped, 10 undefined
Took 0m0.000s

You can implement step definitions for undefined steps with these snippets:

@given(u'I am a logged-in user')
def step_impl(context):
  raise NotImplementedError(u'STEP: Given I am a logged-in user')

@when(u'I create a list with first item "Reticulate Splines"')
def step_impl(context):
[...]
```

Весь этот вывод приятно окрашен, как показано на рис. E.1.

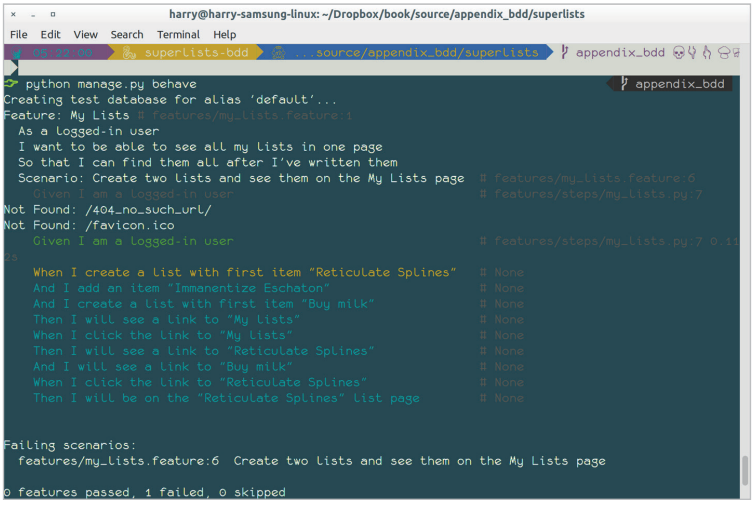


Рис. E.1. Behave с цветным консольным выводом

Он побуждает нас скопировать и вставить эти фрагменты и использовать их в качестве начальных точек для создания наших шагов.

Определение первого шага

Вот первая заглушка при создании шаговой функции для шага «Given I am a logged-in user» («Given я – зарегистрированный пользователь»). Сначала я взял код для `self.create_pre_authenticated_session` из *functional_tests/test_my_lists.py* и слегка его адаптировал (удалил серверную версию, хотя ее будет легко вернуть).

features/steps/my _Hsts.py

```
from behave import given, when, then
from functional_tests.management.commands.create_session import \
    create_pre_authenticated_session
from django.conf import settings

@given('I am a logged-in user')
def given_i_am_logged_in(context):
    '''при условии, что я являюсь зарегистрированным пользователем'''
    session_key = create_pre_authenticated_session(email='edith@example.com')
    ## установить cookie, которые нужны для первого посещения домена.
    ## страницы 404 загружаются быстрее всего!
    context.browser.get(context.get_url("/404_no_such_url/"))
    context.browser.add_cookie(dict(
        name=settings.SESSION_COOKIE_NAME,
        value=session_key,
        path='/',
    ))
```

Переменная `context` требует небольшого разъяснения. Это глобальная переменная в том смысле, что она передается в каждый выполняемый шаг и может использоваться для хранения информации, которой нам нужно обмениваться между шагами. Здесь мы будем хранить в ней объект браузера и URL-адрес сервера `server_url`. По сути, мы используем ее во многом так же, как использовали свойство `self`, когда писали функциональные тесты на основе `unittest`.

Эквиваленты setUp и tearDown в environment.py

Шаговые функции могут вносить изменения в состояния переменной `context`, но место, где делается предварительная установка, – эквивалент `setUp` – находится в файле с именем *environment.py*.

features/environment.py

```
from selenium import webdriver

def before_all(context):
    context.browser = webdriver.Firefox()

def after_all(context):
    context.browser.quit()

def before_feature(context, feature):
    pass
```

Еще один прогон

Для проверки исправности можно выполнить еще один прогон, чтобы убедиться, что новая шаговая функция работает и что мы действительно можем запустить браузер:

```
$ python manage.py behave
[...]
1 step passed, 0 failed, 0 skipped, 9 undefined
```

В результате получим стандартные стопки вывода, но видим, что первый шаг тест прошел успешно. Давайте определим остальные шаги.

Извлечение параметров в шагах

Мы увидим, как команда `behave` позволяет извлекать параметры из описаний шагов. Наш следующий шаг говорит:

features/my_lists.feature

```
    When I create a list with first item "Reticulate Splines"
```

И автоматически сгенерированное определение шага выглядит так:

features/steps/my_lists.py

```
@given('I create a list with first item "Reticulate Splines"')
def step_impl(context):
    raise NotImplementedError(
        u'STEP: When I create a list with first item "Reticulate Splines"'
    )
```

Мы хотим добавить возможность создавать списки с произвольными первыми элементами, поэтому было бы здóрово каким-то образом захватывать то, что между кавычками, и передавать их как аргумент в более

обобщенную функцию. В BDD это обычное условие, и `behave` имеет для него хороший синтаксис, напоминающий новый синтаксический стиль строкового форматирования в Python:

features/steps/my_lists.py (ch35l006)

```
[...]
@when('I create a list with first item "{first_item_text}"')
def create_a_list(context, first_item_text):
    context.browser.get(context.get_url('/'))
    context.browser.find_element_by_id('id_text').send_keys(first_item_text)
    context.browser.find_element_by_id('id_text').send_keys(Keys.ENTER)
    wait_for_list_item(context, first_item_text)
```

Круто, правда?

Извлечение параметров для шагов является одной из наиболее мощных функциональных особенностей синтаксиса BDD.

Как всегда бывает с тестами Selenium, нам потребуется явное ожидание. Давайте снова воспользуемся декоратором `@wait` из *base.py*:

features/steps/my_lists.py (ch35l007)

```
from functional_tests.base import wait
[...]

@wait
def wait_for_list_item(context, item_text):
    context.test.assertIn(
        item_text,
        context.browser.find_element_by_css_selector('#id_list_table').text
    )
```

Точно так же можно добавлять в существующий список и просматривать либо нажимать на ссылки:

features/steps/my_lists.py (ch35l008)

```
from selenium.webdriver.common.keys import Keys
[...]

@when('I add an item "{item_text}"')
def add_an_item(context, item_text):
    context.browser.find_element_by_id('id_text').send_keys(item_text)
```

```
    context.browser.find_element_by_id('id_text').send_keys(Keys.ENTER)
    wait_for_list_item(context, item_text)

@then('I will see a link to "{link_text}"')
@wait
def see_a_link(context, link_text):
    context.browser.find_element_by_link_text(link_text)

@when('I click the link to "{link_text}"')
def click_link(context, link_text):
    context.browser.find_element_by_link_text(link_text).click()
```

Обратите внимание: мы даже можем применить декоратор `@wait` на самих шагах.

И наконец немного более сложный шаг, который говорит, что я нахожусь на странице конкретного списка:

features/steps/my_lists.py (ch35l009)

```
@then('I will be on the "{first_item_text}" list page')
@wait
def on_list_page(context, first_item_text):
    first_row = context.browser.find_element_by_css_selector(
        '#id_list_table tr:first-child'
    )
    expected_row_text = '1: ' + first_item_text
    context.test.assertEqual(first_row.text, expected_row_text)
```

Теперь можем выполнить его и увидеть первую ожидаемую неполадку:

```
$ python manage.py behave
Feature: My Lists # features/my_lists.feature:1
  As a logged-in user
  I want to be able to see all my lists in one page
  So that I can find them all after I've written them
  Scenario: Create two lists and see them on the My Lists page #
features/my_lists.feature:6
  Given I am a logged-in user #
features/steps/my_lists.py:19
  When I create a list with first item "Reticulate Splines" #
features/steps/my_lists.py:31
  And I add an item "Immanentize Eschaton" #
features/steps/my_lists.py:39
  And I create a list with first item "Buy milk" #
features/steps/my_lists.py:31
  Then I will see a link to "My lists" #
```

```
functional_tests/base.py:12
    Traceback (most recent call last):
[...]
      File "features/steps/my_lists.py", line 49, in see_a_link
        context.browser.find_element_by_link_text(link_text)
[...]
      selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: My lists

[...]

Failing scenarios:
  features/my_lists.feature:6 Create two lists and see them on the My Lists
page

0 features passed, 1 failed, 0 skipped
0 scenarios passed, 1 failed, 0 skipped
4 steps passed, 1 failed, 5 skipped, 0 undefined
```

Вы видите, что информация на выходе действительно дает представление о том, насколько далеко мы прошлись по истории теста: нам удается создать два списка, но ссылка «Мои списки» не появляется.

BDD по сравнению с ФТ с локальными комментариями

Я не буду просматривать реализацию этого функционала, но вы можете убедиться, что тест управляет разработкой точно так же, как в ФТ с локальными комментариями.

Давайте взглянем на него для сравнения:

functional_tests/test_my_lists.py

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
    # Эдит является зарегистрированным пользователем
    self.create_pre_authenticated_session('edith@example.com')

    # Она открывает домашнюю страницу и начинает новый список
    self.browser.get(self.live_server_url)
    self.add_list_item('Reticulate splines')
    self.add_list_item('Immanentize eschaton')
    first_list_url = self.browser.current_url

    # Она впервые видит ссылку "Мои списки".
```

```
    self.browser.find_element_by_link_text('My lists').click()

    # Она видит, что ее список там, и он назван
    # по первому элементу списка
    self.wait_for(
        lambda: self.browser.find_element_by_link_text('Reticulate
splines')
    )
    self.browser.find_element_by_link_text('Reticulate splines').click()
    self.wait_for(
        lambda: self.assertEqual(self.browser.current_url, first_list_url)
    )

    # Она решает начать еще один список, просто чтобы убедиться
    self.browser.get(self.live_server_url)
    self.add_list_item('Click cows')
    second_list_url = self.browser.current_url

    # Ее новый список появляется под заголовком «Мои списки»
    self.browser.find_element_by_link_text('My lists').click()
    self.wait_for(
        lambda: self.browser.find_element_by_link_text('Click cows')
    )
    self.browser.find_element_by_link_text('Click cows').click()
    self.wait_for(
        lambda: self.assertEqual(self.browser.current_url, second_list_
url)
    )

    # Она выходит из системы. Опция «Мои списки» исчезает
    self.browser.find_element_by_link_text('Log out').click()
    self.wait_for(lambda: self.assertEqual(
        self.browser.find_elements_by_link_text('My lists'),
        []
    ))
```

Это сравнение сопоставимо не полностью, но мы можем взглянуть на число строк программного кода в табл. Е.1.

Таблица Е.1. Сравнение строк программного кода

BDD	Стандартный ФТ
Файл компонента: 20 (3 необязательных)	тело тестовой функции: 45
Файл шагов: 56 строк	вспомогательные функции: 23

Сравнение не идеальное, но вы можете заметить, что файл компонента и тело тестовой функции стандартного ФТ эквивалентны в том, что они представляют основную историю теста, в то время как шаги и вспомогательные функции представляют скрытые детали реализации. Если их сложить, то общие значения вполне сопоставимы. Но обратите внимание что они по-разному распределены: BDD-тесты сделали историю более сжатой и вытеснили большую часть работы в скрытые детали реализации.

BDD способствует написанию структурированного тестового кода

По крайней мере меня это действительно привлекает: инструмент BDD вынудил нас структурировать тестовый код. В ФТ с локальными комментариями мы свободны использовать столько строк, сколько нам нужно для реализации шага, согласно его описанию в строке комментария. Очень трудно сопротивляться порыву просто скопипастить откуда-нибудь программный код или взять его из более ранних мест в тесте. К этому моменту вы уже убедились, что таким образом я создал несколько вспомогательных функций (например, `get_item_input_box`).

Синтаксис BDD, наоборот, сразу вынудил меня сделать отдельную функцию для каждого шага, и в результате я уже создал многоразовый программный код, чтобы:

- Начать новый список.
- Добавлять элемент в существующий список.
- Нажимать на ссылку с определенным текстом.
- Утверждать, что я смотрю на страницу определенного списка.

BDD действительно подталкивает к написанию тестового кода, который хорошо соответствует предметной области бизнеса, и использовать уровень абстракции между историей ФТ и его реализацией в коде.

Окончательно это проявляется в том, что теоретически если вы захотели бы поменять язык программирования, то могли бы сохранить все свои компоненты в синтаксисе языка Gherkin в том виде, в каком они есть, отбросить шаги на Python и заменить их шагами, реализованными на другом языке.

Страничный шаблон проектирования как альтернатива

В главе 25 книги я привел пример шаблона проектирования Page Pattern, который представляет собой объектноориентированный подход к структурированию тестов на основе Selenium. Вот как он выглядит:

functional_tests/test_sharing.py

```
from .my_lists_page import MyListsPage
[...]

class SharingTest(FunctionalTest):

    def test_can_share_a_list_with_another_user(self):
        # [...]
        self.browser.get(self.live_server_url)
        list_page = ListPage(self).add_list_item('Get help')

        # Она замечает опцию "Поделиться этим списком"
        share_box = list_page.get_share_box()
        self.assertEqual(
                share_box.get_attribute('placeholder'),
                'your-friend@example.com'
        )

        # Она делится своим списком.
        # Страница обновляется, говоря, что он предоставлен Анциферу:
        list_page.share_list_with('oniciferous@example.com')
```

А класс страницы выглядит так:

functional_tests/lists_pages.py

```
class ListPage(object):

    def __init__(self, test):
        self.test = test

    def get_table_rows(self):
        return self.test.browser.find_elements_by_css_selector('#id_list_table tr')

    @wait
    def wait_for_row_in_list_table(self, item_text, item_number):
        row_text = '{}: {}'.format(item_number, item_text)
        rows = self.get_table_rows()
        self.test.assertIn(row_text, [row.text for row in rows])

    def get_item_input_box(self):
        return self.test.browser.find_element_by_id('id_text')
```

Так что в функциональных тестах с локальными комментариями определенно существует возможность реализации аналогичного уров-

ня абстракции и своего рода предметноориентированного языка (DSL) с использованием страничного шаблона проектирования или другой структуры, которую вы предпочитаете. Но теперь это вопрос самодисциплины – без программной инфраструктуры, которая подталкивала бы вас к этому.

