Бессерверные приложения на JavaScript

Преимущества облачных бессерверных веб-приложений бесспорны: меньшая сложность, быстрое продвижение на рынок и автоматическая масштабируемость выгодно отличают их от традиционных серверных проектов. А благодаря поддержке JavaScript в AWS Lambda и мощным новым бессерверным инструментам, таким как библиотека Claudia.js, вы можете создавать и развертывать бессерверные приложения, не изучая новый язык.

Данная книга научит вас проектировать и создавать бессерверные веб-приложения на AWS с использованием JavaScript, Node и Claudia.js. Вы овладеете основными навыками разработки функций AWS Lambda, а также шаблонами бессерверного программирования, такими как API Gateway. Попутно отточите свои новые навыки, создав действующий чат-бот и добавив поддержку голосового помощника Amazon Alexa. Вы также узнаете, как перенести существующие приложения на бессерверную платформу.

Краткое содержание:

- аутентификация и использование баз данных в бессерверных приложениях;
- асинхронные функции;
- разработка бессерверных микросервисов;
- разнообразные практические примеры.

«Помогает быстро организовать выполнение простых операций в AWS Lambda, не влияя на организацию и работу проектов.»
— из предисловия Гойко Аджича, Neuri Consulting

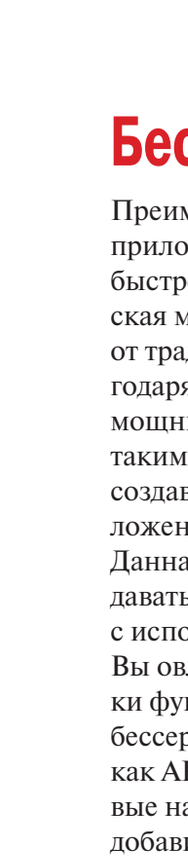
«Отличный источник практических знаний, написанный известными экспертами, который поможет вам в кратчайшие сроки освоить AWS Lambda с использованием Claudia.js.»
— Валентин Гретаз, Consulthys

«Одна из самых полных книг среди посвященных этой теме; содержит множество ссылок на ресурсы в интернете.»
— Дамиан Эстебан, BetterPT

С. Стоянович (S. Stojanovic) и **А. Симович (A. Simovic)** являются обладателями титула AWS Serverless Heroes и основными участниками проекта Claudia.js. Они вместе создали Desole – инструмент с открытым исходным кодом для трассировки ошибок в бессерверных приложениях – и являются ведущими разработчиками Claudia Bot Builder.

Для веб-разработчиков, знакомых с JavaScript и Node.js.

ISBN 978-5-97060-782-4



Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

www.дмк.рф

Бессерверные приложения на JavaScript

Бессерверные приложения на JavaScript

Слободан Стоянович
Александар Симович

MANNING

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Слободан Стоянович
Александар Симович

Бессерверные приложения на JavaScript

Serverless Applications with Node.js

Slobodan Stojanović
Aleksandar Simović

MANNING
SHELTER ISLAND

Бессерверные приложения на JavaScript

Слободан Стоянович
Александар Симович

Перевод с английского Киселева А. Н.

Москва, 2020

УДК 004.42
ББК 32.972
С81

С81 Слободан Стоянович, Александар Симович
Бессерверные приложения на JavaScript / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2020. – 394 с.: ил.

ISBN 978-5-97060-782-4

Преимущества облачных бессерверных веб-приложений бесспорны: меньшая сложность, быстрое продвижение на рынок и автоматическая масштабируемость выгодно отличают их от традиционных серверных проектов. Данная книга научит вас проектировать и создавать бессерверные веб-приложения на AWS с использованием JavaScript, Node и Claudia.js. Новичков издание знакомит не только с AWS Lambda, но и с целым рядом связанных служб, таких как DynamoDB, Cognito, API Gateway. Даже решив позднее взять на вооружение другие инструменты, вы сможете сохранить весь код и просто развернуть его немного иначе. Подробно описывается несколько вариантов практического использования бессерверных платформ, в том числе веб-API, чат-боты, обработка платежей и управление заказами.

Издание предназначено веб-разработчикам, знакомым с JavaScript и Node.js.

УДК 004.42
ББК 32.972

ISBN 978-1-61729-472-3 (англ.)
ISBN 978-5-97060-782-4 (рус.)

Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую предоставлять вам качественные материалы.

Предисловие

Amazon навсегда изменила IT-инфраструктуру, упростив подготовку виртуальных машин в 2007 году. После этого совершенствование архитектуры современных приложений носило постепенный характер. Спустя десятилетие, упростив возможность предоставления отдельных функций, платформа Amazon Lambda дала толчок новой волне глубинных перемен. Эта «бессерверная» экосистема кардинально меняет способы проектирования, разработки и эксплуатации интернет-приложений.

Как один из первых, кто начал использовать эту платформу на практике, я имел честь работать со Слободаном и Александаром и воочию убедиться, насколько сильно влияет «бессерверное» мышление на время выхода на рынок и стоимость эксплуатации. В то же время платформа развивается настолько быстро, что в ней легко запутаться. Чтобы получить настоящие преимущества нового способа работы, разработчики должны пересмотреть стратегии аутентификации, управления сеансами, хранения данных, планирования мощностей и распределения вычислений. В своей книге «Бессерверные приложения на JavaScript» Слободан и Александар предлагают первый отчет об этой революции и бесценное руководство для разработчиков на JavaScript, желающих воспользоваться преимуществами платформ нового поколения.

В этой книге мне понравилось, как она помогает быстро организовать выполнение простых операций в AWS Lambda, не влияя на организацию и работу проектов. Многие бессерверные фреймворки приложений абстрагируют службы AWS, что увеличивает риск замкнуться на одном фреймворке, так как экосистема продолжает быстро развиваться. Авторы не заставляют нас принять их выбор фреймворков, но объясняют, как легко использовать все связанные службы. Новичков в AWS эта книга знакомит не только с AWS Lambda, но и с целым рядом связанных служб, таких как DynamoDB (хранилище данных), Cognito (аутентификация), API Gateway (доступ к работающим веб-службам) и Cloudwatch (обработка и планирование событий). Даже решив позднее взять на вооружение другие инструменты, вы сможете сохранить весь код и просто развернуть его немного иначе.

Еще одна веская причина прочитать эту книгу – в ней описывается несколько вариантов практического использования бессерверных платформ, включая веб-API, чат-боты, обработку платежей и управление заказами. Постепенно создавая онлайн-магазин для вымышленной пиццерии, авторы представляют практически готовые компоненты, необходимые для запуска современных бизнес-сценариев в облаке. Этот способ постепенного формирования знаний позволяет авторам по мере обсуждения исследовать все более сложные вопросы разработки, такие как организация автоматического тестирования и разработка приложений с прицелом на простоту сопровождения. Последняя часть книги посвящена стратегиям миграции и отвечает на некоторые наиболее распространенные вопросы от людей, у которых уже есть приложения

на какой-либо другой облачной платформе и которые хотят быстро получить конкурентные преимущества, сократить время выхода на рынок или уменьшить стоимость эксплуатации.

Я надеюсь, что вы получите от этой книги столько же удовольствия, сколько и я, и обнаружите эффективные способы предоставления услуг с помощью программного обеспечения в облаке.

Гойко Аджич (Gojko Adzic),
партнер в Neuri Consulting LLP

Вступление

Мы оба были разработчиками более 10 лет, оба начинали с наших первых компьютеров в 90-х годах, когда писали первые функции на Pascal и BASIC и даже участвовали в соревнованиях по программированию. Но все изменилось, когда появился интернет. Мы сразу начали создавать свои веб-приложения и веб-страницы со статическими HTML и CSS. Когда JavaScript и jQuery превратились в новый стандарт, мы почти сразу переключились на них (правда, один из нас продолжил экспериментировать с Flash и ActionScript). С появлением Node.js мы естественно переключились на этот фреймворк с языков, на которых писали в ту пору, таких как Python и C#. Хотя иногда мы можем написать несколько функций на этих языках, наш переход на Node.js состоялся бесповоротно.

Примерно три года назад мы обратили наше внимание на бессерверные архитектуры. Гойко Аджич познакомил нас с AWS Lambda на примере своей разработки Claudia.js – инструмента развертывания. Мы были поражены, насколько просто и быстро разрабатывать и развертывать бессерверные приложения и насколько легко их масштабировать, и мы вместе с ним начали работать над созданием Claudia Bot Builder.

Изучение бессерверной архитектуры постепенно изменило наш взгляд на создание и поддержку веб-приложений. Внутренние службы сменили бессерверные функции, и, вместо того чтобы писать сценарии на bash, входить на наши серверы и распределять вычислительные мощности, мы перестали заботиться об этих проблемах и сосредоточились больше на бизнес-логике и ценности приложений.

Мы опубликовали наши первые бессерверные веб-приложения и разработали сотни чат-ботов. Наша продуктивность увеличилась почти в пять раз. Это было невероятно. Месяцы, потраченные на изучение настройки и обслуживания серверов приложений с помощью bash, ssh, rsync и т. д., потеряли свою важность. Все изменилось. С нашей точки зрения бессерверная экосистема прошла долгий путь: бессерверными услугами стало проще пользоваться, и с каждым годом становится все больше и больше компонентов для бессерверных приложений (с Amazon re:Invent).

Случилось так много и так быстро – мы сделали нашу карьеру бессерверной. Мы начали вести дискуссии о бессерверных архитектурах, проводить семинары и давать консультации. Мы попытались объединить наш опыт и знания из множества других источников и изложить их в удобном для изучения и понятном формате.

Благодарности

Работать над этой книгой было трудно, так как это наш первый опыт. Некоторые главы переписывались по пять и более раз, чтобы вам, уважаемый читатель, легче было понять и усвоить обсуждаемые в них сведения. Наши друзья и семьи оказывали нам большую поддержку в процессе, и мы хотели бы поблагодарить всех, кто помогал нам на этом пути.

Прежде всего мы хотели бы поблагодарить Гойко Аджича (Gojko Adzic). Он ввел нас в мир без серверов несколько лет назад. Отдельное спасибо за его комментарии к этой книге, такие как: «эта страница ничего не стоит, удалите ее», «не лгите своим читателям о шагах» и т. п. Они были бесценны для нас.

Мы хотели бы поблагодарить нашего редактора из издательства Manning Тони Арритола (Toni Arritola). Спасибо, что помогли нам выбраться из тупика, когда мы застряли на первых нескольких главах, что были терпеливы, когда мы отставали от графика, и поддерживали нас всем необходимым. Вы всегда требовали от нас высокого качества и тем помогали сделать книгу лучше для читателей. Также мы хотим поблагодарить Майкла Стивенса (Michael Stephens) и Берта Бейтса (Bert Bates), которые помогли лучше объяснить детали бессерверных архитектур и сосредоточиться на важных темах. Спасибо сотрудникам издательства Manning, работавшим над выпуском и продвижением книги, это была сплоченная команда. Спасибо техническому корректору Валентину Креттазу (Valentin Crettaz) и техническому редактору Костасу Пассадису (Kostas Passadis) за тщательный анализ кода.

Спасибо также рецензентам из Manning, которые нашли время, чтобы прочитать нашу рукопись на разных этапах, и дали ценные отзывы, в том числе: Арно Бейли (Arnaud Bailly), Барнаби Норман (Barnaby Norman), Клаудио Бернардо Родригес (Claudio Bernardo Rodríguez), Дамиан Эстебан (Damian Esteban), Ден Балия (Dane Balia), Дипак Бхаскаран (Deepak Bhaskaran), Джасба Симпсон (Jasba Simpson), Джереми Ланге (Jeremy Lange), Кай Стрем (Kaj Ström), Кэтлин Р. Эстрада (Kathleen R. Estrada), Кумар Унникришнан (Kumar Unnikrishnan), Лука Меццалира (Luca Mezzalira), Мартин Денерт (Martin Dehnert), Рами Абдельвахед (Rami Abdelwahed), Сурджит Манхас (Surjeet Manhas), Томас Пеклак (Thomas Peklak), Умур Йильмаз (Umur Yilmaz) и Ивон Вивилль (Yvon Vieville).

Спасибо сотрудникам Amazon и AWS, что создали такую потрясающую компьютерную службу: AWS Lambda. Ваши усилия меняют мир.

Наконец, спасибо тетушке Марии и всем другим вымышленным героям этой книги!

Об этой книге

Основная цель книги «*Бессерверные приложения на JavaScript*» – обучение и помощь в создании бессерверных приложений Node.js. Она отличается прагматическим подходом и рассказывает о вымышленной пиццерии тетушки Марии, чьи проблемы мы будем пытаться решить с помощью бессерверной архитектуры. Книга начинается с объяснения основ бессерверной архитектуры и потом раз за разом описывает решение каждой проблемы, с которой сталкивается тетушка Мария, с применением отдельных идей бессерверных вычислений, помогая тем самым сформировать ясное представление о приемах создания эффективных бессерверных приложений с Node.js.