В принципе вы можете использовать страничный шаблон проектирования с BDD в качестве ресурса, который использует ваши шаги при навигации по страницам сайта.

BDD может быть менее выразительным, чем локальные комментарии

С другой стороны, я вижу возможность некоторых ограничений со стороны синтаксиса Gherkin. Сравните выразительные и читаемые локальные комментарии с немного неуклюжим компонентом BDD:

functional_tests/test_my_lists.py

```
    # Эдит является зарегистрированным пользователем
    # Она открывает домашнюю страницу и начинает новый список
    # Она впервые видит ссылку "Мои списки".
    # Она видит, что ее список там, и он назван
    # по первому элементу списка
    # Она решает начать еще один список, просто чтобы убедиться
    # Ее новый список появляется под заголовком «Мои списки»
    # Она выходит из системы. Опция «Мои списки» исчезает
[...]
```

Они выглядят гораздо более естественными, чем немного натянутые заклинания Given/Then/When, и в некотором смысле могут способствовать мышлению, более ориентированному на пользователя. (В Gherkin есть синтаксическая конструкция для включения комментариев в файл компонента, которая несколько смягчает этот недостаток, но она еще не получила широкого применения.)

Будут ли непрограммисты писать тесты?

Одно из первоначальных обещаний BDD заключалось в том, что непрограммисты (возможно, представители бизнеса или клиентов) смогут писать на языке Gherkin. Я сомневаюсь в реальности такой ситуации, но это не умаляет другие потенциальные выгоды от использования BDD.

Некоторые предварительные выводы

Я только слегка коснулся мира BDD, поэтому не решаюсь делать какие-то окончательные выводы. Правда, я нахожу принудительное структурирование функциональных тестов на шаги весьма привлекательным. Пожалуй, эта методология имеет потенциал в стимулировании многоразового использования в программном коде ФТ. Она аккуратно разделяет компетенции между описанием истории и ее реализацией и вынуждает нас думать о вещах с точки зрения предметной области бизнеса, а не с точки зрения того, «что нам нужно делать на основе Selenium».

Однако бесплатных обедов не бывает. Синтаксис Gherkin строг по сравнению с общей свободой, которую дают локальные комментарии ФТ.

Также я хотел бы видеть, как BDD масштабируется, когда у вас не просто один-два компонента и четыре-пять шагов, а нескольких десятков компонентов и сотни строк программного кода для шагов.

В целом могу сказать, что исследовать эту возможность определенно стоит и я, пожалуй, применю BDD в следующем персональном проекте.

Хочу выразить свою благодарность Дэниелу Поупу, Рэчел Уиллмер и Джареду Контрасеру за их отзывы к этой главе.

Выводы относительно BDD

Поощряет структурированный, многоразовый тестовый код

Разделяя компетенции (разбивая функциональные тесты на человекочитаемый файл компонентов на языке Gherkin и отдельную реализацию шаговых функций), BDD стимулирует написание более управляемого и многократно используемого тестового кода.

Может осуществляться за счет удобочитаемости

Синтаксис языка Gherkin, несмотря на все попытки быть человекочитаемым, в конечном счете накладывает ограничение на естественный язык, поэтому он не может схватывать нюансы и намерения так же хорошо, как это делают комментарии.

Попробуйте! Я точно попробую

Неустанно продолжаю повторять, что не использовал BDD в реальном проекте, поэтому вы должны принимать мои слова с большой осторожностью. Но я хотел бы оказать ему сердечную поддержку. Я собираюсь испытать его в следующем возможном проекте и призываю вас сделать то же самое.

Приложение F

Создание REST API: JSON, Ajax и имитирование на JavaScript

Передача состояния представления (REST) – это подход к разработке веб-службы, который позволяет пользователю получать и обновлять информацию о ресурсах. Он стал доминирующим при разработке API для использования в Веб.

Мы создали рабочее веб-приложение, которое до сих пор не нуждалось в прикладном программном интерфейсе (API). С чего вдруг он нам понадобился? Одна из причин может быть в том, чтобы улучшить опыт взаимодействия пользователей, сделав сайт динамичнее. Вместо того чтобы ожидать обновления страницы после каждого дополнения к списку, мы можем использовать JavaScript, чтобы выполнять асинхронные запросы к нашему API и тем самым давать пользователю ощущение большей интерактивности.

Еще интереснее, что после создания API мы можем взаимодействовать с нашим серверным приложением посредством других механизмов, не только через браузер. Одним из таких новых клиентских приложений могло бы стать мобильное или консольное приложение. Также другие разработчики могли бы создавать библиотеки и инструменты вокруг вашего серверного приложения.

В этом материале вы увидите, как создавать API вручную. Я предоставлю обзор того, как использовать популярный инструмент экосистемы Django под названием Django-Rest-Framework.

Наш подход для этого раздела

Пока я не стану конвертировать приложение в полном объеме. Мы будем исходить из предположения, что у нас есть существующий список. REST определяет связь между URL-адресами и HTTP-методами (GET и POST, а также более экстравагантными PUT и DELETE), которые будут нас направлять в процессе конструирования кода.

Статья в Википедии, посвященная REST[1], предлагает неплохой обзор. Вкратце:

- Наша новая структура URL будет */api/lists/{id}/*.
- GET будет предоставлять подробную информацию о списке (включая все его элементы) в формате JSON.
- POST позволяет добавлять элемент.

Мы возьмем программный код в том состоянии, в котором он был в конце главы 25.

Выбор подхода к тестированию

Если бы мы создавали API, который был бы совершенно нейтральным в отношении своих клиентов, то могли бы подумать, на каких уровнях его тестировать. Эквивалентом функциональных тестов мог бы стать запуск реального сервера (возможно, с использованием `LiveServerTestCase`) и взаимодействие с ним посредством библиотеки `requests`. Пришлось бы тщательно продумать, как устанавливать фикстуры (если мы используем непосредственно сам API, который привносит большое число зависимостей между тестами) и какой дополнительный уровень низкоуровневых/модульных тестов был бы для нас самым полезным. Либо мы можем решить, что достаточно единственного слоя тестов с использованием тестового клиента Django.

Мы создаем API в контексте клиентской стороны, опирающейся на браузер. Мы хотим начать его использовать на нашем производственном сайте, чтобы при этом приложение обеспечивало ту же функциональность, что и прежде. Поэтому наши функциональные тесты продолжат играть роль тестов высшего уровня, проверяя интеграцию между JavaScript и нашим API.

Это делает тестовый клиент Django естественным местом для размещения наших низкоуровневых тестов. Отсюда и начнем.

Организация базовой конвейерной передачи

Мы начинаем с модульного теста, который проверяет, что новая структура URL-адресов на GET-запросы возвращает отклик с кодом состояния 200 и что она использует формат JSON (вместо HTML).

lists/tests/test_api.py

```
import json
from django.test import TestCase
```

[1] См. https://en.wikipedia.org/wiki/Representational_state_transfer%23Relationship_between_URL_and_HTTP_methods

```
from lists.models import List, Item

class ListAPITest(TestCase):
    '''тест API списков'''
    base_url = '/api/lists/{}/'  ❶

    def test_get_returns_json_200(self):
        '''тест: возвращает json и код состояния 200'''
        list_ = List.objects.create()
        response = self.client.get(self.base_url.format(list_.id))
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response['content-type'], 'application/json')
```

❶ Шаблон с использованием константы уровня класса для тестируемого URL является новым элементом, который мы вводим для этого раздела. Он поможет удалять дублирование жестко кодированных URL-адресов. Можно даже применить функцию `reverse`, чтобы еще больше сократить дублирование.

Сначала подключим пару файлов *url*:

superlists/urls.py

```
from django.conf.urls import include, url
from accounts import urls as accounts_urls
from lists import views as list_views
from lists import api_urls
from lists import urls as list_urls

urlpatterns = [
    url(r'^$', list_views.home_page, name='home'),
    url(r'^lists/', include(list_urls)),
    url(r'^accounts/', include(accounts_urls)),
    url(r'^api/', include(api_urls)),
]
```

и

lists/api_urls.py

```
from django.conf.urls import url
from lists import api

urlpatterns = [
    url(r'^lists/(\d+)/$', api.list, name='api_list'),
]
```

Фактическое ядро нашего API может лежать в файле под названием *api.py*. Всего трех строк должно быть достаточно:

lists/api.py

```
from django.http import HttpResponse

def list(request, list_id):
    return HttpResponse(content_type='application/json')
```

Тесты должны пройти успешно. В результате мы собрали базовый конвейер.

```
$ python manage.py test lists
[...]
.............................................
 ---------------------------------------------------------------------
Ran 50 tests in 0.177s

OK
```

Получение фактического отклика

Следующий шаг – заставить API фактически откликаться, предоставляя какую-то информацию – в частности, содержимое элементов списков в формате JSON.

lists/tests/test_api.py (ch36l002)

```
def test_get_returns_items_for_correct_list(self):
    '''тест: получает отклик с элементами правильного списка'''
    other_list = List.objects.create()
    Item.objects.create(list=other_list, text='item 1')
    our_list = List.objects.create()
    item1 = Item.objects.create(list=our_list, text='item 1')
    item2 = Item.objects.create(list=our_list, text='item 2')
    response = self.client.get(self.base_url.format(our_list.id))
    self.assertEqual(
        json.loads(response.content.decode('utf8')),  ❶
        [
            {'id': item1.id, 'text': item1.text},
            {'id': item2.id, 'text': item2.text},
        ]
    )
```

❶ Главное, на что необходимо обратить внимание в этом тесте. Мы ожидаем, что отклик будет в формате JSON; мы используем `json.loads()`, потому что тестировать Python-овские объекты проще, чем копаться в неформатированных строках JSON.

И реализация, с другой стороны, использует `json.dumps()`:

lists/api.py

```
import json
from django.http import HttpResponse
from lists.models import List, Item

def list(request, list_id):
    list_ = List.objects.get(id=list_id)
    item_dicts = [
        {'id': item.id, 'text': item.text}
        for item in list_.item_set.all()
    ]
    return HttpResponse(
        json.dumps(item_dicts),
        content_type='application/json'
    )
```

Хорошая возможность применить генератор последовательности!

Добавление POST-метода

Второе, что нам нужно от API, – это способность добавлять новые элементы в список, используя запрос с методом POST. Начнем со «счастливого пути», то есть сценария по умолчанию:

lists/tests/test_api.py (ch36l004)

```
def test_POSTing_a_new_item(self):
    list_ = List.objects.create()
    response = self.client.post(
        self.base_url.format(list_.id),
        {'text': 'new item'},
    )
    self.assertEqual(response.status_code, 201)
    new_item = list_.item_set.get()
    self.assertEqual(new_item.text, 'new item')
```

И реализация столь же простая – в основном то же самое, что мы делали бы в нашем нормальном представлении, только здесь мы возвращаем код состояния 201, а не переадресацию:

lists/api.py (ch36l005)

```
def list(request, list_id):
    list_ = List.objects.get(id=list_id)
    if request.method == 'POST':
```

```
        Item.objects.create(list=list_, text=request.POST['text'])
        return HttpResponse(status=201)
    item_dicts = [
        [...]
```

Это позволит нам с чего-то начать.

```
$ python manage.py test lists
[...]
Ran 52 tests in 0.177s

OK
```

Одна из забавных вещей, касающаяся создания REST API, – вам приходится использовать еще нескольких кодов состояния HTTP из полного спектра. (См. https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

Тестирование клиентского Ajax при помощи Sinon.js

Даже не думайте заниматься тестированием Ajax без библиотеки объектов-имитаций. Различные тестовые инфраструктуры и инструменты имеют свои собственные библиотеки, а *Sinon* является универсальной. Как мы увидим позже, она также обеспечивает объекты-имитации на JavaScript...

Скачайте ее с сайта http://sinonjs.org/ и положите в папку *lists/static/tests/*.

Теперь мы можем написать наш первый тест Ajax:

lists/static/tests/tests.html (ch36l007)

```
<div id="qunit-fixture">
  <form>
    <input name="text" />
    <div class="has-error">Error text</div>
  </form>
  <table id="id_list_table">  ❶
  </table>
</div>

<script src="../jquery-3.1.1.min.js"></script>
<script src="../list.js"></script>
<script src="qunit-2.0.1.js"></script>
```

```
  <script src="sinon-1.17.6.js"></script>  ❷

  <script>
/* глобальный sinon */

var server;
QUnit.testStart(function () {
  server = sinon.fakeServer.create();  ❸
});
QUnit.testDone(function () {
  server.restore();  ❸
});

QUnit.test("errors should be hidden on keypress", function (assert) {
[...]

QUnit.test("should get items by ajax on initialize", function (assert) {
  var url = '/getitems/';
  window.Superlists.initialize(url);

  assert.equal(server.requests.length, 1);  ❹
  var request = server.requests[0];
  assert.equal(request.url, url);
  assert.equal(request.method, 'GET');
});

  </script>
```

❶ Добавляем новый элемент в фикстуру `div`, чтобы сформировать таблицу списка.

❷ Импортируем *sinon.js* (предварительно скачав и положив в соответствующую папку).

❸ Методы `testStart` и `testDone` – это эквиваленты QUnit для `setUp` и `tearDown`. Они используются, чтобы поручить Sinon запустить инструмент тестирования Ajax, `fakeServer` и сделать его доступным через переменную с глобальной областью видимости `server`.

❹ Это позволяет нам делать утверждения о любых Ajax-запросах, которые выполняются нашим кодом. Мы тестируем, на какой URL-адрес ушел запрос и какой метод HTTP он использовал.

Чтобы фактически выполнить наш Ajax-запрос, мы будем использовать Ajax-помощников jQuery[2]. Это намного проще, чем пытаться использовать низкоуровневые стандартные браузерные объекты `XMLHttpRequest`.