Кому адресована книга

Книга «*Бессерверные приложения на JavaScript*» предназначена для разработчиков веб-приложений на JavaScript, стремящихся узнать, как создавать бессерверные приложения, и понять, как правильно их организовывать, проектировать и тестировать. В интернете можно найти массу информации о Node.js и множество пособий по созданию простых бессерверных приложений, и тем не менее в этой книге мы последовательно расскажем, как применить все эти знания для создания больших бессерверных приложений с Node.js.

Краткое содержание

Книга делится на 3 части и 15 глав.

В первой части описываются основы бессерверных вычислений и как построить бессерверное приложение с базой данных, как подключиться к сторонним службам, как отладить его, как добавить поддержку авторизации и аутентификации и как работать с файлами.

- *Глава 1* знакомит с бессерверной платформой Amazon Web Services и описывает бессерверные вычисления с применением простых аналогий. Здесь вы также познакомитесь с тетушкой Марией, ее пиццерией и проблемами. Наконец, вы увидите, как выглядит типичное бессерверное приложение Node.js, и узнаете, что такое Claudia.js и как этот инструмент помогает развертывать приложения Node.js в AWS Lambda.
- *Глава 2* иллюстрирует разработку простого Pizzeria API с использованием AWS Lambda, API Gateway и Claudia API Builder. Здесь вы также научитесь одной командой развертывать свои API с помощью Claudia.
- *Глава 3* рассказывает, как в бессерверной архитектуре работают базы данных, а также о том, как подключить Pizzeria API к DynamoDB – бессерверной базе данных, предлагаемой AWS.

- *Глава 4* рассказывает, как подключить Pizzeria API к сторонним службам, таким как служба доставки, а также знакомит с некоторыми распространенными проблемами, с которыми можно столкнуться при использовании объектов `Promise` с Claudia API Builder.
- *Глава 5* покажет, как искать ошибки в бессерверных приложениях, как их отлаживать и какие инструменты отладки имеются в вашем распоряжении.
- *Глава 6* показывает, как реализовать аутентификацию и авторизацию в бессерверном приложении. Здесь вы узнаете, чем отличается аутентификация от авторизации в бессерверной среде, как реализовать механизм веб-авторизации с помощью AWS Cognito и как идентифицировать своих пользователей с помощью социальных сетей.
- *Глава 7* описывает возможности хранения файлов в бессерверном окружении и показывает, как создать отдельную функцию для обработки файлов, которая использует хранилище и предоставляет запрошенные файлы другим вашим функциям в AWS Lambda, составляющим ваш бессерверный API.

Во второй части рассказывается, как создавать дополнительные бессерверные приложения, работающие с теми же ресурсами, как создавать чат-ботов, голосовых помощников, SMS-чат-ботов, как добавить обработку естественного языка и как следует организовывать все эти бессерверные приложения вместе.

- *Глава 8* показывает, как создать свой первый чат-бот для Facebook Messenger и как Claudia Bot Builder поможет вам сделать это, написав всего несколько строк.
- *Глава 9* показывает, как добавить простую обработку текстов на естественном языке в свой чат-бот, подключить чат-бот к базе данных DynamoDB и организовать асинхронную отправку отложенных ответов.
- *Глава 10* показывает, как использовать голосового помощника Alexa и SMS-чат-бота Twilio и как Claudia Bot Builder позволяет сделать это невероятно быстро.

Третья часть охватывает более сложные темы: тестирование бессерверных приложений и миграция существующих приложений в бессерверное окружение. Здесь также даются рекомендации, описываются типичные шаблоны программирования, решения распространенных проблем и приводятся ответы на часто задаваемые вопросы. Еще тут будет представлен пример двух компаний, перешедших на бессерверные вычисления.

- *Глава 11* рассказывает о тестировании бессерверных приложений: как писать бессерверные функции, чтобы упростить их тестирование, и как

выполнять автоматизированные тесты локально. Здесь также рассказывается о гексагональной архитектуре и о том, как реорганизовать бессерверные приложения, чтобы упростить их тестирование и устранять потенциальные риски.

- *Глава 12* посвящена обработке платежей с помощью бессерверных приложений, реализации приема платежей в бессерверном API и описанию требований к безопасности при обработке платежей.
- *Глава 13* расскажет все, что вы должны знать о запуске приложений Express.js в AWS Lambda и бессерверной экосистеме, обслуживании статического контента из приложения Express.js, подключении к MongoDB из бессерверного приложения Express.js и об ограничениях и рисках приложения Express.js в бессерверной экосистеме.
- *Глава 14* рассказывает, с чего начать миграцию существующего приложения в бессерверное окружение, как привести структуру приложения в соответствие с характеристиками провайдера услуг бессерверных вычислений, как организовать архитектуру приложения, чтобы она была ориентирована на бизнес и могла развиваться, как учесть архитектурные различия между бессерверными и традиционными серверными приложениями.
- *Глава 15* рассказывает, как CodePen использует преимущества бессерверных вычислений для своих препроцессоров, обеспечивая обработку сотен миллионов запросов, и как MindMup способна обслуживать 400 000 активных пользователей с командой из двух человек благодаря бессерверным технологиям.

Об исходном коде

Эта книга содержит много примеров исходного кода и в виде листингов, и в виде фрагментов в обычном тексте. В обоих случаях исходный код оформляется `моноширинным шрифтом`, чтобы его можно было отличить от обычного текста. Иногда, чтобы подчеркнуть отличия от предыдущего шага и выделить вновь добавленные особенности, код будет оформляться **`жирным моноширинным шрифтом`**.

Во многих случаях оригинальный исходный код был переформатирован; мы добавили переносы строк и изменили ширину отступов, чтобы уместить строки кода по ширине книжной страницы. Кроме того, мы убрали комментарии из кода, если он описывается в тексте книги. Многие листинги сопровождаются дополнительными аннотациями, подчеркивающими наиболее важные идеи.

Исходный код примеров в книге доступен для загрузки на сайте издательства: https://manning.com/books/serverless-apps-with-node-and-claudiajs.

Автор в сети

Одновременно с покупкой книги «*Бессерверные приложения на JavaScript*» вы получаете бесплатный доступ к частному веб-форуму, организованному издательством Manning Publications, где можно оставлять комментарии о книге, задавать технические вопросы, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, откройте в веб-браузере страницу https://forums.manning.com/forums/serverless-apps-with-node-and-claudiajs. Кроме того, узнать больше о правилах поведения на форуме можно по адресу: https://forums.manning.com/forums/about.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Но со стороны автора отсутствуют какие-либо обязательства уделять форуму какое-то определенное внимание – его присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать автору стимулирующие вопросы, чтобы его интерес не угасал!

Другие онлайн-ресурсы

Те, кому понадобится дополнительная помощь, могут:

- перейти на страницу проекта Claudia.js в Gitter: https://gitter.im/claudiajs/claudia, где обычно авторы отвечают на вопросы о Claudia.js, Claudia API Builder и Claudia Bot Builder;
- выполнить поиск по тегу *claudiajs* на сайте Stack Overflow (http://stackoverflow.com/questions/tagged/claudiajs) и найти сообщения с вопросами и ответами, касающимися разработки бессерверных приложений с Node.js и Claudia.js. Здесь же вы сможете помочь другим, столкнувшимся с проблемами, решение которых вам уж известно.

Об авторах

Слободан Стоянович (Slobodan Stojanovic) и **Александар Симович** (Aleksandar Simovic) являются обладателями титула AWS Serverless Heroes и основными участниками проекта Claudia.js. Они занимаются разработкой и сопровождением Claudia Bot Builder и являются соавторами книги «Бессерверные приложения на JavaScript».

Александар больше семи лет работал старшим консультантом и инженером по программному обеспечению, но не только на JavaScript. Он также увлекается языками Swift, Python и Rust. Живет в Белграде и является соорганизатором встреч JS Belgrade.

Слободан – технический директор Cloud Horizon, студии разработки программного обеспечения, базирующейся в Монреале. Живет в Белграде и является соорганизатором встреч JS Belgrade.

Об иллюстрации на обложке

На обложке книги «*Бессерверные приложения на JavaScript*» изображен рисунок с названием «Сербка из Шумадии». Иллюстрация взята из книги «Сербские национальные костюмы» Владимира Кирина (Vladimir Kirin). Кирин (1894–1963) изучал графический дизайн в Лондоне, посещал Академию художеств в Вене, работал художником и иллюстратором. Считается, что его работа способствовала совершенствованию оформления книг в Хорватии.

На протяжении всей своей богатой истории центральный регион Сербии, известный как Шумадия, был культурным центром, и традиционная одежда этого района является стандартом национального костюма Сербии. Как показано на этом изображении, традиционное сербское женское платье состояло из *опанчи*, вышитых шерстяных носков, доходивших до колен. Юбки были разнообразные, плиссированные или собранные в складки и вышитые, с *тканицей*, служившей поясом. Важной частью костюма был *передник*, украшенный цветочными мотивами. Рубашки были в форме туники, богато украшенные серебряной нитью, а поверх рубашки надевались шнуры. Девушки также носили нашейные украшения или монисто из золотых монет вокруг горла, сережки, браслеты, а их волосы были украшены металлическими монетами или цветами.

Мы в издательстве Manning славим изобретательность, предприимчивость и радость компьютерного бизнеса обложками книг, изображающими богатство региональных различий двухвековой давности, оживших благодаря Кирин.

Часть I

Бессерверная пиццерия

Тетушка Мария – волевой человек. Вот уже тридцать лет она управляет своей пиццерией, местом сбора людей из разных поколений, живущих неподалеку: многие проводили там время со своими семьями, смеялись и даже ходили на романтические свидания. Но в последнее время в ее пиццерии наступили тяжелые времена. Число посетителей постепенно уменьшается. Развитие технологий привело к тому, что ее клиенты предпочитают делать заказы онлайн через веб-сайты или телефоны в пиццериях конкурентов.

У ее пиццерии уже есть веб-сайт, но для него нужно написать приложение, обрабатывающее и хранящее информацию о пиццах и заказах.

Наша задача в первой части этой книги – помочь тетушке Марии наверстать упущенное, создав для нее бессерверный API. Но так как вы все еще новичок в разработке бессерверных приложений, сначала мы расскажем вам, что такое бессерверные вычисления и как они могут помочь в создании Pizzeria API (глава 1). Затем вы добавите маршруты в свой API и развернете его в AWS Lambda с помощью Claudia (глава 2). Чтобы сохранить и выполнить все заказы, необходимо связать новый API с таблицей DynamoDB (глава 3) и установить связь со сторонней службой доставки (глава 4).

Во время разработки вы столкнетесь с некоторыми проблемами и узнаете, как отлаживать бессерверные приложения (глава 5).

Чтобы сделать API полностью функциональным, нужно научиться аутентифицировать и авторизовать пользователей (глава 6), а также сохранять и манипулировать изображениями пиццы (глава 7).

Глава 1

Введение в бессерверные вычисления с Claudia

Эта глава охватывает следующие темы:

- что такое бессерверные вычисления;
- основные понятия бессерверных вычислений;
- различия между бессерверными и серверными веб-приложениями;
- назначение Claudia;
- преимущества бессерверных вычислений.

Бессерверные вычисления – это способ развертывания и выполнения приложений в облачной инфраструктуре с оплатой за фактическое использование, без аренды или покупки серверов. За планирование, масштабирование, балансировку и мониторинг вычислительных мощностей отвечает поставщик услуг бессерверных вычислений. Кроме того, поставщик может рассматривать ваши приложения как функции.

Что значит *бессерверные*? Создается впечатление, что это еще одно новомодное словцо, обещающее улучшить вашу жизнь.

В этой книге объясняется, что такое бессерверные вычисления, какие проблемы они решают и где могут или не могут использоваться для ваших приложений, без рекламной шелухи и попытки втюхать вам бессерверные подходы как какой-то модный облачный культ, которому должен следовать каждый. Мы предпримем более прагматический подход и объясним понятия, попутно демонстрируя приемы создания надежных и масштабируемых бессерверных приложений с Node.js и Claudia.js.

В этой главе основное внимание уделяется понятию «бессерверные вычисления»: что это такое, почему важно иметь представление о них и в чем их отличие и сходство с серверными вычислениями. Ваша главная цель в этой

главе – получить хорошее представление об основных понятиях бессерверных вычислений и заложить прочный фундамент для будущих глав.

1.1. Серверы и стиральные машины

Чтобы понять идею бессерверных вычислений, рассмотрим пример со стиральными машинами. Устройство стирки одежды может показаться странной аналогией, но владение сервером в настоящее время похоже на владение стиральной машиной. Всем нужна чистая одежда, и покупка стиральной машины кажется вполне логичным решением. Но большую часть времени стиральная машина простаивает без дела. В лучшем случае она используется от 5 до 15 часов в неделю. То же самое касается серверов. В большинстве случаев средний сервер приложений просто ждет получения запроса, ничего не делая.