[2] См. https://api.jquery.com/jQuery.get/

lists/static/list.js

```
@@ -1,6 +1,10 @@
 window.Superlists = {};
-window.Superlists.initialize = function () {
+window.Superlists.initialize = function (url) {
    $('input[name="text"]').on('keypress', function () {
      $('.has-error').hide();
    });
+
+ $.get(url);
+
 };
+
```

Это должно привести к успешному прохождения теста:

```
5 assertions of 5 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
3. should get items by ajax on initialize (3)
```

Неплохо, мы могли бы отправить на сервер GET-запрос, но что если сделать что-то настоящее? Каким образом мы тестируем фактическую асинхронную часть, где мы имеем дело с (потенциальным) откликом?

Sinon и тестирование асинхронной части Ajax

Это главная причина полюбить библиотеку Sinon. Метод `server.respond()` позволяет точно контролировать поток асинхронного кода.

lists/static/tests/tests.html (ch36l009)

```
QUnit.test("should fill in lists table from ajax response", function (assert) {
  var url = '/getitems/';
  var responseData = [
    {'id': 101, 'text': 'item 1 text'},
    {'id': 102, 'text': 'item 2 text'},
  ];
  server.respondWith('GET', url, [
    200, {"Content-Type": "application/json"}, JSON.
stringify(responseData)  ❶
  ]);
  window.Superlists.initialize(url);  ❷

  server.respond();  ❸

  var rows = $('#id_list_table tr');  ❹
```

```
  assert.equal(rows.length, 2);
  var row1 = $('#id_list_table tr:first-child td');
  assert.equal(row1.text(), '1: item 1 text');
  var row2 = $('#id_list_table tr:last-child td');
  assert.equal(row2.text(), '2: item 2 text');
});
```

❶ Мы организуем немного данных отклика для использования в Sinon, сообщая ей, какой код состояния, какие заголовки и, что важно, какой JSON-отклик сервера нам нужно симулировать.

❷ Затем мы вызываем тестируемую функцию.

❸ Здесь таится волшебство. *Затем* мы можем вызвать `server.respond()`, когда захотим, и это запустит всю асинхронную часть цикла Ajax, то есть любую функцию обратного вызова, которую мы назначили для обработки отклика[3].

❹ Теперь можно незаметно проверить, заполнила ли Ajax-функция обратного вызова нашу таблицу новыми строками списка.

Реализация могла бы выглядеть примерно так:

lists/static/list.js (ch36l010)

```
if (url) {
  $.get(url).done(function (response) {
    var rows = '';
    for (var i=0; i<response.length; i++) {
      var item = response[i];
      rows += '\n<tr><td>' + (i+1) + ': ' + item.text + '</td></tr>';
    }
    $('#id_list_table').html(rows);
  });
}
```

Нам везет из-за того, каким образом jQuery регистрирует свои обратные вызовы для Ajax, когда мы используем функцию `.done()`. Если перейти на более удобный способ организации асинхронного кода в JavaScript, функцию (или так называемый промис) обратного вызова `.then()`, мы получим еще один уровень асинхронизации. QUnit действительно способен решить эту задачу. Обратитесь к документации по функции async (см. http://api.qunitjs.com/async/). Другие тестовые инфраструктуры имеют аналогичный функционал.

[3] Функция обратного вызова (callback) – функция, которая должна быть выполнена после того, как другая функция завершила выполнение (отсюда и название: коллбэк) – *Прим. перев.*

Соединение всего в шаблоне, чтобы убедиться, что это реально работает

Сначала мы нарушаем программный код, удалив из шаблона *lists.html* цикл таблицы списка `{% for %}`.

lists/templates/list.html

```
@@ -6,9 +6,6 @@

 {% block table %}
     <table id="id_list_table" class="table">
-        {% for item in list.item_set.all %}
-            <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
-        {% endfor %}
     </table>

     {% if list.owner %}
```

Это заставит один из модульных тестов не сработать. Все нормально, если этот тест пока удалить.

Мягкая деградация и прогрессивное улучшение

Удалив не-Ajax-версию страницы списков, я лишился возможности мягкой деградации (см. https://www.w3.org/wiki/Graceful_degradation_versus_progressive_enhancement), то есть сохранения версии сайта, которая будет по-прежнему работать без JavaScript.

Отсутствие мягкой деградации когда-то представляло проблему доступности: браузеры с экранным диктором для слабовидящих раньше не имели JavaScript, поэтому в случае полной опоры на JS эти пользователи исключались. Насколько я понимаю, сегодня это уже не представляет серьезной проблемы. Но некоторые пользователи блокируют JavaScript из соображений безопасности.

Другая типичная проблема состоит в отличающихся уровнях поддержки JavaScript в разных браузерах. Особое значение этот вопрос имеет, если вы начинаете искать приключений в направлении разработки современных клиентских приложений и JavaScript версии ES2015.

Короче говоря, всегда неплохо иметь резервную копию без Javascript. Особенно если вы создали сайт, который хорошо работает без него, не торопитесь выбрасывать рабочую «простую» HTML-версию. Я делаю это, только чтобы продемонстрировать.

Это приводит к тому, что наш базовый ФТ не проходит:

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
FAIL: test_can_start_a_list_for_one_user
[...]
  File "/.../superlists/functional_tests/test_simple_list_creation.py",
line 33, in test_can_start_a_list_for_one_user
    self.wait_for_row_in_list_table('1: Buy peacock feathers')
[...]
AssertionError: '1: Buy peacock feathers' not found in []
[...]
FAIL: test_multiple_users_can_start_lists_at_different_urls

FAILED (failures=2)
```

Давайте добавим блок `{% scripts %}` в базовый шаблон, который позже сможем выборочно переопределить на странице списков.

lists/templates/base.html

```
  <script src="/static/list.js"></script>

  {% block scripts %}
    <script>
$(document).ready(function () {
  window.Superlists.initialize();
});
    </script>
  {% endblock scripts %}

</body>
```

И теперь добавляем в *list.html* немного другой вызов `initialize` с правильным URL-адресом:

lists/templates/list.html (ch36l016)

```
{% block scripts %}
  <script>
$(document).ready(function () {
  var url = "{% url 'api_list' list.id %}";
  window.Superlists.initialize(url);
});
  </script>
{% endblock scripts %}
```

И знаете что? Тест проходит!

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
Ran 2 test in 11.730s

OK
```

Довольно хорошо для начала!

Теперь, если вы выполните все ФТ, то увидите, что у нас есть несколько неполадок в других ФТ, которыми придется заняться. Кроме того, мы используем старомодный POST-запрос из формы с обновлением страницы, так что пока еще не достигли новомодного хипстерского одностраничного приложения. Но до него доберемся!

Реализация Ajax POST-запроса, включая маркер CSRF

Сначала назначаем нашей форме со списком идентификатор `id`, чтобы ее было легко обнаруживать в JS:

lists/templates/base.html

```
@@ -56,7 +56,7 @@
           <div class="text-center”>
             <h1>{% block header_text %}{% endblock %}</h1>
             {% block list_form %}
-              <form method="POST" action="{% block form_action %}
{% endblock %}”>
+              <form id="id_item_form" method="POST" action="{% block
form_action %}{% endblock %}">
                 {{ form.text }}
                 {% csrf_token %}
                 {% if form.errors %}
```

Далее поправляем фикстуру в JS-тесте, чтобы отразить этот ID и маркер CSRF, которые в настоящее время находятся на странице.

lists/static/tests/tests.html

```
@@ -9,9 +9,14 @@
 <body>
    <div id="qunit"></div>
    <div id="qunit-fixture">
-     <form>
+     <form id="id_item_form">
```

```
        <input name="text" />
-       <div class="has-error">Error text</div>
+       <input type="hidden" name="csrfmiddlewaretoken" value="tokey" />
+       <div class="has-error">
+         <div class="help-block">
+           Error text
+         </div>
+       </div>
      </form>
```

И вот наш тест:

lists/static/tests/tests.html (ch36l019)

```
QUnit.test("should intercept form submit and do ajax post", function (assert) {
  var url = '/listitemsapi/';
  window.Superlists.initialize(url);

  $('#id_item_form input[name="text"]').val('user input');  ❶
  $('#id_item_form input[name="csrfmiddlewaretoken"]').val('tokeney');  ❶
  $('#id_item_form').submit();  ❶

  assert.equal(server.requests.length, 2);  ❷
  var request = server.requests[1];
  assert.equal(request.url, url);
  assert.equal(request.method, "POST");
  assert.equal(
    request.requestBody,
    'text=user+input&csrfmiddlewaretoken=tokeney'  ❸
  );
});
```

❶ Мы симулируем пользователя, заполняя форму и нажимая кнопку «Отправить».

❷ Ожидаем, что должен быть второй Ajax-запрос (первым был GET для таблицы элементов списка).

❸ Проверяем данные POST при помощи `requestBody`. Как видите, они представляют собой URL-кодированное значение, которое не очень легко протестировать, но его все же можно почти прочитать.

И вот как мы это реализуем:

lists/static/list.js

```
  [...]
    $('#id_list_table').html(rows);
```

```
  });

  var form = $('#id_item_form');
  form.on('submit', function(event) {
    event.preventDefault();
    $.post(url, {
      'text': form.find('input[name=”text”]').val(),
      'csrfmiddlewaretoken': form.find('input[name="csrfmiddlewaretoken"]').val(),
    });
  });
```

Это приводит к тому, что JS-тесты проходят, но одновременно нарушаются наши ФТ, потому что, хоть мы и справляемся с POST-запросом, мы не обновляем страницу после него, чтобы отобразить новый элемент списка:

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

Имитация в JavaScript

Нам нужно, чтобы клиентская сторона обновила таблицу элементов, после того как Ajax POST-запрос завершен. По сути, то же самое мы делаем, когда страница загружается, получая текущий список элементов из сервера и заполняя таблицу элементами.

Похоже, не помешает вспомогательная функция!

lists/static/list.js

```
window.Superlists = {};

window.Superlists.updateItems = function (url) {
  $.get(url).done(function (response) {
    var rows = '';
    for (var i=0; i<response.length; i++) {
      var item = response[i];
      rows += '\n<tr><td>' + (i+1) + ': ' + item.text + '</td></tr>';
    }
    $('#id_list_table').html(rows);
  });
};

window.Superlists.initialize = function (url) {
  $('input[name="text"]').on('keypress', function () {
    $('.has-error').hide();
```

```
  });

  if (url) {
    window.Superlists.updateItems(url);

    var form = $('#id_item_form');
    [...]
```

Это была всего-навсего рефакторизация, мы проверяем, что все JS-тесты по-прежнему проходят:

```
12 assertions of 12 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
3. should get items by ajax on initialize (3)
4. should fill in lists table from ajax response (3)
5. should intercept form submit and do ajax post (4)
```

Как теперь протестировать, что Ajax POST вызывает `updateItems` на успешном POST-запросе? Мы не хотим тупо копировать код, который симулирует отклик сервера и вручную проверяет таблицу элементов... А что насчет имитации?

Сначала организуем так называемую песочницу. Она будет отслеживать все имитации, которые мы создаем, и обеспечит снятие обезьяньих заплаток после каждого теста со всего того, что было сымитировано.

lists/static/tests/tests.html (ch36l023)

```
var server, sandbox;
QUnit.testStart(function () {
  server = sinon.fakeServer.create();
  sandbox = sinon.sandbox.create();
});
QUnit.testDone(function () {
  server.restore();
  sandbox.restore();  ❶
});
```

❶ Функция `.restore()` очень важна, она отменяет все имитации, которые мы создали в каждом тесте.

lists/static/tests/tests.html (ch36l024)

```
QUnit.test("should call updateItems after successful post", function (assert) {
  var url = '/listitemsapi/';
  window.Superlists.initialize(url);  ❶
  var response = [
```

```
    201,
    {"Content-Type": "application/json"},
    JSON.stringify({}),
  ];
  server.respondWith('POST', url, response);  ❶
  $('#id_item_form input[name="text"]').val('user input');
  $('#id_item_form input[name="csrfmiddlewaretoken"]').val('tokeney');
  $('#id_item_form').submit();

  sandbox.spy(window.Superlists, 'updateItems');  ❷
  server.respond();  2

  assert.equal(
    window.Superlists.updateItems.lastCall.args,  ❸
    url
  );
});
```

❶ Первое, на что следует обратить внимание: мы организуем отклик сервера, только *после* того, как выполним инициализацию. Мы хотим, чтобы это было откликом на POST-запрос, который происходит при отправке формы, а не на первоначальный GET-запрос. (Помните наш урок из главы 16? Одним из самых сложных элементом тестирования на JS является управление порядком выполнения.)

❷ Аналогичным образом мы начинаем имитировать вспомогательную функцию только после того, как произойдет первый вызов первоначального GET. Вызов `sandbox.spy` выполняет ту же работу, что и `patch` в тестах на Python. Он подменяет данный объект имитацией.

❸ Функция `updateltems` теперь увеличилась на несколько имитационных дополнительных атрибутов, `lastCall` и `lastCall.args`, которые похожи на `call_args` в имитациях на Python.

Чтобы этот тест прошел успешно, сначала делаем преднамеренную ошибку с целью проверить, что наши тесты действительно тестируют то, что, по нашим планам, должны тестировать:

lists/static/list.js

```
$.post(url, {
  'text': form.find('input[name="text"]').val(),
  'csrfmiddlewaretoken': form.find('input[name="csrfmiddlewaretoken"]').val(),
}).done(function () {
  window.Superlists.updateItems();
});
```

Так точно, мы почти у цели, но еще не совсем:

```
12 assertions of 13 passed, 1 failed.
[...]
6. should call updateItems after successful post (1, 0, 1)
    1. failed
        Expected: "/listitemsapi/"
        Result: []
        Diff: "/listitemsapi/"[]
        Source: file:///.../superlists/lists/static/tests/tests.html:124:15
```

И мы исправляем это таким образом:

lists/static/list.js

```
}).done(function () {
  window.Superlists.updateItems(url);
});
```

Наш ФТ проходит! По крайней мере один из них. У других есть проблемы, к ним мы вернемся чуть позже.