Самое интересное, что у серверов и стиральных машин много общих проблем. И те, и другие имеют предельный вес или объем, который они могут обработать. Владение небольшим сервером аналогично владению небольшой стиральной машиной; если накапливается большая куча белья, машина не сможет обработать все сразу. Вы можете купить большую машину, способную выстирать сразу до 8 кг одежды, но тогда возникнет другая проблема – запуск большой машины для стирки единственной рубашки окажется слишком расточительным. Кроме того, настроить единственный сервер для безопасного и надежного выполнения всех имеющихся приложений сложно, а иногда невозможно. Правильная настройка для одного приложения может совершенно не подходить для другого. Точно так же перед стиркой требуется рассортировать одежду по цвету, а затем выбрать правильную комбинацию программы, моющего средства и смягчителя. Если выбрать неправильные настройки или моющие компоненты, машина может испортить вашу одежду.

Эти проблемы, а также проблема, заключающаяся в том, что не каждый может купить стиральную машину, привели к росту количества прачечных самообслуживания или прачечных-автоматов, предлагающих платные услуги стиральных машин для стирки одежды. Аналогичная потребность в отношении серверов привела к появлению компаний, предоставляющих услуги аренды серверов, как локальных, так и в облаке. Вы можете арендовать сервер, а поставщик услуги позаботится об их сохранности, электропитании и основных настройках. Но как прачечные, так и прокатные серверы решают лишь часть проблем.

Арендуя стиральные машины или серверы, вы все равно должны знать, как сочетать одежду или приложения и выбирать программу стирки, подходящие моющие средства или настраивать среду выполнения на серверах. Вы также должны учесть количество машин и их ограничения по размеру, заранее планируя, сколько машин арендовать.

В мире бытовых услуг во второй половине XX века появилась новая услуга – услуга по вызову. Суть ее заключается в следующем: вы собираете белье в кучу и звоните в прачечную, а служащие прачечной заберут его, постирают, высушат, сложат в аккуратную стопку и, если вы пожелаете, доставят постиранное

белье вам на дом. Сдавать белье в стирку часто можно поштучно, то есть вам не нужно ждать, пока накопится определенный объем грязного белья, и не нужно беспокоиться о стиральных машинах, моющих средствах и программах стирки.

В отличие от индустрии бытовых услуг, индустрия программного обеспечения все еще находится на стадии прачечных самообслуживания, так как многие из нас все еще арендуют серверы или пользуются услугами PaaS (платформа как услуга). Мы по-прежнему должны оценить количество потенциальных запросов (количество одежды), которые собираемся обработать, и резервируем достаточное число серверов, чтобы (мы надеемся) справиться с нагрузкой, часто тратя наши деньги на серверы, которые либо не работают на полную мощность, либо перегружены и не могут обработать все запросы наших клиентов.

1.2. Основные понятия

Итак, что меняет внедрение бессерверных вычислений? Судя по названию технологии, она подразумевает отсутствие серверов, что выглядит более чем нелогично. Вернемся к определению, которое было дано в начале главы:

Что такое бессерверные вычисления?

Бессерверные вычисления – это способ развертывания и выполнения приложений в облачной инфраструктуре с оплатой за фактическое использование, без аренды или покупки серверов.

Вопреки своему названию, технология бессерверных вычислений не исключает существования серверов; программному обеспечению нужна аппаратура, на которой оно будет выполняться. Под словом «бессерверный» просто подразумевается, что компаниям, организациям или разработчикам не требуется приобретать либо арендовать физический сервер.

Возможно, вам интересно узнать, почему было выбрано такое название. Потому что эта технология основывается на абстрагировании от понятия «сервер». Вместо приобретения или аренды сервера для своего приложения, настройки и развертывания среды выполнения вы просто выгружаете приложение в облако поставщика услуг бессерверных вычислений, который сам позаботится о том, чтобы выделить серверы, хранилища, настроить среду выполнения для приложения и запустить его.

ПРИМЕЧАНИЕ. Кому-то из вас может быть интересно, избавляет ли технология бессерверных вычислений от необходимости иметь в компаниях свои подразделения сопровождения и эксплуатации программного обеспечения. В большинстве случаев ответ на этот вопрос: да, избавляет.

Точнее, поставщик услуг сохранит ваше приложение в некотором контейнере. Контейнер представляет изолированное окружение, содержащее все, необходимое вашему приложению для работы. Контейнер можно представить как горшок для комнатного растения. Он содержит грунт со всеми питательными веществами, необходимыми вашему растению.

Как и горшок с растением, контейнер позволяет поставщику услуги бессерверных вычислений безопасно перемещать и хранить ваше приложение, а также выполнять его и копировать в зависимости от ваших потребностей. Но главное преимущество бессерверной технологии – отсутствие необходимости выполнять какие-либо настройки, балансировать, масштабировать серверы, то есть решать любые задачи управления сервером. Провайдер сам управляет всем этим за вас, а также гарантирует, что с увеличением нагрузки на ваше приложение он создаст достаточное количество копий контейнера с приложением для обработки всех вызовов, и каждый контейнер будет точной копией исходного. Если потребуется, провайдер создаст тысячи копий. Решение о запуске еще одной копии контейнера принимается провайдером, только когда количество запросов к вашему приложению становится настолько большим, что текущее число действующих контейнеров не успевает обработать их все.

Если к вашему приложению вообще не поступает запросов, провайдер остановит все экземпляры, соответственно, оно не будет расходовать память и процессорное время сервера. Провайдер услуги бессерверных вычислений отвечает за все детали, касающиеся работы: он знает, где хранится ваше приложение, как и куда его копировать, когда запускать новые контейнеры и когда уменьшать количество контейнеров при снижении нагрузки.

Продолжая аналогию со стиральными машинами, процесс предоставления услуги бессерверных вычислений напоминает услуги прачечной по вызову; служащий прачечной появляется у вашей двери, чтобы забрать грязное белье, затем оно стирается в прачечной и потом возвращается к вам. Независимо от того, сколько у вас одежды и каких видов (шерсть, хлопок, кожа и т. д.), прачечная берет на себя всю ответственность за сортировку белья, выбор моющих средств и программ.

Бессерверные вычисления и FaaS

Первоначально термин «бессерверные вычисления» интерпретировался иначе, чем сейчас. Подразумеваемая под ним технология называлась *сервер как услуга* (Backend as a Service, BaaS) и предназначалась для приложений, которые частично или полностью зависят от сторонних услуг по предоставлению серверной логики. Позднее эта технология стала называться *функция как услуга* (Function as a Service, FaaS), поскольку провайдеры услуг бессерверных вычислений интерпретируют приложения как функции, вызывая их только по запросу.

1.3. Как работают бессерверные вычисления?

Как отмечалось выше, провайдеры услуг бессерверных вычислений предлагают изолированные контейнеры для приложений. Контейнер управляется событиями, поэтому активируется только при появлении определенного события.

События – это конкретные внешние воздействия, подобные физическим выключателям. Возьмем в качестве примера освещение в доме: события, включающие его, могут отличаться. Свет может быть включен человеком, щелкнувшим обычным выключателем; датчиком движения; датчиком освещенности, включающим свет, когда садится солнце. Но контейнеры не ограничиваются приемом определенных событий и вызовом содержащихся в них функций; они также позволяют вашим функциям самим создавать события или, точнее, их генерировать. В техническом смысле в бессерверных вычислениях контейнеры функций являются и *приемниками*, и *источниками событий*.

Наконец, провайдеры предлагают различные события, способные запускать ваши функции. Список событий зависит от провайдера и реализации, но часто их роль играют HTTP-запросы, выгрузка файлов в хранилище, обновление базы данных, события интернета вещей (Internet of Things, IoT) и множество других.

> **ПРИМЕЧАНИЕ.** Бессерверные функции запускаются только по событиям, и вы платите лишь за время выполнения. После выполнения провайдер отключает функцию, сохраняя возможность повторного ее запуска по следующему событию.

1.4. Бессерверные вычисления на практике

Ландшафт бессерверных вычислений содержит множество движущихся частей, поэтому далее мы познакомимся с ними поближе. С этой целью мы создадим пример приложения, разрабатывая его поэтапно, чтобы вы могли видеть, как оно конструируется. По мере знакомства с новыми понятиями мы будем расширять пример приложения.

В этой книге мы напишем пример совершенно нового приложения (оно будет создано «с нуля»), решающего проблемы небольшой компании, а точнее – пиццерии. Пиццерия управляется вашей вымышленной тетушкой Марией. В течение книги тетушка Мария столкнется со множеством реальных проблем, и наша цель – помочь ей в этом, попутно усваивая понятия бессерверных вычислений. Бессерверные вычисления, как и любая новая технология, вводят множество новых понятий, с которыми сложно справиться, если рассматривать их все сразу.

> **ПРИМЕЧАНИЕ.** В случае проблем при переносе существующего приложения на бессерверную платформу не стесняйтесь обращаться к последней части книги. Если вы пока незнакомы с бессерверными вычислениями, прочитайте хотя бы несколько первых глав, прежде чем переходить к последней части книги.

1.4.1. Бессерверная пиццерия тетушки Марии

Тетушка Мария – волевой человек. Вот уже тридцать лет она управляет своей пиццерией, местом сбора людей из разных поколений, живущих неподалеку: многие проводили там время со своими семьями, смеялись и даже ходили на романтические свидания. Но в последнее время в ее пиццерии наступили тяжелые времена. Число посетителей постепенно уменьшается. Многие из ее клиентов теперь предпочитают делать заказы онлайн, через веб-сайты или телефоны в пиццериях конкурентов. Некоторые новые компании начали переманивать ее клиентов. Например, в новой пиццерии Chess запустили мобильное приложение с предварительным просмотром пиццы и возможностью сделать заказ онлайн, а также чат-бота для заказов через различные приложения мгновенного обмена сообщениями. Клиенты нашей тетушки любят ее пиццерию, но многие предпочитают заказывать пиццу с доставкой на дом, поэтому ее тридцатилетний бизнес начал угасать. В пиццерии уже есть веб-сайт, но для обработки и хранения информации о пиццах и заказах требуется внутреннее (серверное) приложение.

1.4.2. Распространенный подход

Учитывая ограниченность ресурсов тетушки Марии, самым простым решением является создание небольшого API с использованием популярного фреймворка Node.js, такого как Express.js или Hapi, и настройка базы данных (скорее всего, MongoDB, MySQL или PostgreSQL).

Код типичного API делится на несколько уровней и напоминает трехуровневую архитектуру. То есть код должен делиться на такие уровни, как представление, бизнес-логика и данные.

> **Трехуровневая архитектура**
>
> *Трехуровневая архитектура* – это шаблон архитектуры клиент-серверного программного обеспечения, в котором пользовательский интерфейс (представление), логика работы («бизнес-правила»), хранение данных и доступ к ним разрабатываются и поддерживаются как независимые модули, чаще всего на отдельных платформах.
>
> Узнать больше о трехуровневой архитектуре можно на странице Википедии https://ru.wikipedia.org/wiki/Трехуровневая_архитектура.

На рис. 1.1 изображен дизайн типичного трехуровневого приложения с отдельными маршрутами для пиццы, заказов и пользователей. В нем также должны иметься точки входа для чат-ботов и обработчика платежей. Все маршруты должны запускать некоторые функции-обработчики на уровне бизнес-логики, а результаты обработки – отправляться на уровень данных, в базу данных и хранилище файлов и изображений.

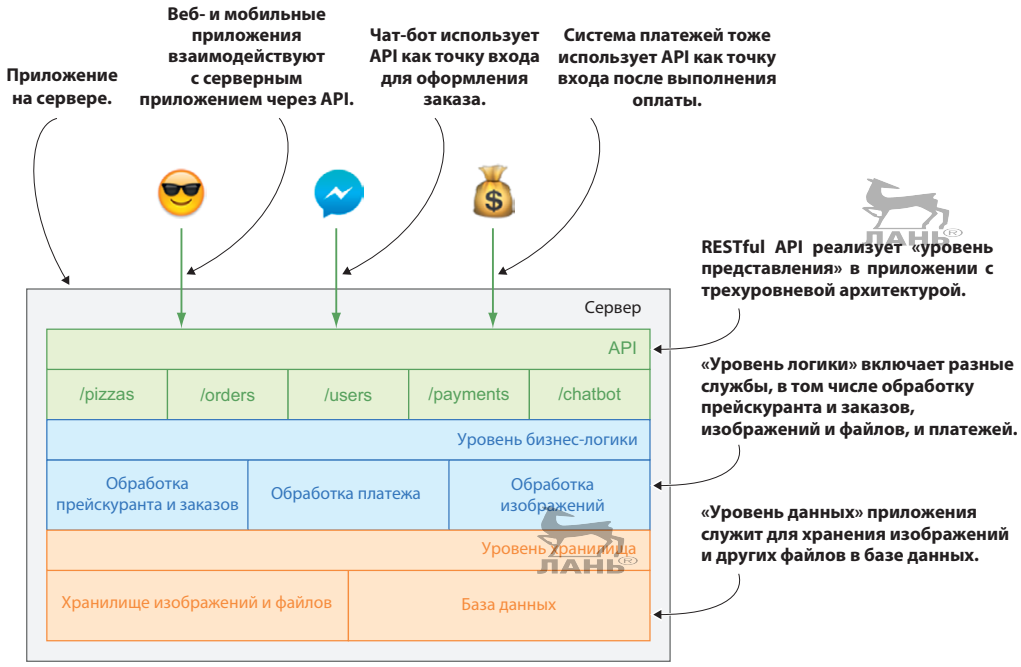


Рис. 1.1. Pizza API с типичным трехуровневым дизайном

Этот подход идеален для небольших приложений, в том числе и для нашего Pizza API, по крайней мере до тех пор, пока количество заказов на пиццу не вырастет до определенного уровня. После этого вам потребуется масштабировать инфраструктуру.

Но, чтобы масштабировать монолитное приложение, необходимо отделить слой данных (чтобы не копировать базу данных ради согласования данных). После этого приложение будет выглядеть так, как показано на рис. 1.2. И у нас все равно остается единый конгломерат со всеми маршрутами и бизнес-логикой. Такое приложение можно реплицировать (запускать дополнительные копии, чтобы увеличить пропускную способность), если у вас слишком много пользователей, но в каждом экземпляре будут присутствовать все службы приложения, независимо от интенсивности их использования.

Монолитное приложение

Монолитным называют приложение, в котором интерфейс пользователя и код доступа к данным объединены в одну программу на одной платформе. Монолитное приложение является автономным и независимым от других приложений.

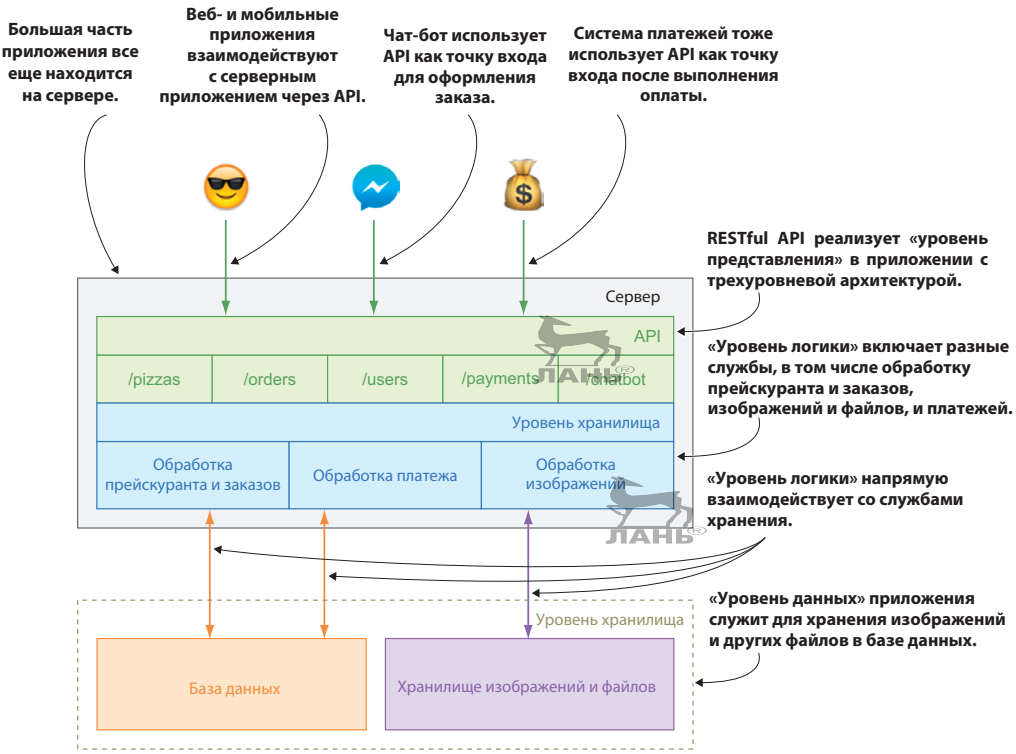


Рис. 1.2. Типичная архитектура с внешней базой данных и хранилищем файлов

1.4.3. Бессерверное решение

Для использования бессерверных вычислений требуется иной подход, так как приложения в этом случае управляются событиями и являются распределенными.

Все части бессерверного приложения с конечными точками API и бизнес-логикой изолируются в независимых и автоматически масштабируемых контейнерах.

В бессерверном приложении запросы обрабатываются на уровне маршрутизатора API, который решает единственную задачу: принимает HTTP-запросы и направляет их в службы уровня бизнес-логики. Маршрутизатор API в бессерверной архитектуре всегда действует независимо. Это означает, что от разработчиков приложений не требуется поддерживать маршрутизацию API, и поставщик услуг бессерверных вычислений автоматически масштабирует приложение, чтобы обеспечить своевременную обработку всех HTTP-запросов, поступающих в адрес вашего API. Также вы платите только за запросы, которые обрабатываются.

В примере нашего Pizza API маршрутизатор будет принимать все запросы от мобильных и веб-приложений и, если необходимо, обслуживать точки входа для чат-ботов и системы обработки платежей.

После получения запроса маршрутизатор передает его для обработки в другой контейнер со службой уровня бизнес-логики.

В бессерверных приложениях бизнес-логика часто разбивается на более мелкие единицы. Размер каждой единицы зависит от предпочтений разработчика. Единицей может быть единственная функция или целое монолитное приложение. В большинстве случаев размер единицы не влияет напрямую на сумму оплаты услуг инфраструктуры, поскольку вы платите за выполнение функции. Кроме того, единицы масштабируются автоматически, и вам не придется платить за единицы, которые ничего не обрабатывают, поэтому владение одной или дюжиной единиц обходится одинаково.

Однако в ситуациях с небольшими приложениями или когда обрабатывается не очень большой объем информации, можно сэкономить на хостинге и обслуживании, объединив функции, связанные с одной службой, в одну бизнес-единицу. Для Pizza API вполне разумно будет создать одну единицу для обработки прейскуранта и заказов, одну для обработки платежей, одну для обработки сообщений от чат-бота и одну для обработки изображений и файлов.

Последняя часть нашего бессерверного API – уровень данных, который мало чем отличается от уровня данных в масштабируемом монолитном приложении с отдельно масштабируемой базой данных и службой хранения файлов. Было бы лучше, если бы база данных и хранилище файлов были также независимыми и автоматически масштабируемыми.

Еще одно преимущество бессерверных приложений – уровень данных может вызывать бессерверную функцию «из коробки». Например, когда в хранилище выгружается изображение пиццы, есть возможность запустить службу обработки изображений, которая изменит размер фотографии и свяжет ее с конкретной пиццей в прейскуранте.

Потоки обработки данных в бессерверном Pizza API можно видеть на рис. 1.3.

1.5. Бессерверная инфраструктура – AWS

Нашему бессерверному Pizza API нужна инфраструктура для работы. Технология бессерверных вычислений еще очень молода, и на данный момент имеется лишь несколько вариантов инфраструктуры. Большинство вариантов принадлежит крупным поставщикам, поскольку для бессерверных вычислений требуется большая и развитая инфраструктура с поддержкой масштабирования. Самыми известными и наиболее продвинутыми инфраструктурами являются бессерверный контейнер Amazon AWS Lambda, Microsoft Azure Functions и Google Cloud Functions.

В этой книге мы будем использовать AWS Lambda, потому что это самая зрелая из доступных на рынке бессерверных инфраструктур, она имеет стабильный API и множество успешных историй использования.

AWS Lambda – это бессерверная вычислительная платформа, управляемая событиями, которая предлагается компанией Amazon как часть Amazon Web

Services. Это вычислительная служба, которая запускает код в ответ на события и автоматически управляет вычислительными ресурсами, необходимыми этому коду.

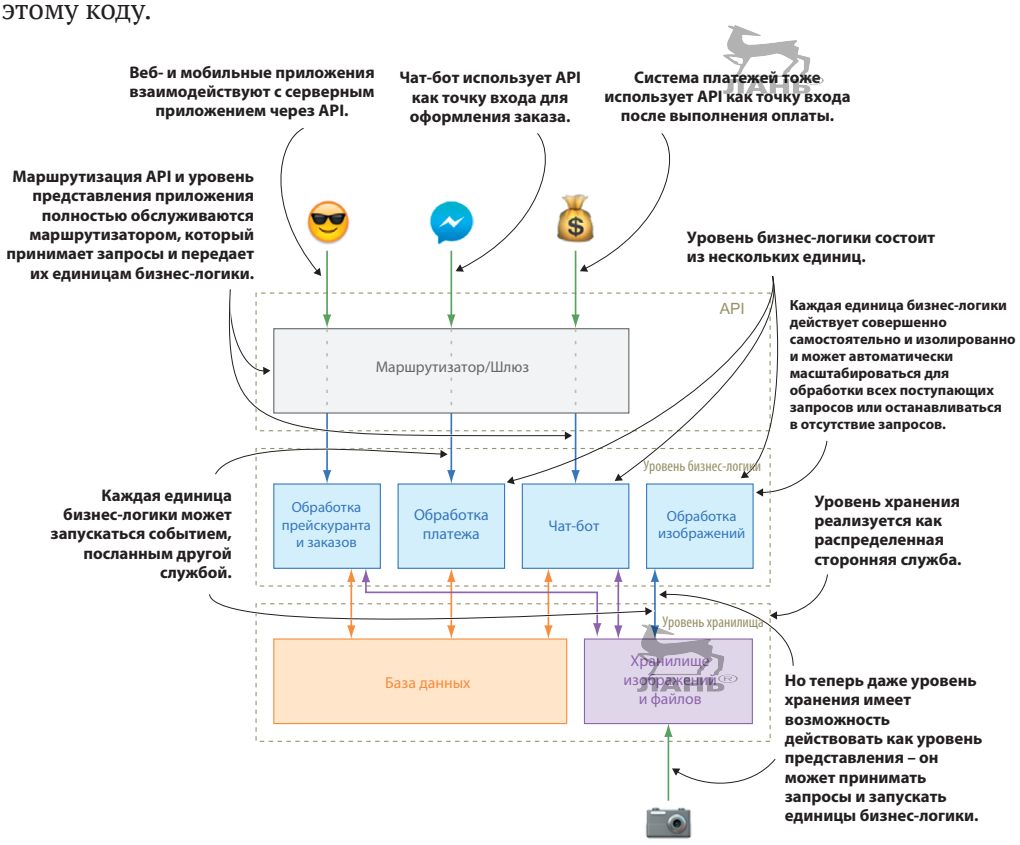


Рис. 1.3. Бессерверная реализация Pizza API

Google Cloud Functions и Microsoft Azure Functions

Компания Google запустила услугу Google Cloud Functions в середине 2016 года, в ответ на появление Amazon AWS Lambda. Инфраструктура Google Cloud Functions позиционируется как набор легковесных микросервисов, управляемых событиями, которые позволяют запускать функции на JavaScript в среде выполнения Node.js. Ваша функция может быть вызвана в ответ на HTTP-запрос, событие из Google Cloud Storage и других служб Google Cloud Pub/Sub. На момент написания этих строк инфраструктура Google Cloud Functions еще находилась в стадии разработки, поэтому цены не были известны. Дополнительные подробности вы можете узнать на официальном сайте: https://cloud.google.com/functions/.

Реализация бессерверных вычислений от корпорации Microsoft – Azure Functions – является частью ее платформы облачных вычислений Azure.

Microsoft описывает ее как инфраструктуру бессерверных вычислений, управляемых событиями, которая ускоряет разработку, осуществляет масштабирование в зависимости от спроса и взимает плату только за фактически потребленные ресурсы. Инфраструктура Azure Functions позволяет писать функции на JavaScript, C#, F#, Python и других языках сценариев. Цены на услуги Azure аналогичны ценам на AWS Lambda: вы платите 20 центов за 1 миллион выполнений и 0,000016 доллара США за гигабайт потребленных ресурсов в месяц, при этом обработка первого миллиона запросов и 400 000 Гбайт каждый месяц предоставляются бесплатно. За дополнительной информацией обращайтесь на официальный веб-сайт: https://azure.microsoft.com/ru-ru/services/functions/.

ПРИМЕЧАНИЕ. Большинство из того, с чем вы познакомитесь в этой книге, также можно реализовать с помощью других поставщиков услуг бессерверных вычислений, но некоторые услуги могут отличаться, поэтому для отдельных решений может потребоваться использовать немного иной подход.