Завершение рефакторизации: приведение тестов в соответствие с кодом

Я не успокоюсь, пока мы не доведем до конца эту рефакторизацию и не приведем наши модульные тесты в чуть большее соответствие с программным кодом:

lists/static/tests/tests.html

```
@@ -50,9 +50,19 @@ QUnit.testDone(function () {
 });

-QUnit.test("should get items by ajax on initialize", function (assert) {
+QUnit.test("should call updateItems on initialize", function (assert) {
    var url = '/getitems/';
+   sandbox.spy(window.Superlists, 'updateItems');
    window.Superlists.initialize(url);
+   assert.equal(
+     window.Superlists.updateItems.lastCall.args,
+     url
+   );
+});
+
+QUnit.test("updateItems should get correct url by ajax", function (assert) {
+   var url = '/getitems/';
```

```
+  window.Superlists.updateItems(url);

   assert.equal(server.requests.length, 1);
   var request = server.requests[0];
@@ -60,7 +70,7 @@ QUnit.test("should get items by ajax on initialize",
function (assert) {
   assert.equal(request.method, 'GET');
 });

-QUnit.test("should fill in lists table from ajax response", function (assert) {
+QUnit.test("updateItems should fill in lists table from ajax response",
 function (assert) {
  var url = '/getitems/';
  var responseData = [
    {'id': 101, 'text': 'item 1 text'},
@@ -69,7 +79,7 @@ QUnit.test("should fill in lists table from ajax response",
function (assert) {
  server.respondWith('GET', url, [
    200, {“Content-Type”: "application/json"}, JSON.stringify(responseData)
  ]);
- window.Superlists.initialize(url);
+ window.Superlists.updateItems(url);

  server.respond();
```

С этим изменением тест покажет следующие результаты:

```
14 assertions of 14 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
3. should call updateItems on initialize (1)
4. updateItems should get correct url by ajax (3)
5. updateItems should fill in lists table from ajax response (3)
6. should intercept form submit and do ajax post (4)
7. should call updateItems after successful post (1)
```

Валидация данных. Упражнение для читателя

Если вы выполните полный прогон теста, то обнаружите, что два валидационных ФТ не проходят:

```
$ python manage.py test
[...]
ERROR: test_cannot_add_duplicate_items
(functional_tests.test_list_item_validation.ItemValidationTest)
[...]
```

```
ERROR: test_error_messages_are_cleared_on_input
(functional_tests.test_list_item_validation.ItemValidationTest)
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to
locate element: .has-error
```

Я не буду расписывать все подробно, дам по крайней мере модульные тесты, которые вам понадобятся:

lists/tests/test_api.py (ch36l027)

```
from lists.forms import DUPLICATE_ITEM_ERROR, EMPTY_ITEM_ERROR
[...]

    def post_empty_input(self):
        list_ = List.objects.create()
        return self.client.post(
            self.base_url.format(list_.id),
            data={'text': ''}
        )

    def test_for_invalid_input_nothing_saved_to_db(self):
        self.post_empty_input()
        self.assertEqual(Item.objects.count(), 0)

    def test_for_invalid_input_returns_error_code(self):
        response = self.post_empty_input()
        self.assertEqual(response.status_code, 400)
        self.assertEqual(
            json.loads(response.content.decode('utf8')),
            {'error': EMPTY_ITEM_ERROR}
        )

    def test_duplicate_items_error(self):
        list_ = List.objects.create()
        self.client.post(self.base_url.format(list_.id), data={'text':
'thing'})
        response = self.client.post(self.base_url.format(list_.id),
data={'text': 'thing'})
        self.assertEqual(response.status_code, 400)
        self.assertEqual(
            json.loads(response.content.decode('utf8')),
            {'error': DUPLICATE_ITEM_ERROR}
        )
```

И на стороне js:

lists/static/tests/tests.html (ch36l029-2)

```
QUnit.test("should display errors on post failure", function (assert) {
  var url = '/listitemsapi/';
  window.Superlists.initialize(url);
  server.respondWith('POST', url, [
    400,
    {"Content-Type": "application/json"},
    JSON.stringify({'error': 'something is amiss'})
  ]);
  $('.has-error').hide();

  $('#id_item_form').submit();
  server.respond(); // post

  assert.equal($('.has-error').is(':visible'), true);
  assert.equal($('.has-error .help-block').text(), 'something is amiss');
});
QUnit.test("should hide errors on post success", function (assert) {
    [...]
```

Вам понадобится несколько модификаций в *base.html*, чтобы сделать его совместимым при отображении ошибок Django (которые домашняя страница пока использует по-прежнему) и ошибок из JavaScript:

lists/templates/base.html (ch36l031)

```
@@ -51,17 +51,21 @@
          <div class="col-md-6 col-md-offset-3 jumbotron">
            <div class="text-center">
              <h1>{% block header_text %}{% endblock %}</h1>
+
              {% block list_form %}
               <form id="id_item_form" method="POST" action="{% block form_
action %}{% endblock %}">
                 {{ form.text }}
                 {% csrf_token %}
-                {% if form.errors %}
-                  <div class="form-group has-error">
-                    <div class="help-block">{{ form.text.errors }}</div>
+                <div class="form-group has-error">
+                  <div class="help-block">
+                    {% if form.errors %}
+                      {{ form.text.errors }}
+                    {% endif %}
                   </div>
```

```
-                 {% endif %}
+               </div>
              </form>
            {% endblock %}
+
          </div>
        </div>
      </div>
```

В итоге вы должны прийти к прогону js-теста, который будет выглядеть примерно так:

```
20 assertions of 20 passed, 0 failed.
1. errors should be hidden on keypress (1)
2. errors aren't hidden if there is no keypress (1)
3. should call updateItems on initialize (1)
4. updateItems should get correct url by ajax (3)
5. updateItems should fill in lists table from ajax response (3)
6. should intercept form submit and do ajax post (4)
7. should call updateItems after successful post (1)
8. should not intercept form submit if no api url passed in (1)
9. should display errors on post failure (2)
10. should hide errors on post success (1)
11. should display generic error if no error json (2)
```

И полный прогон теста должен пройти успешно, включая все ФТ:

```
$ python manage.py test
[...]
Ran 81 tests in 62.029s

OK
```

Пр-р-ревосходно[4]!

Это будет вашим собранным вручную REST API на основе Django. Если вы нуждаетесь в подсказке по поводу самостоятельного завершения финальной стадии, обратитесь к репозиторию.

Но я бы никогда не предложил создавать REST API в Django без ознакомления с инфраструктурой *Django-Rest-Framework*. Это тема следующего дополнительного материала! Продолжайте читать, Макдафф.

[4] Этому восклицанию вы можете придать свою собственную интонацию.

Советы по REST API

Дедупликация URL-адресов

URL-адреса для API играют в некотором смысле более важную роль, чем в браузерном приложении. Попытайтесь уменьшить число раз, когда вы их жестко кодируете в своих тестах.

Не работайте с неформатированными строками JSON

Вашими друзьями являются функции json.loads и json.dumps.

Всегда пользуйтесь библиотекой объектов-имитаций Ajax для своих тестов на JS

Sinon сойдет. Jasmine имеет свои собственные, впрочем, как и Angular.

Учитывайте мягкую деградацию и прогрессивное улучшение

Особенно если вы переходите со статического сайта на более JavaScript-управляемый, рассмотрите возможность оставить как минимум ядро функциональности вашего сайта без JavaScript.

Приложение G

Django-Rest-Framework

В предыдущей главе мы собрали собственный REST API. Пришла пора взглянуть на инфраструктуру Django-Rest-Framework (DRF)[1], которая является дежурным выбором для многих разработчиков на Python/Django, создающих API. Точно так же, как программная инфстраструктура Django, которая призвана предоставить вам все основные инструменты, которые потребуются для создания веб-сайта, управляемого базой данных (ORM, шаблоны и т. д.), DRF имеет целью дать инструменты, требуемые для создания API, и тем самым избавить вас от необходимости снова и снова писать стереотипный код.

При написании этого материала я главным образом боролся за то, чтобы получить тот же самый API, который только что реализовал вручную с целью его репликации на основе DRF. Получить ту же самую схему URL и ту же самую структуру данных JSON, которые я определил ранее, оказалось настоящей проблемой, и я почувствовал, что боролся с программной инфраструктурой.

Это всегда предупредительный знак. Разработчики Django-Rest-Framework намного умнее меня, и видели намного больше реализаций REST API, чем я. И если у них сложилось представление, как все должно выглядеть, то, возможно, имеет смысл адаптироваться и работать с их представлением о мире, вместо того чтобы навязывать свое собственное предвзятое мнение.

«Не идите против инфраструктуры» – один из величайших советов, которые я слышал. Либо двигайтесь вместе с потоком, либо сделайте переоценку, хотите ли вы находиться в рамках этой инфраструктуры вообще.

Мы будем работать исходя из API, который у нас был в конце предыдущей главы, и убедимся, что мы сможем его переписать для использования DRF.

Инсталляция

Быстрая команда `pip install` доставит нам DRF. Я использую версию 3.5.4, которая была последней в момент написания данного материала.

1 См. http://www.django-rest-framework.org/

```
$ pip install djangorestframework
```

Добавляем `rest_framework` в раздел `INSTALLED_APPS` в *settings.py*:

superlists/settings.py

```
INSTALLED_APPS = [
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'lists',
    'accounts',
    'functional_tests',
    'rest_framework',
]
```

Сериализаторы (на самом деле – объекты ModelSerializer)

Учебное руководство Django-Rest-Framework – довольно хороший ресурс для изучения DRF. Первое, с чем вы столкнетесь, это сериализаторы и в частности в нашем случае – сериализаторы моделей (ModelSerializers). Они представляют собой то, как в DRF принято конвертировать модели баз данных Django в JSON (или, возможно, другие форматы), который вы можете отправлять по проводам.

lists/api.py (ch37l003)

```
from lists.models import List, Item
[...]
from rest_framework import routers, serializers, viewsets

class ItemSerializer(serializers.ModelSerializer):

    class Meta:
        model = Item
        fields = ('id', 'text')

class ListSerializer(serializers.ModelSerializer):
    items = ItemSerializer(many=True, source='item_set')

    class Meta:
```

```
        model = List
        fields = ('id', 'items',)
```

Объекты Viewset (на самом деле – объекты ModelViewset) и объекты Router

ModelViewSet представляют собой то, как в DRF принято определять разнообразные способы взаимодействия с объектами для определенной модели через API. Сообщите ей, в каких моделях вы заинтересованы (через атрибут `queryset`) и как их сериализовать (`serializer_class`) – она сделает все остальное: автоматически создаст представления, которые позволят вам перечислять, получать, обновлять и даже удалять объекты.

Вот что нам нужно сделать, чтобы набор ViewSet смог получить элементы конкретного списка:

lists/api.py (ch37l004)

```
class ListViewSet(viewsets.ModelViewSet):
    queryset = List.objects.all()
    serializer_class = ListSerializer

router = routers.SimpleRouter()
router.register(r'lists', ListViewSet)
```

Маршрутизатор (router) – это то, как в DRF принято автоматически создавать конфигурацию URL-адресов и отображать их на функциональность, обеспечиваемую за счет ViewSet.

В этой точке мы можем начать указывать наш *urls.py* на новый маршрутизатор, обойдя старый код API и посмотрев, как наши тесты поживают в новой обстановке:

superlists/urls.py (ch37l005)

```
[...]
# from lists.api import urls as api_urls
from lists.api import router

urlpatterns = [
    url(r'^$', list_views.home_page, name='home'),
    url(r'^lists/', include(list_urls)),
    url(r'^accounts/', include(accounts_urls)),
    # url(r'^api/', include(api_urls)),
    url(r'^api/', include(router.urls)),
]
```

Это приводит к тому, что уйма наших тестов не срабатывают:

```
$ python manage.py test lists
[...]
django.urls.exceptions.NoReverseMatch: Reverse for 'api_list' not found.
'api_list' is not a valid view function or pattern name.
[...]
AssertionError: 405 != 400
[...]
AssertionError: {'id': 2, 'items': [{'id': 2, 'text': 'item 1'}, {'id': 3,
'text': 'item 2'}]} != [{'id': 2, 'text': 'item 1'}, {'id': 3, 'text': 'item
2'}]
 ----------------------------------------------------------------------
Ran 54 tests in 0.243s

FAILED (failures=4, errors=10)
```

Давайте сначала посмотрим на эти 10 ошибок, которые говорят, что не могут инвертировать `api_list`. Все потому, что маршрутизатор DRF использует способ именования в URL-адресах, отличный от того, который мы использовали, когда программировали его вручную. По результатам обратной трассировки вы увидите, что ошибки появляются, когда выводится шаблон в качестве HTML. Это *list.html*. Мы можем это исправить всего в одном месте; `api_list` становится `list-detail`:

lists/templates/list.html (ch37l006)

```
  <script>
$(document).ready(function () {
  var url = "{% url 'list-detail' list.id %}";
});
  </script>
```

Это приведет нас всего к четырем неполадкам:

```
$ python manage.py test lists
[...]
FAIL: test_POSTing_a_new_item (lists.tests.test_api.ListAPITest)
[...]
FAIL: test_duplicate_items_error (lists.tests.test_api.ListAPITest)
[...]
FAIL: test_for_invalid_input_returns_error_code
(lists.tests.test_api.ListAPITest)
[...]
FAIL: test_get_returns_items_for_correct_list
(lists.tests.test_api.ListAPITest)
[...]
FAILED (failures=4)
```

Давайте закомментируем все валидационные тесты (добавив префикс DONT) и оставим эту сложность на потом:

lists/tests/test_api.py (ch37l007)

```
[...]
    def DONTtest_for_invalid_input_nothing_saved_to_db(self):
        [...]
    def DONTtest_for_invalid_input_returns_error_code(self):
        [...]
    def DONTtest_duplicate_items_error(self):
        [...]
```

И теперь у нас всего две неполадки.

```
FAIL: test_POSTing_a_new_item (lists.tests.test_api.ListAPITest)
[...]
    self.assertEqual(response.status_code, 201)
AssertionError: 405 != 201
[...]
FAIL: test_get_returns_items_for_correct_list
(lists.tests.test_api.ListAPITest)
[...]
AssertionError: {'id': 2, 'items': [{'id': 2, 'text': 'item 1'}, {'id': 3,
'text': 'item 2'}]} != [{'id': 2, 'text': 'item 1'}, {'id': 3, 'text': 'item
2'}]
[...]
FAILED (failures=2)
```

Сначала посмотрим на последнюю.