На платформе Amazon слово *бессерверные* обычно напрямую связано с AWS Lambda. Но для бессерверных приложений, таких как Pizza API, AWS Lambda является лишь одним из строительных блоков. Чтобы создать полноценное приложение, часто нужны другие службы, такие как службы хранения, маршрутизации и базы данных. В табл. 1.1 перечислены все необходимые службы, предлагаемые платформой AWS:

- Lambda – используется для вычислений;
- API Gateway (шлюз API) – выполняет маршрутизацию, принимая HTTP-запросы и вызывая другие службы в зависимости от маршрута;
- DynamoDB – автоматически масштабируемая база данных;
- Simple Storage Service (S3) – служба хранилища, реализующая абстракцию обычного жесткого диска и предлагающая неограниченное пространство для хранения.

Таблица 1.1. Строительные блоки бессерверных приложений в AWS

Назначение	Служба AWS	Краткое описание
Вычисления	Lambda	Вычислительный компонент для бизнес-логики
Маршрутизация	API Gateway	Компонент маршрутизации, используется для маршрутизации HTTP-запросов к функциям Lambda
База данных	DynamoDB	Автоматически масштабируемая документ-ориентированная база данных
Хранилище	S3	Служба автоматически масштабируемого хранилища файлов

Lambda – это самая важная часть инфраструктуры бессерверных вычислений, с которой вы обязательно должны разобраться, потому что она содержит вашу бизнес-логику. Lambda – это бессерверный вычислительный контейнер AWS, который запускает вашу функцию по событию. Он автоматически масштабируется, если в функцию поступает сразу большое число событий. Чтобы создать Pizza API в виде бессерверного приложения, необходимо использовать вычислительный бессерверный контейнер AWS Lambda.

Когда возникает определенное событие, такое как HTTP-запрос, служба Lambda запускает функцию и передает ей аргументы с данными из события, контекстом и функцию обратного вызова для отправки ответа. Функция в терминологии Lambda – это самая обычная функция-обработчик, написанная на одном из поддерживаемых языков. На момент написания этих строк AWS Lambda поддерживала следующие языки:

- Node.js;
- Python;
- Java (Java 8) и другие языки JVM;
- C# (.NET Core).

В Node.js данные события, контекст и функция обратного вызова передаются как объекты JSON. Объект контекста `context` содержит подробную информацию о вашей функции и ее текущем выполнении, такую как время выполнения, причина вызова функции и т. д. Третий аргумент, который получает ваша функция, – это функция обратного вызова, которая позволит вам вернуть ответ с некоторой полезной информацией или ошибкой, которая будет отправлена обратно, службе, сгенерировавшей событие. В листинге 1.1 показан пример небольшой функции AWS Lambda, которая возвращает текст *Hello from AWS Lambda*.

Листинг 1.1. Пример минимальной действующей функции Lambda на Node.js

```
function lambdaFunction(event, context, callback) {
    callback(null, 'Hello from AWS Lambda')
}

exports.handler = lambdaFunction
```

Функция принимает событие, контекст и функцию обратного вызова.

Функция обратного вызова вызывается для возврата сообщения об успехе.

Функция экспортируется как обработчик.

ПРИМЕЧАНИЕ. Как показано в листинге 1.1, функция должна экспортироваться присваиванием ссылки на нее свойству `exports.handler`, а не `module.exports`, стандартному свойству для Node.js. Это объясняется тем, что в AWS Lambda экспортируемый элемент должен быть объектом с методом `handler`, а не функцией непосредственно.

Как упоминалось выше, событие `event` в вашей функции – это данные, переданные службой, которая запустила вашу функцию. В AWS функции могут вызываться разными службами, такими как маршрутизатор HTTP-запросов, S3, выполняющей операции с файлами, или более экзотическими, такими как службы развертывания кода, изменения инфраструктуры, и даже консольными командами из AWS SDK.

Вот список наиболее важных событий и служб, которые могут вызвать функцию AWS Lambda, и их аналоги в API Pizza:

- *HTTP-запросы, направляемые службой API Gateway*, – на веб-сайт пиццерии поступил новый запрос;
- *выгрузка изображений, удаление или изменение файлов в S3* – выгружено новое изображение пиццы;
- *изменения в базе данных DynamoDB* – получен новый заказ на доставку пиццы;
- *разные уведомления от службы Simple Notification Service (AWS SNS)* – пицца доставлена;
- *результат обработки голосовой команды помощником Amazon Alexa* – клиент заказал пиццу, используя голосовой интерфейс.

Полный список событий, запускающих функции, можно найти по адресу: http://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-function.html.

Функции Lambda имеют некоторые ограничения, например ограниченное время выполнения и объем доступной памяти. По умолчанию функции дается на выполнение до трех секунд, это означает, что работа функции будет прервана по тайм-ауту, если она попытается потратить больше времени на обработку данных. Функция получает 128 Мбайт ОЗУ, то есть она не может выполнять слишком много сложных вычислений.

> **ПРИМЕЧАНИЕ.** Оба эти ограничения можно изменить в настройках функции. Время выполнения можно увеличить до 15 минут, а объем памяти – до 3 Гбайт. Увеличение обоих ограничений может повлиять на стоимость выполнения вашей функции.

Другой важной характеристикой функций Lambda является отсутствие состояния, то есть состояние вычислений не сохраняется между вызовами.

Как показано на рис. 1.4, типичный порядок выполнения функции Lambda выглядит следующим образом:

- происходит определенное событие, и служба, обрабатывающая его, вызывает функцию Lambda;
- функция, как та, что была показана в листинге 1.1, запускается на выполнение;
- функция завершается и возвращает сообщение об успехе или ошибке, или ее выполнение прерывается по истечении установленного времени.

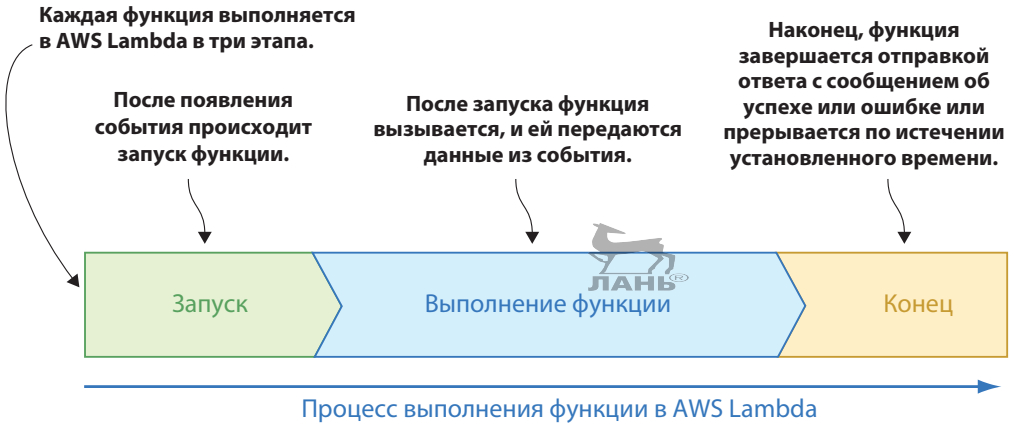


Рис. 1.4. Порядок выполнения функции в AWS Lambda

Стоимость бессерверных вычислений

Одним из уникальных преимуществ бессерверных вычислений является невысокая стоимость. Amazon принимает почасовую оплату за аренду своих стандартных виртуальных серверов Elastic Compute Cloud (Amazon EC2). Стоимость AWS Lambda выше, чем EC2, если сравнивать стоимость часа, но зато вам не придется платить за периоды времени, когда ваша функция не работала. Вы платите 20 центов за 1 миллион выполнений и 0,000016 доллара США за гигабайт потребленных ресурсов в месяц, при этом обработка первого миллиона запросов и 400 000 Гбайт каждый месяц предоставляются бесплатно.

Тетушка Мария нисколько не должна будет платить за работу Pizza API, пока не достигнет 1 миллиона выполнений в месяц. Если это число будет достигнуто, значит, вам удалось помочь ей.

За более подробной информацией о ценах обращайтесь на официальный вебсайт https://aws.amazon .com/lambda/pricing/.

Еще один важный аспект, который может повлиять на наш бессерверный Pizza API, – это задержка в работе функции. Поскольку контейнеры с функциями Lambda управляются провайдером, а не владельцем приложения, невозможно узнать, будет ли событие обслужено существующим контейнером или платформа создаст новый. Если до выполнения функции потребуется создать и инициализировать новый контейнер, на это может уйти чуть больше времени – эта ситуация называется *холодным запуском*, как показано на рис. 1.5. Время, необходимое для запуска нового контейнера, зависит от размера приложения и платформы, используемой для его запуска. Работая над этой книгой, мы установили опытным путем, что задержки с Node.js и Python заметно ниже, чем с Java.

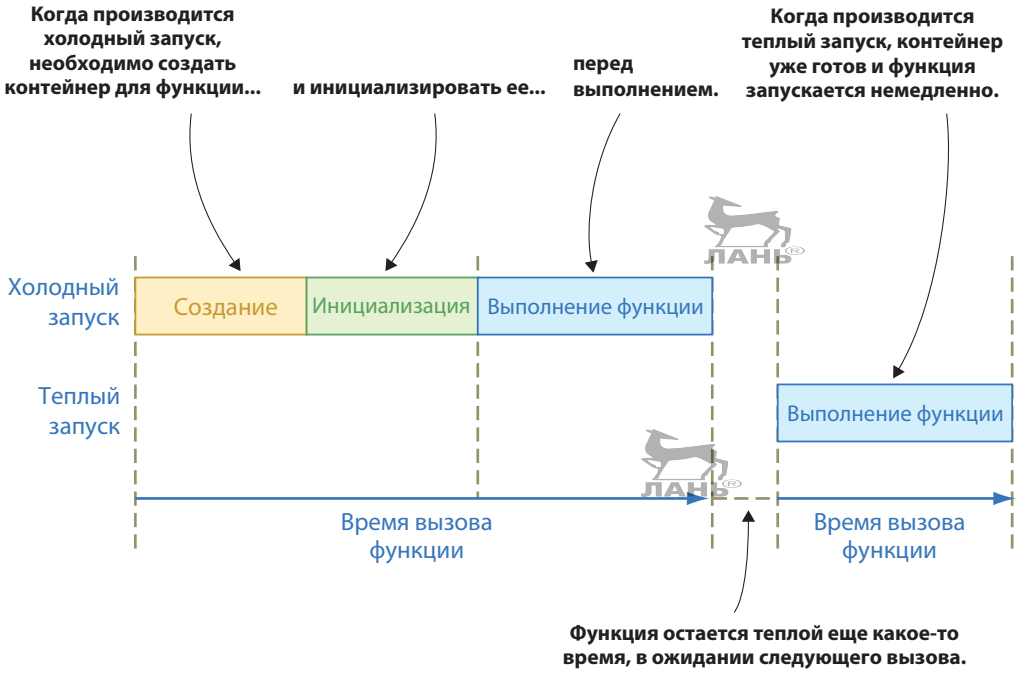


Рис. 1.5. Холодный и теплый запуски функции в AWS Lambda

ПЛАТА ЗА PIZZA API. Разработка Pizza API, описываемая в следующих главах, обойдется вам дешевле чашки кофе. Сама услуга AWS Lambda получится бесплатной, но некоторые службы, используемые в разработке Pizza API, такие как DynamoDB и Simple Storage Service, взимают небольшую плату за хранение ваших данных. Обе эти услуги и их стоимость описаны в последующих главах. Окончательная цена будет зависеть от объема данных и их использования, но если вы будете следовать примерам в книге, она не превысит 1 доллар в месяц.

Функции Lambda просты в освоении и использовании. Самое сложное – это этап развертывания.

Развернуть бессерверное приложение в AWS Lambda можно несколькими способами: с помощью визуального интерфейса консоли AWS Lambda; с помощью интерфейса командной строки из терминала AWS, с использованием AWS API; или напрямую, с помощью AWS SDK для одного из поддерживаемых языков. Развертывание бессерверного приложения проще, чем развертывание традиционного приложения, но его можно упростить еще больше.

1.6. Что такое и для чего используется Claudia?

Claudia – это библиотека Node.js, упрощающая развертывание проектов на Node.js в AWS Lambda и API Gateway. Она автоматизирует все трудоемкие опе-

рации, связанные с развертыванием и настройкой, и подготавливает приложение к работе.

Claudia реализована на основе AWS SDK с целью упростить разработку. Это не замена AWS SDK или AWS CLI, а расширение, которое упрощает решение некоторых типичных задач, таких как развертывание и настройка.