Принятая по умолчанию конфигурация DRF предоставляет структуру данных, слегка отличающуюся от той, которую мы создали вручную: выполнение GET-запроса для списка дает его ID и затем элементы списка внутри ключа items. Это означает небольшую модификацию в нашем модульном тесте, чтобы он снова проходил успешно:

lists/tests/test_api.py (ch37l008)

```
@@ -23,10 +23,10 @@ class ListAPITest(TestCase):

         response = self.client.get(self.base_url.format(our_list.id))
         self.assertEqual(
             json.loads(response.content.decode('utf8')),
-            [
+            {'id': our_list.id, 'items': [
                 {'id': item1.id, 'text': item1.text},
                 {'id': item2.id, 'text': item2.text},
```

```
-            ]
+            ]}
        )
```

Это GET-запрос, который служит для получения сортированных элементов списка (как мы увидим позже, у нас есть много чего еще совершенно задаром). Как насчет того, чтобы добавить новые элементы с помощью POST-запроса?

Другой URL для элемента с POST-запросом

В этой точке я бросил бороться с инфраструктурой и увидел, где DRF хотела меня поймать. Выполнение POST-запроса к списочному ViewSet, чтобы добавить элемент в список, – возможный, но довольно муторный вариант.

Самый простой выход – сделать POST-запрос к представлению элемента, а не к представлению списка:

lists/api.py (ch37l009)

```
class ItemViewSet(viewsets.ModelViewSet):
    serializer_class = ItemSerializer
    queryset = Item.objects.all()

[...]
router.register(r'items', ItemViewSet)
```

А это значит, что мы слегка изменяем тест, перемещая все POST-тесты из `ListAPITest` в новый тестовый класс – `ItemsAPITest`:

lists/tests/test_api.py (ch37l010)

```
@@ -1,3 +1,4 @@
 import json
+from django.core.urlresolvers import reverse
 from django.test import TestCase
 from lists.models import List, Item
@@ -31,9 +32,13 @@ class ListAPITest(TestCase):
+
+class ItemsAPITest(TestCase):
+    base_url = reverse('item-list')
+
     def test_POSTing_a_new_item(self):
         list_ = List.objects.create()
         response = self.client.post(
-            self.base_url.format(list_.id),
-            {'text': 'new item'},
+            self.base_url,
```

```
+            {'list': list_.id, 'text': 'new item'},
         )
         self.assertEqual(response.status_code, 201)
```

Это даст нам

```
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

И так будет до тех пор, пока мы не добавим ID списка в нашу сериализацию элементов, иначе мы не знаем, для чего этот список:

lists/api.py (ch37l011)

```
class ItemSerializer(serializers.ModelSerializer):

    class Meta:
        model = Item
        fields = ('id', 'list', 'text')
```

Это вызывает еще одно небольшое связанное с ним изменение в тесте.

lists/tests/test_api.py (ch37l012)

```
@@ -25,8 +25,8 @@ class ListAPITest(TestCase):
         self.assertEqual(
             json.loads(response.content.decode('utf8')),
             {'id': our_list.id, 'items': [
-                {'id': item1.id, 'text': item1.text},
-                {'id': item2.id, 'text': item2.text},
+                {'id': item1.id, 'list': our_list.id, 'text': item1.text},
+                {'id': item2.id, 'list': our_list.id, 'text': item2.text},
             ]}
         )
```

Адаптирование на стороне клиента

Наш API больше не возвращает плоский массив элементов в списке. Он возвращает объект с атрибутом `.items`, который представляет элементы. Это означает небольшую поправку в нашей функции `updateItems`:

lists/static/list.js (ch37l013)

```
@@ -3,8 +3,8 @@ window.Superlists = {};
 window.Superlists.updateItems = function (url) {
   $.get(url).done(function (response) {
     var rows = '';
-    for (var i=0; i<response.length; i++) {
-      var item = response[i];
```

```
+    for (var i=0; i<response.items.length; i++) {
+      var item = response.items[i];
       rows += '\n<tr><td>' + (i+1) + ': ' + item.text + '</td></tr>';
     }
     $('#id_list_table').html(rows);
```

Ввиду того, что мы используем разные URL-адреса для получения списков методом GET и для отправки элементов методом POST, слегка подправим функцию `initialize`. Вместо многочисленных аргументов перейдем на использование объекта `params`, содержащего требуемую конфигурацию:

lists/static/list.js

```
@@ -11,23 +11,24 @@ window.Superlists.updateItems = function (url) {
   });
 };
-window.Superlists.initialize = function (url) {
+window.Superlists.initialize = function (params) {
   $('input[name="text"]').on('keypress', function () {
     $('.has-error').hide();
   });

- if (url) {
-   window.Superlists.updateItems(url);
+ if (params) {
+   window.Superlists.updateItems(params.listApiUrl);

    var form = $('#id_item_form');
    form.on('submit', function(event) {
      event.preventDefault();
-     $.post(url, {
+     $.post(params.itemsApiUrl, {
+       'list': params.listId,
        'text': form.find('input[name="text"]').val(),
        'csrfmiddlewaretoken': form.find('input[name="csrfmiddlewaretoken"]').
val(),
      }).done(function () {
        $('.has-error').hide();
-       window.Superlists.updateItems(url);
+       window.Superlists.updateItems(params.listApiUrl);
      }).fail(function (xhr) {
        $('.has-error').show();
        if (xhr.responseJSON && xhr.responseJSON.error) {
```

Мы отражаем это в *list.html*:

lists/templates/list.html (ch37l014)

```
$(document).ready(function () {
  window.Superlists.initialize({
    listApiUrl: "{% url 'list-detail' list.id %}",
    itemsApiUrl: "{% url 'item-list' %}",
    listId: {{ list.id }},
  });
});
```

Фактически этого достаточно, чтобы базовый ФТ снова заработал:

```
$ python manage.py test functional_tests.test_simple_list_creation
[...]
Ran 2 tests in 15.635s

OK
```

Предстоит еще внести дополнительные изменения в обработку ошибок. Если вам любопытно, можете исследовать этот вопрос сами, обратившись в репозиторий[2].

Что дает инфраструктура Django-Rest-Framework

Возможно, вы хотели бы знать, в чем был смысл использования этой инфраструктуры.

Конфигурация вместо кода

В сущности, первое преимущество заключается в том, что я трансформировал старую процедурную функцию представления в более декларативный синтаксис:

lists/api.py

```
def list(request, list_id):
    list_ = List.objects.get(id=list_id)
    if request.method == 'POST':
        form = ExistingListItemForm(for_list=list_, data=request.POST)
        if form.is_valid():
            form.save()
            return HttpResponse(status=201)
        else:
            return HttpResponse(
               json.dumps({'error': form.errors['text'][0]}),
               content_type='application/json',
```

[2] См. https://github.com/hjwp/book-example/blob/appendix_DjangoRestFramework/lists/api.py

```
                status=400
            )
    item_dicts = [
        {'id': item.id, 'text': item.text}
        for item in list_.item_set.all()
    ]
    return HttpResponse(
        json.dumps(item_dicts),
        content_type='application/json'
    )
```

Если сравнитваь это с финальной DRF-версией, то обратите внимание, что фактически теперь мы полностью сконфигурировали:

lists/api.py

```
class ItemSerializer(serializers.ModelSerializer):
    text = serializers.CharField(
        allow_blank=False, error_messages={'blank': EMPTY_ITEM_ERROR}
    )

    class Meta:
        model = Item
        fields = ('id', 'list', 'text')
        validators = [
            UniqueTogetherValidator(
                queryset=Item.objects.all(),
                fields=('list', 'text'),
                message=DUPLICATE_ITEM_ERROR
            )
        ]

class ListSerializer(serializers.ModelSerializer):
    items = ItemSerializer(many=True, source='item_set')

    class Meta:
        model = List
        fields = ('id', 'items',)

class ListViewSet(viewsets.ModelViewSet):
    queryset = List.objects.all()
    serializer_class = ListSerializer

class ItemViewSet(viewsets.ModelViewSet):
    serializer_class = ItemSerializer
```

```
    queryset = Item.objects.all()