Вот некоторые основные достоинства Claudia:

- создание и изменение функции одной командой (избавляет от необходимости вручную архивировать приложение и затем выгружать архив через пользовательский интерфейс AWS Dashboard);
- избавляет от шаблонных операций, позволяя сосредоточиться на более интересной работе, и сохраняет настройки проекта;
- упрощает управление версиями;
- проста в освоении – чтобы освоить ее, достаточно нескольких минут.

Claudia действует как инструмент командной строки и позволяет создавать и обновлять функции из терминала. Однако в экосистеме Claudia имеется еще две полезные библиотеки Node.js: Claudia API Builder (позволяет создавать API в API Gateway) и Claudia Bot Builder (дает возможность создавать чат-боты для разных платформ обмена мгновенными сообщениями).

В отличие от Claudia, которая используется на стороне клиента и никогда не внедряется в AWS, API Builder и Bot Builder всегда развертываются в AWS Lambda (см. рис. 1.6).

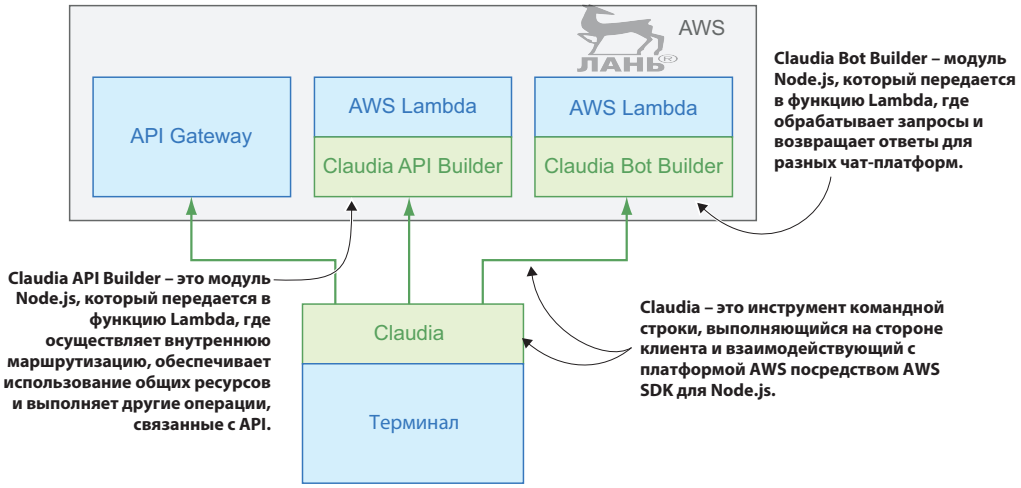


Рис. 1.6. Визуальное представление связей Claudia, API Builder и Bot Builder с платформой AWS

Вы можете работать с AWS Lambda и API Gateway без Claudia, используя экосистему AWS непосредственно или некоторые альтернативы.

К наиболее известным альтернативам относятся следующие:

- Serverless Application Model (SAM), созданная в AWS, – позволяет создавать и развертывать бессерверные приложения через AWS CloudFormation. Подробности ищите по адресу: https://github.com/awslabs/serverless-application-model;
- Serverless Framework – реализует подход, напоминающий SAM, но дополнительно поддерживает другие платформы, такие как Microsoft Azure. Подробности ищите по адресу: https://serverless.com;
- Apex – еще один инструмент командной строки, помогающий развертывать бессерверные приложения, но поддерживает более обширный список языков программирования, таких как Go. Подробности ищите по адресу: http://apex.run.

ПРИМЕЧАНИЕ. Почти все, что можно сделать с Claudia, можно сделать и с использованием перечисленных альтернатив.

Вы, наверное, удивляетесь, почему мы решили использовать Claudia. Лучше всего наши мотивы объясняют часто задаваемые вопросы по Claudia:

- Claudia – это утилита развертывания, а не фреймворк. Она не абстрагирует службы AWS, а упрощает их использование. В отличие от Serverless и Seneca, Claudia не пытается изменить структуру или способ запуска проектов. Необязательный API Builder, который упрощает веб-маршрутизацию, является единственной дополнительной зависимостью времени выполнения, но он создавался с прицелом на минимализм и автономность. Микросервисные фреймворки имеют много хороших плагинов и расширений, помогающих выполнять стандартные задачи, но Claudia намеренно фокусируется только на развертывании. Одна из наших ключевых целей разработки – не вводить слишком много магии и позволить людям структурировать код так, как они хотят;
- Claudia *ориентирована на Node.js*. В отличие от Apex и других подобных инструментов развертывания, Claudia имеет гораздо более узкую область применения. Она работает только с Node.js, но работает очень хорошо. Универсальные фреймворки поддерживают больше окружений времени выполнения, но перекладывают на разработчика решение задач, специфических для языка. Поскольку Claudia фокусируется на Node.js, она автоматически устанавливает шаблоны для преобразования параметров и результатов в объекты, которые код на JavaScript может легко использовать, и действует именно так, как ожидают разработчики на JavaScript.

Дополнительные подробности вы найдете по адресу: https://github.com/claudiajs/claudia/blob/master/FAQ.md.

Цель этой книги состоит в том, чтобы научить вас думать в терминах бессерверных вычислений и разрабатывать и развертывать бессерверные приложения. Непосредственное использование экосистемы AWS требует отвлечения внимания на множество побочных аспектов, таких как обучение взаимодействию и настройке различных частей платформы AWS. Claudia не стремится заменить AWS SDK, но, построенная на его основе, она позволяет выполнить большинство рутинных задач одной командой.

Claudia отдает предпочтение коду перед настройками. В результате в библиотеке почти нет настроек. Это облегчает обучение и исследование. Написание качественного приложения требует правильного тестирования; наличие большого количества настроек не означает, что их не нужно тестировать.

Claudia имеет минимальный набор удобных команд для создания бессерверных приложений. Две основные идеи Клаудии – минимум магии и прозрачное отображение происходящего при вызове команды.

Несмотря на немногочисленный API, библиотека Claudia позволяет выполнять множество действий: создавать бессерверные приложения с нуля, переносить текущие приложения на Express.js в бессерверное окружение и даже создавать свои собственные бессерверные чат-боты и голосовые помощники.

1.7. Когда и где использовать бессерверные вычисления

Бессерверная архитектура – не панацея от всех болезней. Она не решает всех проблем и может не решить ваших.

Например, если вы создаете приложение, которое интенсивно использует веб-сокеты, бессерверные вычисления не для вас. Приложение в AWS Lambda может работать до 15 минут, после чего не сможет продолжать принимать сообщения через веб-сокеты.

Контейнеры запускаются довольно быстро, но не мгновенно. На запуск контейнера может уходить до нескольких десятков миллисекунд, что для некоторых приложений может оказаться неприемлемым.

Отсутствие конфигурации является одним из главных преимуществ бессерверных вычислений, но это преимущество может стать серьезным препятствием для некоторых типов приложений. Если вы решите создать приложение, требующее конфигурации на уровне системы, вам лучше рассмотреть возможность использования традиционного подхода. AWS Lambda поддерживает возможность настройки до некоторой степени; вы можете предоставить статический двоичный файл и использовать Node.js для его вызова, но во многих случаях это может быть сопряжено с большими накладными расходами.

Другим важным недостатком является так называемое замыкание на поставщике. Сами функции не являются большой проблемой, потому что это обычные функции Node.js, но если вы решите приложение целиком реализовать как бессерверное, с некоторыми службами придется повозиться. Однако

это распространенная проблема – она характерна не только для бессерверных вычислений, и ее можно минимизировать, выбрав хорошую архитектуру для приложения.

Тем не менее бессерверные вычисления имеют больше преимуществ, чем недостатков, и остальная часть этой книги показывает некоторые из хороших вариантов использования.

В заключение

- Бессерверная архитектура абстрагирует серверы от разработки программного обеспечения.
- Бессерверное приложение отличается от традиционного тем, что управляется событиями и автоматически распределяется и масштабируется.
- Есть несколько вариантов бессерверных архитектур, и наиболее совершенным является Amazon AWS Lambda.
- AWS Lambda – это платформа бессерверных вычислений, управляемых событиями, позволяющая запускать функции, написанные на Node.js, Python, C# или Java и других языках JVM.
- AWS Lambda имеет определенные ограничения, например на время выполнения, которое можно увеличить до 15, и на объем доступной памяти, который можно увеличить до 3 Гбайт.
- Самым сложным в бессерверных вычислениях является развертывание функций в AWS и их настройка.
- Некоторые инструменты и фреймворки могут сделать процесс развертывания и настройки более простым. Самым простым из них является библиотека Claudia в комплексе с ее API Builder и Bot Builder.
- Claudia – это инструмент командной строки, предлагающий минимальный набор удобных команд для сборки бессерверных приложений.
- Бессерверная архитектура не является панацеей, и в некоторых ситуациях это не лучший выбор, например для приложений реального времени с веб-сокетами.

Глава 2

Создание первого бессерверного API

Эта глава охватывает следующие темы:

- создание и развертывание API с помощью Claudia;
- как Claudia развертывает API в AWS;
- как работает API Gateway.

Главная цель данной главы – помочь вам создать свой первый бессерверный API и развернуть его в AWS Lambda и API Gateway с помощью Claudia. Вы также увидите различия в структуре традиционных и бессерверных приложений и будете лучше понимать Claudia, когда узнаете, что эта библиотека делает за кулисами. Чтобы извлечь максимальную пользу из этой главы, вы должны понимать основные идеи бессерверных вычислений, описанные в главе 1.

2.1. Приготовление пиццы из ингредиентов: сборка API

Тетушка Мария счастлива и благодарна, что вы решили помочь ей встать на ноги. Она даже приготовила вам свою знаменитую пиццу пеперони! (Хорошо, если вы сейчас не голодны!)

У тетушки Марии уже есть веб-сайт, поэтому от вас требуется создать серверное приложение – точнее API, – чтобы ее клиенты могли просматривать и заказывать пиццу. API будет отвечать за передачу информации о пиццах и порядке оформления заказа, а также за обработку заказов на пиццу. Позднее тетушка Мария хотела бы добавить также мобильное приложение, которое будет использовать ваш API.

На начальном этапе наши первые конечные точки API будут выполнять простую бизнес-логику и возвращать статические объекты JSON. На рис. 2.1 представлена общая схема первоначальной структуры приложения и поток HTTP-запросов через API.

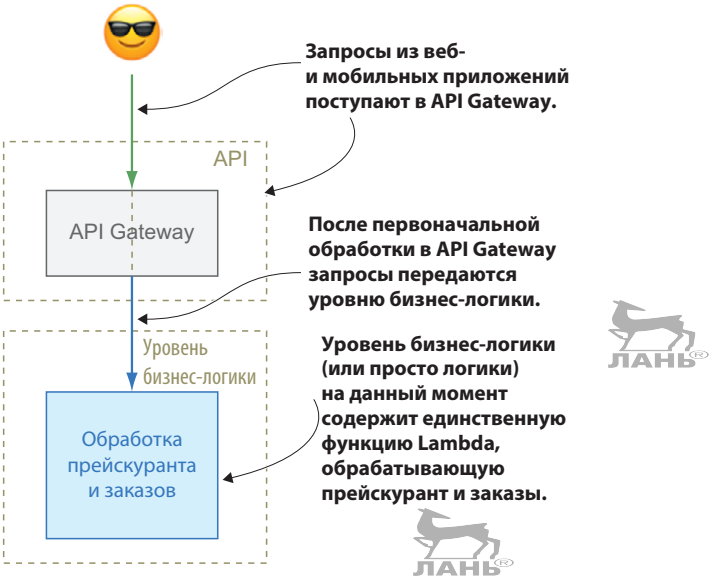


Рис. 2.1. Общая схема Pizza API, который мы построим в этой главе

Вот список функций, которые будет поддерживать первоначальный вариант API:

- вывод прейскуранта с изображениями разных пицц;
- прием заказа на доставку пиццы;
- создание заказа;
- изменение/корректировка заказа;
- отмена заказа.

Все это довольно маленькие и простые функции, поэтому реализуем их в одной функции Lambda.

Даже если вы чувствуете, что эти функции следует отделить друг от друга, прямо сейчас мы не будем делать этого и просто поместим все функции в одну функцию Lambda, потому что все функции тесно связаны. Если бы мы дополнительно следили за запасами, мы с самого начала создали бы для этого отдельную функцию.

Для каждой из перечисленных функций необходимо определить отдельный маршрут к соответствующему обработчику. Маршрутизацию можно реализовать вручную, но в Claudia есть инструмент, который поможет решить эту задачу: Claudia API Builder.

Claudia API Builder – это инструмент, помогающий обрабатывать входящие запросы и ответы API Gateway, а также их конфигурацию, контекст и параметры, и позволяет определить внутренние маршруты в функции Lambda. Он имеет Express-подобный синтаксис конечной точки, поэтому, если вы знакомы с Express, использование Claudia API Builder не вызовет у вас сложностей.