router = routers.SimpleRouter()
router.register(r'lists', ListViewSet)
router.register(r'items', ItemViewSet)
```

Бесплатная функциональность

Второе преимущество заключается в том, что, используя ModelSerializer, ViewSet и маршрутизаторы DRF, я пришел к намного более многофункциональному API, чем тот, который был собран вручную.

- Все методы HTTP: GET, POST, PUT, PATCH, DELETE и методы OPTIONS теперь работают, что называется, из коробки, для всех списочных и элементных URL-адресов.
- И доступная для просмотра/самодокументированная версия API находится по адресу http://local host:8000/api/lists/ и http://localhost:8000/api/items. (рис. G.1; попробуйте!)

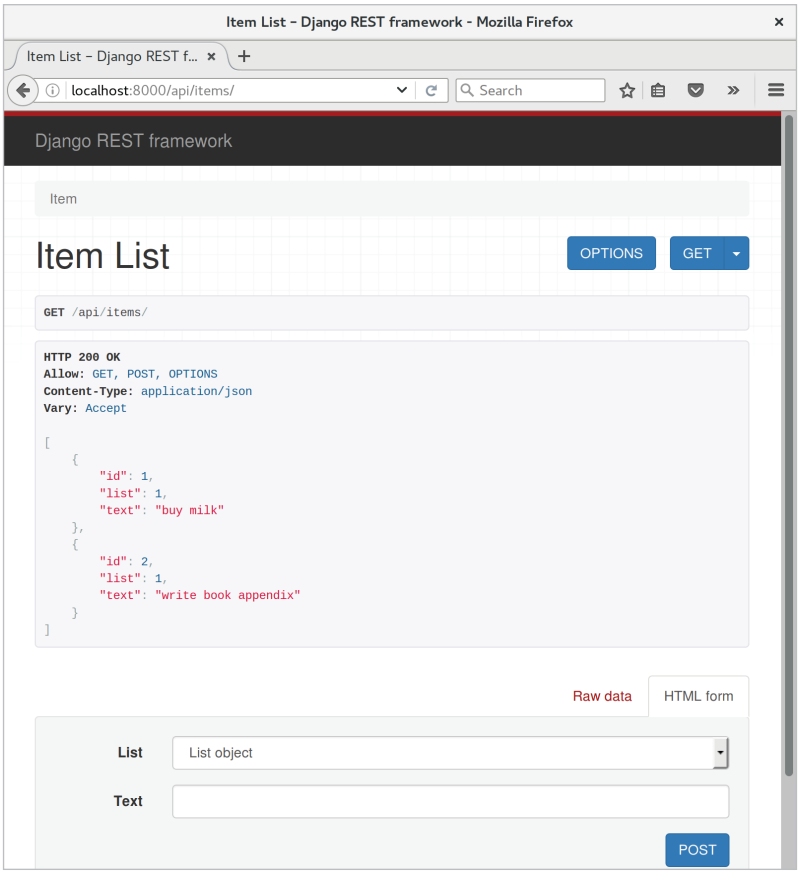


Рис. G.1. Бесплатный доступный для просмотра API для ваших пользователей

Более подробная информация имеется в документации по DRF[3], но обе эти крутые функциональные возможности вполне можно предложить конечным пользователям вашего API.

В общем, DRF является отличным способом почти автоматической генерации прикладных программных интерфейсов (API), которая основана на существующей структуре моделей. Если вы используете Django, то непременно попробуйте его до того, как начнете собирать свой собственный программный код API.

Советы относительно Django-Rest-Framework

Не боритесь с инфраструктурой

Движение в потоке часто является лучшим способом сохранить продуктивность. Иными словами: или откажитесь от использования инфраструктуры, или используйте ее на более низком уровне.

Объекты Router и Viewset для принципа наименьшей неожиданности

Одно из преимуществ DRF – его универсальные инструменты, такие как объекты Router и объекты Viewset. Они дадут вам очень предсказуемый API с разумными стандартными настройками для его конечных точек, структуры URL и откликов для разных методов HTTP.

Обратитесь к самодокументированной, доступной для просмотра версии

Проверьте свои конечные точки API в браузере. DRF откликается по-другому, когда обнаруживает, что ваш API адресуется «нормальным» веб-браузером и выводит на экран весьма симпатичную, самодокументированную версию самого себя, которой вы можете поделиться с пользователями.

[3] См. http://www.django-rest-framework.org/topics/documenting-your-api/%23self-describing-apis

Приложение H

Шпаргалка

По многочисленным просьбам эта шпаргалка отчасти основана на небольших резюме/сводках в конце каждой главы. Смысл ее – предоставить несколько напоминаний и ссылок на главы, где можно узнать больше, освежить свою память. Надеюсь, она будет вам полезной!

Начальная настройка проекта

- Начните с *истории пользователя* и отобразите ее на первый функциональный тест.
- Выберите тестовую платформу. Сойдет `unittest`, другие варианты – `py.test`, `nose` или `Green` – также могут предложить некоторые преимущества.
- Выполните функциональный тест и посмотрите на первую *ожидаемую неполадку*.
- Выберите для веб-разработки программную инфраструктуру Django и узнайте, как выполнять относительно нее *модульные тесты*.
- Создайте свой первый *модульный тест*, чтобы решить текущую неполадку ФТ и увидеть, что он не срабатывает.
- Выполните первую фиксацию в системе управления версиями (VCS), такой как Git.

Соответствующие главы: 1, 2 и 3.

Основной поток операций TDD

- Двойной цикл разработки на основе тестирования (TDD) (рис. H.1).
- Красный, зеленый, рефакторизуй.
- Триангуляция.
- Ведение блокнота.
- Клюнуло трижды – рефакторизуй.
- Переход от рабочего состояния к рабочему состоянию.
- Вам это не понадобится! (YAGNI)

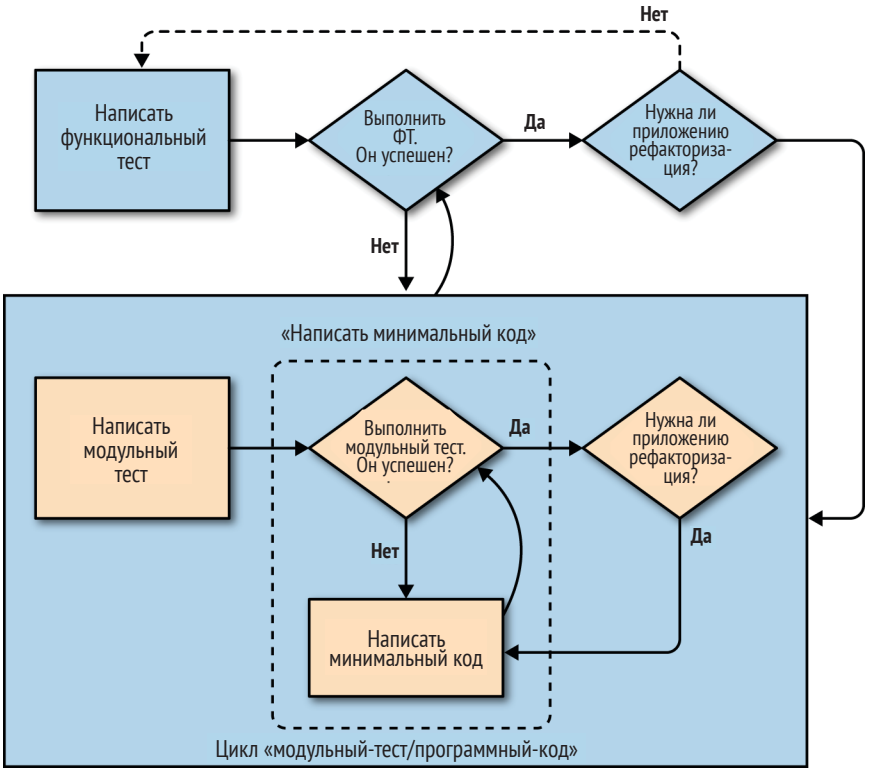


Рис. Н.1. Процесс TDD с функциональными и модульными тестами

Соответствующие главы: 4, 5 и 7.

Выход за пределы тестирования только на сервере разработки

- Начните раннее тестирование системы. Обеспечьте слаженную работу компонентов: веб-сервер, статическое наполнение, база данных.
- Создайте промежуточную среду (среду подготовки), которая соответствует вашей производственной (эксплуатационной) среде, и выполните свой комплект ФТ относительно нее.
- Автоматизируйте промежуточную и производственную среды:
  a) Инфраструктура как услуга (PaaS) против виртуального выделенного сервера (VPS);
  b) Fabric;
  c) Управление конфигурацией (Chef, Puppet, Salt, Ansible);
  d) Vagrant.

- Продумайте болевые точки развертывания: база данных, статические файлы, зависимости, как индивидуализировать настройки и т. д.
- Создайте CI-сервер как можно раньше, чтобы не опираться на самодисциплину слежения за выполнением тестирования.

Соответствующие главы: 9, 11, 24, приложение C.

Общие приемы тестирования

- Каждый тест должен тестировать один компонент.
- Один тестовый файл на каждый файл с исходным кодом приложения.
- Как минимум подумайте о тесте-заготовке для каждой функции и класса, каким бы простыми они ни были.
- Не тестируйте константы.
- Старайтесь тестировать поведение, а не реализацию.
- Старайтесь выходить за пределы заколдованного круга через программный код и обдумывать граничные и ошибочные случаи.

Соответствующие главы: 4, 13 и 14.

Лучшие приемы на основе Selenium / функциональных тестов

- Применяйте явные ожидания вместо неявных и схему «взаимодействие/ожидание».
- Избегайте дублирования тестирующего программного кода – вспомогательные методы в базовом классе либо страничный шаблон проектирования (Page Pattern) являются одним из способов решения этой проблемы.
- Избегайте функциональности двойного тестирования. Если у вас есть тест, который охватывает длительный процесс (например, вход в систему), рассмотрите способы пропустить его в других тестах (но учитывайте неожиданные взаимодействия между не связанными на вид фрагментами функциональности).
- Обратитесь к инструментам разработки на основе поведения (BDD) как к еще одному способу структурирования ФТ.

Соответствующие главы: 21, 24 и 25.

«Снаружи внутрь», изоляция тестов против интегрированных тестов и имитация

Помните о причинах, почему мы сначала пишем тесты:

- Гарантировать правильность и предотвратить регрессию.
- Обеспечить написание чистого и удобного в сопровождении кода.
- Задействовать быстрый, продуктивный поток операций.

С учетом этих целевых установок подумайте о различных типах тестов и компромиссах между ними:

Функциональные тесты

- Обеспечивают лучшую гарантию, что приложение будет работать правильно с точки зрения пользователя.
- Но имеют более медленный цикл обратной связи.
- Не всегда помогают написать чистый код.

Интегрированные тесты (зависят, например, от объектно-реляционного преобразователя (ORM) или тестового клиента Django)

- Быстро пишутся.
- Просты для понимания.
- Предупреждают о любых проблемах интеграции.
- Но не всегда вычленяют хорошую структуру кода (обязанность остается на вас!).
- Обычно медленнее, чем изолированные тесты.

Изолированные тесты (с имитациями или mock-объектами)

- Более трудоемкие.
- Их сложнее прочесть и понять.
- Являются самым лучшим вариантом для достижения самой оптимальной структуры кода.
- Выполняются быстрее всех.

Если вы пишете тесты с большим количеством объектов-имитаций и они кажутся вам вымученными, вспомните принцип «слушать свои тесты» – уродливые, тесты с имитациями подсказывают, что ваш программный код можно упростить.

Соответствующие главы: 22, 23 и 26.

Приложение I

Что делать дальше

Здесь я выскажу несколько предложений о том, как развить свои навыки тестирования и применить их к некоторым крутым новым технологиям в области веб-разработки (являвшимся таковыми на момент написания книги).

Я надеюсь превратить каждое из этих предложений как минимум в своего рода пост в блоге, а то и в будущий дополнительный материал к книге. Также со временем надеюсь для всех них подготовить примеры программного кода. Обращайтесь на http://www.obeythetestinggoat.com по поводу возможных обновлений.

Или почему бы вам не попытаться меня опередить и не создать свой собственный пост с хронологическим описанием вашей попытки воплотить любой из них?

Буду рад ответить на вопросы, дать советы и подсказки по всем этим темам. Так что если вы попытаетесь решить одну из них и зайдете в тупик – пожалуйста, не стесняйтесь, напишите мне: obeythetestinggoat@gmail.com!

Уведомления – на сайте и по электронной почте

Было бы неплохо, если бы пользователи получали уведомления, когда кто-то делится с ними списком.

Вы можете использовать уведомления Django, чтобы показывать сообщение в следующий раз, когда они обновляют экран. Для этого в вашем ФТ потребуется два браузера.

Также вы могли бы отправлять уведомления по электронной почте. Исследуйте возможности Django по проверке почты. Затем вы можете принять решение, что это критически важно и вам нужны реальные тесты с реальными электронными письмами. Используйте библиотеку IMAPClient для выборки фактических электронных писем из тестовой учетной записи веб-почты.

Переходите на Postgres

SQLite представляет собой замечательную небольшую СУБД, но она не сможет в достаточной мере справляться, как запросы вашего сайта будут принимать более одного рабочего веб-процесса. СУБД Postgres сегодня является всеобщим любимцем, поэтому выясните, как ее установить и сконфигурировать.

Нужно определиться с местом хранения имен пользователей и паролей для локального, промежуточного и производственного серверов Postgres. Поскольку в целях безопасности вы, вероятно, не захотите хранить их в репозитории исходного кода, измените ваши сценарии развертывания так, чтобы передавать их из командной строки. Популярное решение вопроса с местом их хранения – переменные окружения.

Проведите эксперимент: продолжая выполнять модульные тесты вместе с SQLite, сравните, насколько они быстрее, чем их выполнение относительно Postgres. Настройте систему так, чтобы ваша локальная машина использовала SQLite для тестирования, а CI-сервер использовал Postgres.

Выполняйте тесты относительно разных браузеров

Selenium поддерживает самые разные браузеры, включая Chrome и Internet Explorer. Попробуйте оба и убедитесь, что поведение вашего комплекта ФТ отличается.

Вам также необходимо попробовать «бездисплейный» браузер PhantomJS.

На моем опыте переход на другой браузер часто помогает вскрыть разного рода ошибки ситуации состязания в тестах на основе Selenium. Скорее всего, вам придется намного чаще применять схему «взаимодействие/ожидание» (особенно при использовании PhantomJS).

Тесты на коды состояния 404 и 500

Профессиональный сайт должен иметь красивые страницы ошибок. Тестирование страницы с кодом состояния 404 не представляет труда, но для тестирования страниц с кодом 500 вам, вероятно, потребуется индивидуализировать представление «поднимать исключения умышленно».

Сайт администратора Django

Представьте ситуацию, когда пользователь просит предоставить ему анонимный список. Для этого администратор сайта вручную изменяет запись, использующую сайт администратора Django.

Узнайте, как включить сайт администратора, и поэкспериментируйте с ним. Напишите ФТ, который показывает, как нормальный, не зарегистрировавшийся в системе пользователь создает список, затем пользователь-администратор входит в систему, переходит на сайт администратора и назначает список пользователю. Затем пользователь видит его на своей странице «Мои списки».

Напишите несколько тестов защищенности

Подробно остановитесь на входе в систему, «Моих списках» и обмене списками – что нужно написать, чтобы убедиться, что пользователи могут делать только то, что они авторизованы делать?

Тест на мягкую деградацию

Что произойдет, если сервер Persona «ляжет»? Сможем ли мы показать нашим пользователям хотя бы сообщение об ошибке с извинениями?

- Совет: один из способов симулировать падение сервера Persona – подправить файл *hosts* (на узлах */etc/* или *c:\Windows\Sytem32\drivers\etc*). Не забудьте отменить изменения в тестовой функции `tearDown`!
- Подумайте о серверной и о клиентской сторонах.

Кэширование и тестирование и производительности

Узнайте, как установить и сконфигурировать `memcached` – ПО, которое реализует службу кэширования данных в оперативной памяти. Узнайте, как использовать утилиту тестирования производительности `ab` разработки Apache. Какова производительность с кэшированием и без него? Можно ли написать автоматизированный тест, который перестанет работать, если кэширование не будет включено? Что насчет ужасающей проблемы аннулирования кэша? Смогут ли тесты помочь вам обеспечить несокрушимость программной логики аннулирования кэша?

MVC-инфраструктуры для JavaScript

Библиотеки JavaScript, которые позволяют реализовывать шаблон проектирования «Модель-Представление-Контроллер» на клиентской стороне, – последний писк моды. Списки неотложных дел являются для них одним из любимых демонстрационных приложений, так что не составит труда конвертировать сайт в одностраничный, где все дополнения в список происходят на JavaScript.

Выберите библиотеку (возможно, Backbone.js или Angular.js) и выполните импульсную реализацию. Каждая библиотека имеет свои собственные предпочтения по поводу того, как писать модульные тесты. Изучите тот способ, который к ней прилагается, и посмотрите, устраивает ли он вас.

Async и протокол Websocket

Предположим, два пользователя работают с одним списком одновременно. Разве не круто увидеть обновления в реальном времени – когда один человек добавляет элемент к списку, а другой его сразу видит? Реализовать постоянное соединение между клиентом и сервером позволяет использование протокола веб-сокетов.

Попробуйте один из Python-овских асинхронных веб-серверов – Tornado, gevent, Twisted – и убедитесь, что их можно использовать для реализации динамических уведомлений.

Для тестирования вам понадобятся два экземпляра браузера (как для тестирования обмена списками). Проверьте, что уведомления о действиях в одном экземпляре появляются в другом без обновления страницы.

Перейдите на использование py.test

Программная инфраструктура *py.test* позволяет писать модульные тесты с меньшим объемом стереотипного кода. Попытайтесь конвертировать несколько своих модульных тестов для использования *py.test*. Возможно, понадобится плагин, чтобы заставить его слаженно работать с Django.

Попробуйте coverage.py

Утилита *coverage.py* Неда Бэтчелда (Ned Batchelder) сообщит вам о покрытии вашего кода тестами, то есть какой его процент покрыт тестами. Теоретически, теперь у нас всегда должно быть 100%-ное покрытие, потому что мы использовали строгую методологию TDD. Но хочется знать наверняка. Этот инструмент также очень полезен для работы с проектами, в которых с самого начала тесты не использовались.

Шифрование на стороне клиента

Вот забавная ситуация: что, если наши пользователи параноидально настроены в отношении Агентства национальной безопасности (NSA) и больше не доверяют свои списки «облаку»? Реально ли создать систему шифрования на JavaScript, где пользователь может вводить пароль, чтобы шифровать свой текст элемента в списке перед его отправкой на сервер?

Один из способов протестировать это мог бы состоять в том, чтобы существовал некий пользователь-администратор, который переходит к представлению администратора Django, чтобы пронспектировать списки пользователей, и проверяет, что они сохранены в базе данных в зашифрованном виде.

Здесь место для ваших предложений

Как вы думаете, что еще нужно здесь разместить? Ваши предложения, пожалуйста!

Приложение J

Примеры исходного кода

Все примеры программного кода, которые я использовал в книге, доступны в моем репозитории на GitHub. Поэтому, если вы захотите сравнить свой код с моим, можете найти его там.

Каждая глава имеет свою ветку с именем по названию главы, как тут:

Глава 1

https://github.com/hjwp/book-example/tree/chapter_01

Имейте в виду, что каждая ветка содержит все фиксации (комиты) для данной главы, поэтому ее состояние представляет программный код в конце главы.

Полный список ссылок для каждой главы

Глава 1

https://github.com/hjwp/book-example/tree/chapter_01

Глава 2

https://github.com/hjwp/book-example/tree/chapter_02_unittest

Глава 3

https://github.com/hjwp/book-example/tree/chapter_unit_test_first_view

Глава 4

https://github.com/hjwp/book-example/tree/chapter_philosophy_and_refactoring

Глава 5

https://github.com/hjwp/book-example/tree/chapter_post_and_database

Глава 6

https://github.com/hjwp/book-example/tree/chapter_explicit_waits_1

Глава 7

https://github.com/hjwp/book-example/tree/chapter_working_incrementally

Глава 8

https://github.com/hjwp/book-example/tree/chapter_prettification

Глава 9

https://github.com/hjwp/book-example/tree/chapter_manual_deployment

Глава 10

https://github.com/hjwp/book-example/tree/chapter_making_deployment_production_ready

Глава 11

https://github.com/hjwp/book-example/tree/chapter_automate_deployment_with_fabric

Глава 12

https://github.com/hjwp/book-example/tree/chapter_organising_test_files

Глава 13

https://github.com/hjwp/book-example/tree/chapter_database_layer_validation

Глава 14

https://github.com/hjwp/book-example/tree/chapter_simple_form

Глава 15

https://github.com/hjwp/book-example/tree/chapter_advanced_forms

Глава 16

https://github.com/hjwp/book-example/tree/chapter_javascript

Глава 17

https://github.com/hjwp/book-example/tree/chapter_deploying_validation

Глава 18

https://github.com/hjwp/book-example/tree/chapter_spiking_custom_auth

Глава 19

https://github.com/hjwp/book-example/tree/chapter_mocking

Глава 20

https://github.com/hjwp/book-example/tree/chapter_fixtures_and_wait_decorator

Глава 21

https://github.com/hjwp/book-example/tree/chapter_server_side_debugging

Глава 22

https://github.com/hjwp/book-example/tree/chapter_outside_in

Глава 23

https://github.com/hjwp/book-example/tree/chapter_purist_unit_tests

Глава 24

https://github.com/hjwp/book-example/tree/chapter_CI

Глава 25

https://github.com/hjwp/book-example/tree/chapter_page_pattern

Приложение B

https://github.com/hjwp/book-example/tree/appendix_Django_Class-Based_Views

Приложение E

https://github.com/hjwp/book-example/tree/appendix_bdd

Приложение F

https://github.com/hjwp/book-example/tree/appendix_rest_api

Приложение G

https://github.com/hjwp/book-example/tree/appendix_DjangoRest-Framework

Использование Git для проверки вашего прогресса

Если вы намерены развить свои способности Git-Fu чуть больше, можете добавить мой репозиторий как *удаленный*:

```
git remote add harry https://github.com/hjwp/book-example.git
git fetch harry
```

И затем сверить свою разницу в конце главы 4:

```
git diff harry/chapter_philosophy_and_refactoring
```

Git справляется с многочисленными удаленными репозиториями, поэтому вы по-прежнему можете это делать, даже если уже публикуете свой код на GitHub или Bitbucket.

Имейте в виду, что точный порядок, скажем, методов в классе может различаться в наших версиях. Это может затруднить чтение разниц.

Скачивание ZIP-файла для главы

Если по какой-то причине вы хотите в определенной главе «начать с нуля», или пропустить[1], или вам неудобно с Git – можете скачать версию моего исходного кода в виде ZIP-файла с URL-адресов согласно следующей схеме:

> https://github.com/hjwp/book-example/archive/chapter_01.zip
> https://github.com/hjwp/book-example/archive/chapter_philosophy_and_refactoring.zip

Не позволяйте этому превращаться в костыль!

Попытайтесь не заглядывать в ответы, если только вы действительно не попали в тупике. Как я уже говорил в начале последней главы, гораздо важнее отлаживать ошибки самому. Ведь реальной ситуации не будет никаких «репозиториев Гарри», с которыми можно свериться и в которых можно найти все ответы.

[1] Я не рекомендую пропускать. Я скомпоновал главы так, что они не являются независимыми; каждая глава опирается на предыдущие, так что она может оказаться запутаннее, чем что-либо еще…

Предметный указатель

A

ALLOWED_HOSTS 216
angular.js 333
Ansible 236, 539
API форм 275. См. валидация данных формы
A-записи 201

B

BDD. См. разработка на основе поведения (BDD)
Behave 550
Bootstrap
  большие поля ввода 185
  документация 177, 258
  интеграция 180
  класс jumbotron 184
  скачивание 176
  стилистическое оформление таблицы 185

C

call_args, свойство 387
Chutzpah 318
collectstatic, команда 187
console.log 326
CSS (Каскадные таблицы стилей)
  CSS-платформы 176
  сложности, связанные со статическими файлами 173
  создание и применение 185
Cucumber 550

D

django-allauth 340
django-crispy-forms 277
django-floppyforms 277
Django-Rest-Framework (DRF) 587
  ModelSerializer 588
  ModelViewset 589
  POST-запросы 592
  адаптирование на стороне клиента 593
  инсталляция 587
  преимущества 595
  советы 598
  учебные руководства 587
Django, программная инфраструктура
  выполнение функциональных и/или модульных тестов 122
  документация 385
  и PythonAnywhere 525
  инсталляция 22
  инфраструктура сообщений 371
  команды и понятия
    python functional_tests.py 67
    python manage.py runserver 67
    python manage.py test 67
    python manage.py test functional_tests 123
    python manage.py test lists 123
    цикл "модульный-тест/программный-код" 67
  модульное тестирование 53
  наследование шаблонов 178
  настройка 34
    создание проекта 37, 599
  обобщенные представления на основе классов 312, 528
  объектно-реляционный преобразователь (ORM) 100
  отправка электронных писем 341, 362, 414
  статические файлы 182
  структура программного кода 53
  тестовый клиент 79
  учебные руководства 22, 101
DRY. См. не повторяйся, Dont Repeat Yourself (DRY)
dumpdata, команда 409

F

Fabric
  выполнение на промежуточном сайте 230
  документация 231
  дополнительные ресурсы 236
  инсталляция и настройка 224
  использование напрямую из Python 421
  конфигурирование 231
  лучшие приемы автоматизированного развертывания 237

перемещение развертывания в Ansible 542
сценарий развертывания 225
factory_boy 405
Firefox
и PythonAnywhere 523
инсталляция 26
обновление 123
преимущества 23
form_valid 530
f-string, синтаксическая конструкция 21
full_clean, метод 257

G

Geckodriver
инсталляция 26
обновление 123
get_absolute_url 270
get_user, метод 382
GET-запросы 281
Gherkin 550
Git
diff -b 180
reset --hard 175
конфигурирование 25
локальные переменные 203
назначение тегов релизам 235
обнаружение перемещенных файлов 122
перемещение файлов 119
скачивание 24
создание веток 340
создание репозиториев 40
фиксации (комиты) 40, 50
Gmail 343
Green 318
grep, команда 282
Gunicorn
автоматическая начальная загрузка/перезагрузка 217
добавление в requirements.txt 218
конфигурирование с использованием sed 234
настройка журналирования 411
переход на него 212
переход на сокеты домена Unix 215
преимущества 222

H

HTML
GET-запросы 281
POST-запросы
обработка 92
переадресация 108, 163
создание 89
сохранение 105
схема обработки в Django 263
дампы снимков экрана 489, 497
учебные руководства 22
HTML5 291, 294

J

Jasmine 318
Jenkins
инсталляция 479
конфигурирование 479
настройка виртуального дисплея 487
настройка проекта 484
первая сборка 485
тесты QUnit на JavaScript 318
увеличение тайм-аута 492
Jest 318
jQuery 320
jsUnit 318
jumbotron, класс (Bootstrap) 184

K

Karma 318

L

lambda, функции 248
Lettuce 550
LiveServerTestCase, класс 50
loaddata, команда 409

M

MacOS 25
mail.outbox 362
Meta, класс 301
Mocha 318
ModelForm, класс 277

N

Nginx
  инсталляция 198
  конфигурирование 206
  конфигурирование при помощи sed 234
  переход на сокеты домена Unix 215
  подтверждение операции 201
  раздача статических файлов 214
  устранение неполадок 208
nose 318
NoSuchElementException 125

O

Oauth 338
Openid 338

P

patch, декоратор 390, 398
PhantomJS 494
POST-запросы
  обработка 92
  переадресация 108, 163
  создание 88, 118, 130, 172
  сохранение 105
  схема обработки в Django 263
pytest 318
Python 3
  property, декоратор 442
  with, оператор 256
  библиотека Mock 367, 397
  инсталляция и настройка
    инсталляция в MacOS 25
    инсталляция в Windows 21
    инсталляция и активация virtualenv 26
    на промежуточных сайтах 200
  лямбда-функции 248
  ознакомительные книги 21
  против Python 2 21
PythonAnywhere 197, 523
python-social-auth 340

Q

Qunit 318
QUnit 318, 332, 494

R

React 333
requirements.txt 204, 218
response.context 166
RSpec 333

S

sed, потоковый редактор 234
Selenium
  и JavaScript 333
  и PythonAnywhere 523
  инсталляция 28
  лучшие приемы непрерывной интеграции 497
  обновление 123
  тестирование взаимодействия пользователя 71
self.assertRaises, контекстный менеджер 255
self.browser.refresh() 184
self.wait_for, вспомогательный метод 250, 253, 405
send_mail, функция 341
socket.error
  [WinError 10054] 184
StaleElementException 125
StaticLiveServerTestCase 183
stderr 350
superlists, приложение 38
Systemd 217

T

TDD "снаружи-внутрь"
  внешний уровень 431
  недостатки 443
  определение 443
  против подхода "изнутри-наружу" 433
  управляемая ФТ разработка 433
  уровень контроллера 431
  уровень модели 438
  уровень представлений 437
TDD с подходом снаружи-внутрь 427
time.sleep 89, 124

U

unittest 318
unittest, модуль
  и модуль mock 367

V

Vagrant 543

W

wait_for_row_in_list_table, вспомогательный метод 125
wait_for, вспомогательный метод 253
wait_to_be_logged_in/out, методы 405
Windows
  поддержка Gunicorn 218
  советы 24
with, оператор 256

X

Xvfb 523

Y

YAGNI, You Aint Gonna Need It (Вам это не понадобится!) 132
YUI 318

А

автоматизированное развертывание. См. Fabric
  дополнительные ресурсы 236
  лучшие приемы 236
  подготовка 219
  преимущества 192
архитектурные решения 516
атрибуты mail.outbox 362
аутентификация 338
  беспарольная 338
  и данные cookie 401
  индивидуализированная авторизация в Django 346
  индивидуализированные модели аутентификации 345
  минимальная индивидуализированная модель пользователя 354
  модель маркера для соединения электронных адресов с уникальным идентификатором 358
  на стороне клиента в пользовательском интерфейсе 341
  отправка электронных писем из Django SendingEmailsfromDj 341
  предотвращение секретов в исходном коде 344
  пропуск регистрации в системе в функциональных тестах 399
  строковые маркеры в базах данных 344

Б

библиотеки для тестирования 318
Билли-тестировщик
  определение 34
  переход из рабочего состояния в рабочее состояние 130, 156
  философия 521
блокнот с рабочим списком неотложных дел 117

В

валидация. См. валидация данных формы; См. валидация на уровне модели
валидация данных формы
  использование собственного метода save формы 295
  использование форм в представлениях 281
  лучшие приемы 297
  недопущение повторяющихся элементов 298
  обработка POST-запросов 285
  обработка POST- и GET-запросов 288
  перемещение валидационной логики из формы 274
  преимущества 274, 295
валидация на уровне модели
  вывод на поверхность ошибок в представлении 257
  выполнение полной валидации 256
  контекстный менеджер self.assertRaises 255
  обработка POST-запросов 263
  предотвращение повторяющихся элементов 299
  преимущества и недостатки 255, 273
  удаление жестко кодированных URL-адресов 269
валидация уникальных значений 308. См. тестирование на наличие дулирующихся элементов

взаимодействия пользователя
  валидация вводимых данных на уровне базы данных 254
  валидация данных формы 274
  недопущение повторяющихся элементов 298
  тестирование ввода в базу данных 88
  тестирование при помощи Selenium 71
видеообучение 30
виртуальная среда (virtualenv)
  инсталляция и настройка 26
  на основе сервера 192
  создание вручную 204
виртуальные дисплеи 487
внешние зависимости 362, 398
вопросы безопасности и ее настройка
  ALLOWED_HOSTS 216
  безопасность сервера 223
  подделка межсайтовых запросов 91
  системы регистрации 397
вспомогательные методы 250, 289, 405

Г

генераторное выражение 73
генератор последовательности 73
гибкая динамика 131
глобальное состояние 326, 333

Д

данные cookie 228
декораторы
  patch 390
  property 442
  wait 405
  преимущества 408
документирование 357
доменные имена 196
дублирования, уменьшение 408, 501
дублирования, устранение 97, 385, 398

Ж

жадные регулярные выражения 163
журналирование 350, 411, 426

И

идемпотентность 237
изолированные тесты 510
изоляция, обеспечение
  в функциональных тестах 118
  использование имитаций 446
  преимущества и недостатки 444, 478
  пример несработавшего теста 444
  против интегрированных тестов 475
  рефакторизация уродливых тестов 449
  удаление избыточного программного кода 472
  уровень представления 450
  уровень форм 457
имитации
  mock_auth, переменная 390
  mock.return_value 388
  библиотека Mock языка Python 367, 397
  внедрение индивидуализированной аутентификации 378
  в ручном режиме 363
  изолирование тестов 438, 446
  подготовка 362
  практическое применение 394
  преимущества и недостатки 362, 478
  ссылка на выход из системы 396
  устранение дублирования 385
  функциональный тест 392
импульсное исследование и внедрение его результатов
  ветвление вашей системы управления версий 340
  внедрение результатов 351, 378
  вывод результатов в stderr 350
  определение 338, 361
инструменты непрерывного развертывания 539
инструменты управления конфигурацией 237, 539
интеграционные тесты 510
интегрированная среда разработки (IDE) 23
интегрированные тесты
  архитектурные решения 516
  преимущества и недостатки 478, 519
  против изолированных тестов 475
  против модульных тестов 101, 510
история пользователя 51
итеративный стиль разработки 88

К

классы управления формой (Bootstrap) 184, 257
код с душком 98
комбинаторный взрыв 385
контекстный менеджер self.assertRaises 255
контекстный объект отклика 166
крупномасштабные конструктивные изменения 131

Л

локальные комментарии 563
лямбда-функции. См. lambda, функции

М

макет См. тестирование дизайна и макета; См. CSS
маломасштабные конструктивные изменения против крупномасштабных 130, 170
маркеры 344
миграции баз данных 102, 112, 209, 544
многосложные представления против тонких представлений 297
модульные тесты
  API форм 275
  JavaScript 324
  в Django
    модульное тестирование представления 62
    написание базового модульного теста 55, 56
    тестовые базы данных 119
    цикл "модульный-тест/программный-код" 64
  длина 105
  использование для разведочного программирования 64, 67, 97
  недостатки "чистых" модульных тестов 512
  правило "Не тестировать константы" 74
  преимущества "чистых" модульных тестов 511
  против интегрированных тестов 101, 510
  против функциональных тестов 53