На рис. 2.2 показана более подробная схема маршрутизации с функциями обработки прейскуранта и заказов внутри функции Lambda, которая будет построена с помощью Claudia API Builder. На рисунке видно, что после получения запросов от API Gateway Claudia API Builder пересылает их по определенным вами маршрутам соответствующим обработчикам.

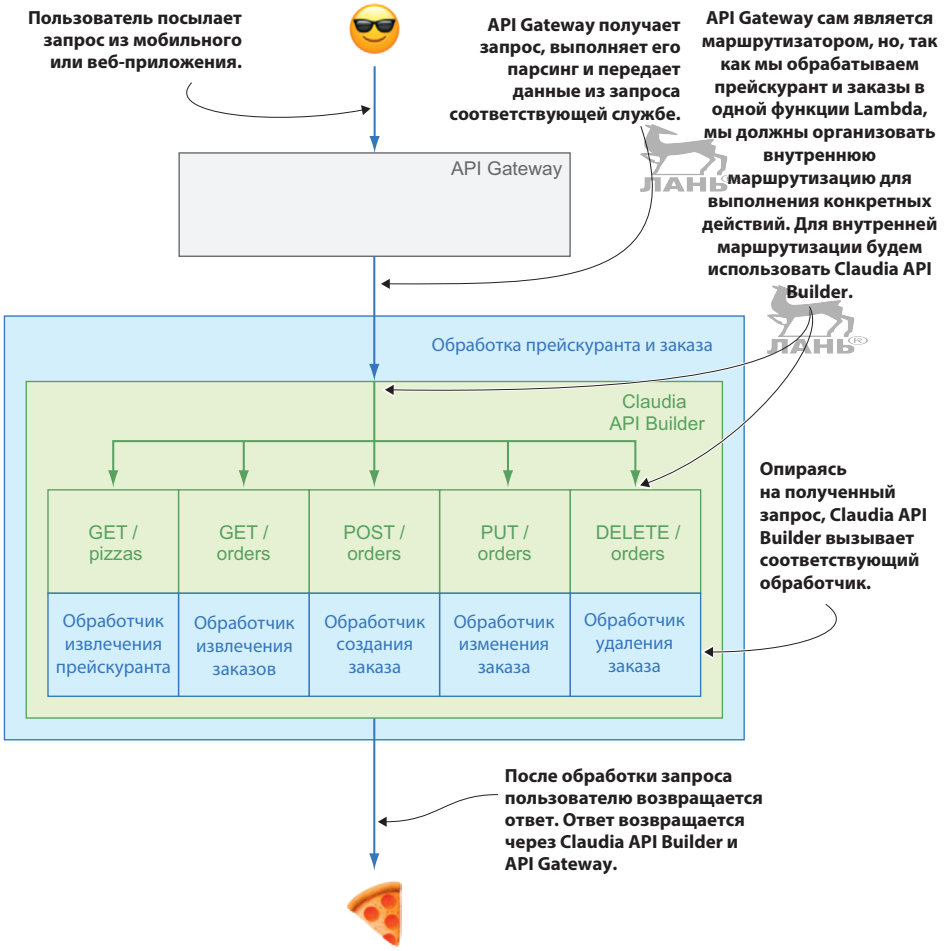


Рис. 2.2. Структура функции AWS Lambda, обрабатывающей прейскурант и заказы

ПРИМЕЧАНИЕ. На момент написания этих строк AWS API Gateway можно было использовать в двух режимах:

- с моделями и отображаемыми шаблонами для запросов и ответов;
- с промежуточной логикой.

Claudia API Builder использует промежуточную логику, которая извлекает информацию из HTTP-запросов и преобразует ее в представление, более удобное для JS-разработчика.

За более подробной информацией о режимах работы AWS API Gateway обращайтесь к официальной документации: http://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-method-settings.html.

2.1.1. Какие пиццы можно заказать?

Первый метод в нашем Pizza API реализует службу `GET`, которая возвращает прейскурант со всеми доступными для заказа пиццами. Чтобы создать этот метод, вам потребуется:

- получить учетную запись AWS и создать файл с настройками;
- установить Node.js и диспетчер пакетов NPM;
- установить Claudia с помощью NPM как глобальную зависимость.

Если прежде вам не доводилось выполнять эти шаги или вы не уверены, выполнили ли их, перейдите к приложению А, которое проведет вас через каждый этап процесса установки.

> **ПРИМЕРЫ КОДА.** С этого момента вы увидите много примеров программного кода. Мы настоятельно рекомендуем опробовать их все, даже если они кажутся вам знакомыми. Вы можете использовать ваш любимый текстовый редактор, если не указано иное.

Теперь, когда все готово, начнем с создания пустой папки для нашего первого бессерверного приложения. Вы можете назвать папку проекта как угодно, но в этой книге мы будем использовать имя `pizza-api`. После создания папки откройте терминал, перейдите в созданную папку и инициализируйте приложение Node.js. После инициализации установите модуль `claudia-api-builder` с помощью NPM, как описано в приложении А.

Следующий шаг – создание точки входа в приложение. Создайте файл с именем `api.js` в папке `pizza-api` и откройте его в текстовом редакторе.

> **СИНТАКСИС ES6 В ПРИМЕРАХ КОДА.** Все примеры кода в книге написаны с использованием синтаксиса ES6/ES2015. Если вы незнакомы с такими особенностями ES6, как стрелочные функции и/или шаблонные строки, прочитайте книгу Уэса Хигби (Wes Higbee) «*ES6 in Motion*» (Manning) или второе издание «*Secrets of the JavaScript Ninja*»[1] Джона Резига (John Resig).

Для создания внутреннего маршрута необходимо создать экземпляр Claudia API Builder, потому что это класс, а не функция. Поэтому в начале файла `api.js` создайте экземпляр `claudia-api-builder`.

Теперь можно задействовать встроенный маршрутизатор Claudia API Builder. Для реализации маршрута `GET /pizzas` необходимо использовать ме-

[1] *Резиг Джон, Бибо Бер, Марас Иосип.* Секреты JavaScript ниндзя. 2-е изд. М.: Вильямс, 2017. ISBN: 978-5-9908911-8-0. – *Прим. перев.*

тод `get` экземпляра Claudia API Builder. Метод `get` получает два аргумента: маршрут и функцию-обработчик. В качестве маршрута передайте строку `/pizzas`, а в качестве обработчика – анонимную функцию.

Анонимная функция-обработчик для Claudia API Builder имеет одно существенное отличие, по сравнению с Express.js. В Express.js вы передаете в функцию обратного вызова аргументы с ответом и запросом, но в Claudia API Builder функция обратного вызова принимает только запрос. Чтобы отправить ответ, достаточно вернуть его в виде возвращаемого значения.

Обработчик маршрута `GET /pizzas` должен вернуть список пицц (прейскурант), но пока мы будем возвращать статический массив с названиями пицц, которые готовятся в пиццерии тетушки Марии: Capricciosa (Капричоза), Quattro Formaggi (Четыре сыра), Napoletana (Наполетана) и Margherita (Маргарита).

Наконец, нужно экспортировать экземпляр API, который Claudia API Builder встраивает в функцию Lambda как промежуточный обработчик.

На данный момент код должен выглядеть, как показано в листинге 2.1.

Листинг 2.1. Обработчик `GET /pizzas` в Pizza API

```
'use strict'

const Api = require(''claudia-api-builder')
const api = new Api()

api.get('/pizzas', () => {
  return [
    'Capricciosa',
    'Quattro Formaggi',
    'Napoletana',
    'Margherita'
  ]
})
module.exports = api
```

Подключение модуля Claudia API Builder.

Создание экземпляра Claudia API Builder.

Определение маршрута и обработчика.

Возврат простого списка пицц.

Экспорт экземпляра Claudia API Builder.

Вот и все, что нужно для создания простой бессерверной функции. Однако, перед тем как открыть шампанское и отпраздновать это событие, развернем наш код в функции Lambda. Для этого вернемся к терминалу и воспользуемся мощью библиотеки Claudia.

Поскольку одной из основных целей Claudia является развертывание одной командой, для развертывания нашего API требуется выполнить лишь одну простую команду `claudia create`. Эта команда принимает два параметра: регион AWS, в котором вы хотите развернуть API, и точку входа в приложение. Параметры передаются в виде флагов, поэтому для развертывания API просто выполните команду `claudia create` с флагами `--region` и `--api-module`, как показано в листинге 2.2. Подробное описание команды `claudia create` вы найдете в разделе 2.2.

ВЫПОЛНЕНИЕ КОМАНД В WINDOWS. Некоторые команды в книге разбиты на несколько строк для удобства чтения и комментирования. Если вы пользуетесь ОС Windows, вам может понадобиться объединить эти команды в одну строку и удалить символ обратного слеша (\).

Листинг 2.2. Развертывание API в AWS Lambda и API Gateway с помощью Claudia

```
claudia create \                    ← Создает и развертывает новую функцию Lambda.
  --region eu-central-1 \           ← Определяет регион, где должна быть развернута функция.
  --api-module api                  ← Сообщает библиотеке Claudia, что создается API с точкой входа api.js.
```

Выберите регион, ближайший к вашим пользователям, чтобы минимизировать задержки. Пиццерия тетушки Марии находится во Франкфурте (Германия), поэтому ближайший регион называется `eu-central-1`. Все доступные регионы перечислены в официальной документации AWS: http://docs.aws.amazon.com/general/latest/gr/rande.html#lambda_region.

Файл `api.js` служит точкой входа в наш API. Claudia автоматически добавит расширение `.js`, поэтому укажите имя точки входа как `api`, без расширения.

ПРИМЕЧАНИЕ. Имя и местоположение точки входа вы можете выбирать по своему усмотрению; достаточно указать правильный путь к точке входа в команде `claudia create`. Например, если вы дадите файлу имя `index.js` и поместите его в папку `src`, тогда флаг команды Claudia должен иметь вид: `--api-module src/index`.

Через минуту или чуть больше Claudia развернет наш API. Вы увидите ответ, как показано в листинге 2.3. В ответе имеется полезная информация о развернутой функции Lambda и нашем API, такая как базовый URL, имя функции Lambda и регион.

ПРОБЛЕМЫ РАЗВЕРТЫВАНИЯ. Если при развертывании вы столкнетесь с проблемами, например связанными с учетной записью, проверьте еще раз все настройки, которые описываются в приложении А.

Листинг 2.3. Ответ на команду `claudia create`

```
{
  "lambda": {                       ← Информация о функции Lambda.
    "role": "pizza-api-executor",
    "name": "pizza-api",
    "region": "eu-central-1"
  },
```

```
  "api": {
    "id": "g8fhlgccof",
    "module": "api",
    "url": "https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/latest"
  }
}
```

Информация об API.

Базовый URL.

В процессе развертывания Claudia создаст файл `claudia.json` в корневом каталоге проекта и поместит в него похожую информацию, но без базового URL. Этот файл используется библиотекой Claudia, чтобы связать код API с конкретной функцией Lambda и экземпляром API Gateway. Файл предназначен только для библиотеки Claudia; не изменяйте его вручную.

Теперь пришло время «попробовать на вкус» наш API. Для этого можно воспользоваться любым веб-браузером. Просто введите в адресную строку базовый URL из ответа, который вернула команда `claudia create`, не забыв добавить к нему свой маршрут. Он должен выглядеть примерно так: https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/latest/pizzas. Когда вы откроете эту ссылку в браузере, вы должны увидеть следующее:

```
["Capricciosa","Quattro Formaggi","Napoletana","Margherita"]
```

> **АДРЕСА URL ДЛЯ ПРИМЕРОВ В КНИГЕ.** Вместо `latest` URL для каждого примера в книге будет содержать разные версии в формате: `chapterX_Y`, где `X` – номер главы, а `Y` – номер примера в этой главе. Мы сделали это, чтобы вы могли запускать примеры, просто скопировав URL из книги. При опробовании своих примеров ваши URL должны содержать `latest` вместо `chapterX_Y`.
>
> Например, первый пример доступен по URL: https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_1/pizzas.

Поздравляю – вы только что создали бессерверный API с помощью Claudia! Если это ваш первый опыт, можете гордиться собой. А сейчас приостановимся ненадолго.

2.1.2. Структурирование API

Прежде чем добавлять новые возможности, всегда следует потратить несколько минут на переосмысление структуры и организации API. Обработка всех маршрутов в основном файле затрудняет программирование и поддержку, поэтому в идеале обработчики следует отделить от маршрутизации. Маленькие файлы с кодом легче понять и сопровождать, чем один большой файл.