  рефакторизация в модульные тесты 75, 81
  рефакторизация на несколько файлов 251
  тестирование только одной единицы кода 109, 297, 538
модульные тесты с единственным утверждением 538

Н

настройки DEBUG 216
недопустимые входные данные 261. См. валидация на уровне модели
необходимое программное обеспечение 23
неожиданные неполадки 94, 116
не повторяйся, Dont Repeat Yourself (DRY) 98
непрерывная интеграция (CI)
  дополнительные применения 497
  и промежуточный этап 497
  конфигурирование сервера Jenkins 481
  настройка виртуального дисплея 487
  настройка проекта 484
  первая сборка 485
  преимущества 479
  расширение тайм-аута 492
  сервер на выбор 487
  снимки экрана 489
  советы 497, 521
  тесты QUnit на JavaScript 494
  установка сервера Jenkins 479

О

обезьяньи заплатки 363, 398
обеспечение работы сервера 196, 210
обобщенные представления на основе классов
  лучшие приемы 537
  против представлений на основе простых классов 528
обобщенный вспомогательный метод явного ожидания 253
обобщенных представления на основе классов
  домашняя страница в качестве FormView 529
  дублирующиеся представления 533
  индивидуальная настройка CreateView 530
  ключевые тесты и утверждения 528
  сравнение старой и новой версий 536
обратный просмотр 167
объектно-реляционный преобразователь (ORM) 100, 458, 478
ожидаемые неполадки 51
операторы print 89
отладка
  Systemd 217

на стороне сервера
внедрение программного кода журналирования 424
использование промежуточных сайтов 410
тестирование почтовых сообщений POP3 414
управление тестовыми базами данных 418
установка секретных переменных окружения 413
обеспечение работы сервера 210
ручное посещение страницы 112
снимки экранов 489
страница DEBUG в Django 90
улучшение сообщений об ошибках 95
функциональных тестов 89
отчет об обратной трассировке 61
ошибки целостности данных 256, 273, 274

П

пароли 338
передача состояния представления (REST)
дополнительные ресурсы 565
и творческое вдохновление 132
определение 565
советы по REST API 585
создание REST API 565
переменные окружения 344, 413
переход из рабочего состояния в рабочее состояние 156, 170
платформа как услуга (PaaS) 197, 222
платформы на основе MVC 333
подделка межсайтовых запросов (CSRF) 91
получение справки 52, 198, 527, 603
постоянная переадресация (301) 163
правило "если клюнуло трижды, рефакторизуй" 98, 117, 268
правило "красный/зеленый/рефакторизуй" 97, 116, 253
правило "Не тестировать константы" 74
представления строковых значений 301
преобразование URL-адресов 59, 146, 269
примеры программного кода, получение и использование 551, 608
производственные базы данных 424, 425
промежуточные сайты
адаптирование функциональных тестов 193
доменные имена 196
и непрерывные интеграции 497
и фикстуры 426
локальные сессии и сессии на промежуточном сервере 422
отлавливание финальных дефектов 410
преимущества 192, 210
ручное обеспечение работы сервера 196
ручное развертывание кода 201
управление тестовыми базами данных 418
процесс регистрации, пропуск 399. См. аутентификация

Р

разведочное программирование 277. См. импульсное исследование и его внедрение
развертывание
автоматизация при помощи Fabric 224, 539
опасные области 192
процедура 334
тестирование с использованием промежуточных сайтов 191
развертывание готовое к эксплуатации
использование Gunicorn 212
использование Systemd для автоматической начальной загрузки/перезагрузки 217
лучшие практические методы 222
переход на сокеты домена Unix 215
подготовка для автоматизации 219
раздача статических файлов при помощи Nginx 214
разработка на основе поведения
преимущества и недостатки 564
разработка на основе поведения (BDD) 550
извлечение параметров в шагах 556
инструменты 550
написание структурированного тестового кода 561
определение 550
по сравнению с ФТ с локальными комментариями 559
против локальных комментариев 563
создание каталога 551
страничный шаблон проектирования 561

функциональный тест с использованием языка Gherkin 552
шаговые функции 553
разработка на основе тестирования (TDD)
будущие исследования 603
видеообучение 30
дополнительные ресурсы 52, 518
инкрементное адаптирование существующего программного кода 130, 172
необходимость 16, 68
необходимые предварительные знания 21
подход "снаружи-внутрь" 427
полный процесс 85, 133, 599
понятия
блокнот для рабочего списка неотложных дел 117
если клюнуло трижды, рефакторизуй 117, 268
история пользователя 51
красный/зеленый/рефакторизуй 97, 116, 253
неожиданные неполадки 94, 116
ожидаемые неполадки 51
регрессия 116
триангуляция 97, 116
цикл "модульный-тест/программный-код" 64
тестирование на JavaScipt 332
философия
YAGNI 131, 170
аналогия с ведром воды 69
переход из рабочего состояния в рабочее состояние 156, 170
разбивка работы на подзадачи 170
цели теста 514
регрессия 116, 134
регулярные выражения 163
рефакторизация 75, 81, 98, 251, 253, 449

C

сеансы, предварительное создание 399, 418
секретные значения 227, 413
серверы Linux 197
Символы
{% csrf_token %} 91
{% for .. in ..%} 111
@patch, декоратор 390
@property, декоратор 442
{% url %} 269
системные тесты 510
системы управления версиями (VCS) 23. См. Git
службы хостинга 197
снимки экрана 489, 497
сокеты домена Unix 212
сообщения об ошибках 89. См. устранение неполадок; См. устранение неполадок
сопутствующее видео 30
списковое включение. См. генератор последовательности
статические файлы
запросы URL 182
поиск 182
раздача при помощи Nginx 214, 222
сбор для развертывания 187
сложности 173, 192
стилистическое оформление таблицы (Bootstrap) 185
страничный шаблон проектирования 561
страничный шаблон проектирования (Page pattern)
практический опыт 506
преимущества 508
расширение функциональных тестов на дополнительных пользователей 504
сокращение дублирования 501
ФТ с многочисленными пользователями 499
сценарии, создание самостоятельных сценариев 418

T

тестирование базы данных
POST-запросы
обработка 92
переадресация 108
создание 89
сохранение 105
валидация на уровне базы данных 254
вывод в качестве HTML элементов в шаблоне 111
миграции 544
недопустиммый ввод 261

обеспечение сохранности производственных баз данных 424, 426
объектно-реляционный преобразователь (ORM) 100
правило есликлюнулотриждырефакторизуй ThreeStrikes 98
синтаксис шаблонов 93
создание производственной базы данных 112
управление тестовыми базами данных 418
тестирование дизайна и макета
CSS-платформы 176
инструменты Bootstrap 184
интеграция Bootstrap 180
лучшие приемы 190
наследование шаблонов в Django 178
отбор целей для тестирования 172
сбор статических файлов для развертывания 187
создание и применение CSS 185
тестирование лучших практических приемов 109, 128, 297, 514, 538, 601
тестирование многочисленных списков
URL-адреса элементов списка 146
добавление элементов в существующие списки 157
инкрементная реализация структуры кода 133
корректировки модели 151
маломасштабные конструктивные изменения против крупномасштабных 130
рефакторизация с использованием URL-включений 168
тест на регрессию 134
тестирование на JavaScript
jQuery и элемент div для фикстур 320
библиотеки тестирования 318, 333
в Jenkins вместе с PhantomJS 494
в цикле TDD 332
дополнительные ресурсы 316
дополнительные соображения 332
ключевые проблемы 326
модульные тесты 324
синтаксические ошибки 332
стереотипный код и организация пространства имен 330
управление глобальным состоянием 326, 333
функциональный тест 316
тестирование на наличие повторяющихся элементов
в представлении для отображения списка 310
сложные формы 308
функциональный тест 298
тестирование с проверкой на наличие повторяющихся элементов
на уровне представлений 307
тестовые файлы
организация и рефакторизация 252
разбиение модульных тестов на несколько файлов 251
тестовые фикстуры 404, 408
тестовый клиент (Django) 79
тесты в качестве документирования 357
тонкие представления против многосложных представлений 297
триангуляция 97, 116

У

упорядочение набора queryset 301
устранение неполадок
активация virtualenv 27
зависание функциональных тестов 123
работа Nginx 208
утиная типизация 442

Ф

фантазийное программирование 435
фикстуры
и промежуточный сайт 426
фикстуры JSON 404, 409
фикстуры JSON 404, 408
фикстуры для элемента div 320, 326
функциональное программирование 249, 406
функциональные тесты (ФТ)
JavaScript 316
вспомогательные методы 250
для объектов-имитаций 392
для повторяющихся элементов 298
и импульсный программный код 361
использование языка Gherkin 552

обеспечение изоляции 118, 444
отладка
методы 89
подход "снаружи-внутрь" 428
преимущества и недостатки 478
против модульных тестов 53
против приемочных и системных тестов 510
с многочисленными пользователями 499
создание 35
структурирование тестирующего программного кода 499, 561
устранение неполадок с зависшими тестами 123
явные/неявные ожидания и time.sleep 124
функция поиска и замены 282

Ш

шаблон проектирования "Модель-Представление-Контроллер (MVC)" 56, 132
шаблоны
иерархия наследования 433
и уровень представления 436
конструирование API с их использованием 434
наследование шаблонов в Django 178
передача переменных 93
синтакс 93
сохранение файлов конфигурации для обеспечения работы сервера 219
теги
{% csrf_token %} 91
{% for .. in ..%} 111
{% url %} 269
шпаргалка
выход за пределы тестирования только на сервере разработки 601
изолированные тесты против интегрированных тестов 602
настройка проекта 599
поток операций TDD 600
тестирование лучших приемов 601

Э

электронные письма, отправка из Django 341, 362, 414
элементы списка 96, 110, 146
эстетика, тестирование 173. См. тестирование дизайна и макета

Я

явные и неявные ожидания 124, 405

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. +7(499) 782-38-89

Электронный адрес: **books@alians-kniga.ru**.

Гарри Персиваль

Python. Разработка на основе тестирования

Повинуйся Билли-тестировщику, используя Django, Selenium и JavaScript

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Печать цифровая.
Усл. печ. л. 50,62. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

O'REILLY®

Проводя вас по процессу разработки реального веб-приложения от начала до конца, второе издание книги демонстрирует преимущества методологии разработки на основе тестирования (TDD) с использованием языка Python. Вы научитесь писать и выполнять тесты до написания любого фрагмента вашего приложения и затем разрабатывать минимальный объем программного кода, необходимого для прохождения этих тестов. В результате вы получите чистый программный код, который работает!

Также вы узнаете основы Django, Selenium, Git, jQuery и Mock. Если вы готовы поднять свои навыки программирования на Python на следующий уровень, то эта книга – обновленная до Python 3.6 – продемонстрирует вам, как методология TDD способствует созданию простой структуры кода и вселяет в вас уверенность в своих силах.

Вы сможете:

- окунуться в поток операций TDD, включая цикл «модульный тест/программный код» и рефакторизацию;
- использовать модульные тесты для классов и функций, а также функциональные тесты для взаимодействий пользователя внутри браузера;
- узнать, когда и как использовать mock-объекты, и аргументы «за» и «против» изолированных тестов по сравнению интегрированными тестами;
- тестировать и автоматизировать процесс развертывания на промежуточном сервере;
- применять тесты к сторонним плагинам, которые вы интегрируете в ваш сайт;
- выполнять тесты автоматически, применяя среду непрерывной интеграции;
- применять методологию TDD для создания REST API с клиентским интерфейсом на основе Ajax.

Гарри Дж. У. Персиваль (Harry J.W. Percival) *работает в PythonAnywhere LLP и распространяет евангелие методологии TDD по всему миру в беседах, на семинарах и конференциях со всей своей страстью и энтузиазмом новообращенного. Он является магистром в области Computer Science, Ливерпуль, и магистром философии Кэмбриджского университета.*

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
orders@alians-kniga.ru
Оптовая продажа:
"Альянс-книга"
тел.(499)782-38-89
books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-594-3