На момент написания этих строк не существовало каких-либо конкретных рекомендаций по организации приложений. Кроме того, Claudia дает вам полную свободу в этом отношении. Так как для обработки прейскуранта и заказов не потребуется много кода, все обработчики маршрутов для нашего

Pizza API можно поместить в отдельную папку и оставить в файле `api.js` только маршруты. Кроме того, поскольку каждая пицца в прейскуранте должна будет иметь больше атрибутов, чем просто название, его следует переместить в отдельный файл. Можно пойти еще дальше и создать папку для данных, как мы сделали это для списка пицц, упоминавшегося выше. После применения этих рекомендаций структура нашего кода должна выглядеть примерно так, как показано на рис. 2.3.

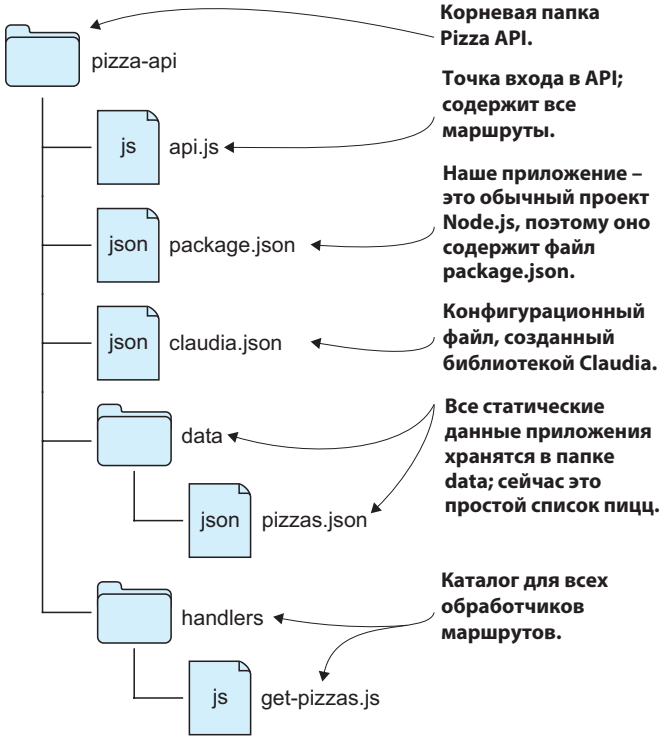


Рис. 2.3. Первый вариант структуры проекта Pizza API

Для начала переместим прейскурант в отдельный файл и добавим в него дополнительную информацию: идентификатор пиццы и ингредиенты. Для этого создайте в корне проекта Pizza API папку с именем `data`. Затем создайте в новой папке файл с именем `pizzas.json`. Добавьте в новый файл код из листинга 2.4.

Листинг 2.4. Прейскурант в формате JSON с информацией о пиццах

```
[
  {
    "id": 1,
    "name": "Capricciosa",
    "ingredients": [
```

Этот файл JSON содержит массив объектов пицц.

Для каждой пиццы определен ее числовой идентификатор, название и список ингредиентов.

```
      "tomato sauce", "mozzarella", "mushrooms", "ham", "olives"
    ]
  },
  {
    "id": 2,
    "name": "Quattro Formaggi",
    "ingredients": [
      "tomato sauce", "mozzarella", "parmesan cheese", "blue cheese", "goat cheese"
    ]
  },
  {
    "id": 3,
    "name": "Napoletana",
    "ingredients": [
      "tomato sauce", "anchovies", "olives", "capers"
    ]
  },
  {
    "id": 4,
    "name": "Margherita",
    "ingredients": [
      "tomato sauce", "mozzarella"
    ]
  }
]
```

Далее поместим обработчик `getPizzas` в отдельный файл. Создайте в корне проекта папку с именем `handlers` и внутри нее создайте файл с именем `get-pizzas.js`.

В новый файл `get-pizzas.js` поместите обработчик `getPizzas`, возвращающий прейскурант из листинга 2.4. Для этого вам потребуется импортировать файл JSON с прейскурантом. Затем вам нужно создать функцию-обработчик `getPizzas` и экспортировать, чтобы получить возможность сослаться на нее из файла с точкой входа. Затем, вместо того чтобы просто возвращать прейскурант, сделайте еще один шаг и верните информацию только для одной пиццы, если обработчику `getPizzas` был передан параметр с идентификатором пиццы. Чтобы вернуть только одну пиццу, можно использовать метод `Array.find`, который отыщет пиццу по идентификатору. Если он найдет пиццу, верните ее в качестве результата. Если пицца с запрошенным идентификатором не будет найдена, верните признак ошибки.

Дополненный код обработчика должен выглядеть, как показано в листинге 2.5.

Листинг 2.5. Обработчик `getPizzas` с фильтром по числовому идентификатору в отдельном файле

```
const pizzas = require('../data/pizzas.json')

function getPizzas(pizzaId) {
  if (!pizzaId)
    return pizzas

  const pizza = pizzas.find((pizza) => {
    return pizza.id == pizzaId
  })

  if (pizza)
    return pizza

  throw new Error('The pizza you requested was not found')
}

module.exports = getPizzas
```

Импортировать прейскурант из каталога data.

Объявление функции-обработчика getPizzas.

Если идентификатор пиццы не указан, вернуть полный список.

Иначе найти пиццу по идентификатору.

Обратите внимание, что используется == вместо ===. Причина в том, что в pizzaId передается строка и поэтому нам не нужно строгое сравнение, так как в базе данных идентификаторы могут храниться как целые числа.

Если пицца не найдена, сгенерировать ошибку.

Экспортировать обработчик getPizzas.

Также удалите предыдущий код обработчика `getPizzas` из файла `api.js`. Удалите весь код, находящийся между инструкцией импортирования Claudia API Builder и инструкцией экспортирования экземпляра Claudia API Builder.

После строки с инструкцией импортирования Claudia API Builder добавьте инструкцию импортирования нового обработчика `get-pizzas` из папки `handlers`:

```
const getPizzas = require('./handlers/get-pizzas')
```

> **ПРИМЕЧАНИЕ.** Создайте также обработчик `GET`-маршрута для корневого пути (`/`), который должен возвращать статическое сообщение пользователю. Это необязательно, но мы настоятельно рекомендуем добавить его. Пользователю будет удобнее, если он получит дружественное сообщение, а не ошибку, обратившись к базовому URL вашего API.

Затем добавьте маршрут для получения списка пицц, но на этот раз используйте только что созданный обработчик `get-pizzas`. Для этого импортируйте файл с обработчиком в начале файла `api.js`. Если вы помните, наш обработчик `get-pizzas` может выполнить поиск пиццы по ее идентификатору, поэтому добавьте еще один маршрут, возвращающий одну пиццу. Оформите этот маршрут так, чтобы он принимал запрос `GET` с URL `/pizzas/{id}`. Часть `/{id}` – это параметр динамического маршрута, который сообщает обработчику, какой

идентификатор пиццы запрошен пользователем. Как и Express.js, Claudia API Builder поддерживает параметры динамического маршрута, но использует иной синтаксис: `/{id}` вместо `/:id`. Параметры динамического пути доступны в объекте `request.pathParams`. Наконец, если обработчик не нашел запрошенную пиццу, верните ошибку 404:

```
api.get('/pizzas/{id}', (request) => {
  return getPizzas(request.pathParams.id)
}, {
  error: 404
})
```

По умолчанию в ответ на все запросы API Gateway возвращает код HTTP 200. Claudia API Builder помогает изменить некоторые из умолчаний, например вернуть код 500 в случае ошибки, чтобы клиентское приложение могло обработать ошибку.

Чтобы вернуть ошибки, передайте третий параметр в функцию `api.get`. Например, в функцию обработки маршрута `get/pizza/{id}`, помимо пути и функции обработчика, можно передать объект с пользовательскими заголовками и кодами. Чтобы установить код ошибки 404, передайте объект со значением `error: 404`.

Дополненный файл `api.js` должен выглядеть, как показано в листинге 2.6.

Листинг 2.6. Дополненный файл `api.js`

```
'use strict'

const Api = require('claudia-api-builder')
const api = new Api()

const getPizzas = require('./handlers/get-pizzas')

api.get('/', () => 'Welcome to Pizza API')

api.get('/pizzas', () => {
  return getPizzas()
})

api.get('/pizzas/{id}', (request) => {
  return getPizzas(request.pathParams.id)
}, {
  error: 404
})

module.exports = api
```

Импортировать обработчик get-pizzas из каталога handlers.

Добавить обработчик для базового URL, возвращающий простой статический текст, чтобы сделать API более дружественным.

Заменить встроенную функцию-обработчик вызовом новой импортированной функции.

Добавить маршрут для поиска пиццы по идентификатору.

Настроить коды успешной и неудачной обработки запроса.

Теперь снова развернем API. Чтобы заменить существующую функцию Lambda вместе с ее маршрутами в API Gateway, запустите в терминале команду `claudia update`:

```
claudia update
```

> **ПРИМЕЧАНИЕ.** Благодаря файлу `claudia.json` команда `claudia update` точно знает, в какую функцию Lambda следует развернуть файлы. Команду можно дополнительно настроить с помощью флага `--config`. За дополнительной информацией обращайтесь к официальной документации https://github.com/claudiajs/claudia/blob/master/docs/update.md.

Через минуту или чуть больше должен появиться ответ, подобный представленному в листинге 2.7. После выполнения команды и повторного развертывания приложения Claudia выведет в терминал некоторую полезную информацию о функции Lambda и API, включая имя функции, версию среды выполнения Node.js, величину тайм-аута, объем памяти, доступный функции, и базовый URL нашего API.

Листинг 2.7. Информация, возвращаемая командой claudia update

```
{
  "FunctionName": "pizza-api",          ← Имя функции AWS Lambda.
  "Runtime": "nodejs6.10",              ← Версия среды выполнения Node.js.
  "Timeout": 3,                         ← Тайм-аут для функции (в секундах).
  "MemorySize": 128,                    ← Объем памяти, доступной для функции.
  "Version": "2",                       ← Версия развертывания.
  "url": "https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/
      chapter2_2",                                  ← Базовый URL вашего API.
  "LastModified": "2017-07-15T14:48:56.540+0000",
  "CodeSha256": "0qhstkwwkQ4aEFSXhxV/zdiiS1JUIbwyKOpBup35l9M=",
  // Дополнительные метаданные
}
```

Открыв этот маршрут в браузере (который должен выглядеть примерно так: https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_2/pizzas), вы увидите массив со всеми объектами-пиццами из файла `data/pizza.js`.

Открыв другой маршрут (который должен выглядеть примерно так: https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_2/pizzas/1), вы увидите только первую пиццу, как показано ниже:

```
{"id":1,"name":"Capricciosa","ingredients":["tomato
     sauce","mozzarella","mushrooms","ham","olives"]}
```

Для проверки работы API попробуйте также запросить пиццу с несуществующим идентификатором. Для этого введите URL, например такой: https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_2/pizzas/42. В этом случае вы должны получить ответ:

```
{"errorMessage" : "The pizza you requested wasn't found"}
```

Отлично! Теперь наш Pizza API может возвращать прейскурант с пиццами клиентам тетушки Марии! Это добавит радости тетушке Марии, но наш API еще не закончен. Мы должны еще реализовать самую главную функцию: создание заказа на доставку пиццы.

2.1.3. Отправка заказа

Возможность создать заказ с помощью API очень важна для тетушки Марии. Даже не будучи столь же технически подкованной, как вы, она понимает, что это ускорит оформление заказов и поможет ей быстро обслуживать клиентов, проживающих в ее районе или даже во всем городе.

> **ПРИМЕЧАНИЕ.** В этом примере вы познакомитесь с базовой структурой приложения, которое для простоты демонстрации нигде не сохраняет полученные заказы. Поддержку хранилища мы добавим в главе 3.

Для функции приема заказов нам понадобится определить маршрут «создание заказа на пиццу» и обработчик «создание заказа», а это означает, что мы должны создать новый файл в папке `handlers` в нашем проекте Pizza API. Как обычно, старайтесь выбирать простые и осмысленные имена для файлов. В данном случае прекрасно подойдет имя `create-order.js`.

Сначала создайте новый файл для обработчика и откройте его в текстовом редакторе. Затем объявите функцию `createOrder` и экспортируйте ее в конце файла. Функция-обработчик должна принимать некоторую информацию о заказе в форме объекта заказа `order`. На данный момент этот объект имеет только два атрибута: идентификатор пиццы и адрес клиента, куда нужно доставить пиццу.

Первым делом нужно проверить наличие значений в атрибутах переданного объекта. Если какое-то значение отсутствует, вернем ошибку.

Следующим шагом должно бы быть сохранение заказа в базе данных, но пока мы не готовы к этому и поэтому просто вернем пустой объект, если объект заказа `order` прошел проверку. Можно было бы сохранить объект в файл, но функция Lambda может быть запущена в нескольких разных контейнерах, и мы не можем управлять этим процессом, поэтому очень важно исключить любые ссылки на локальное состояние. В следующей главе вы узнаете, как подключить бессерверную функцию к базе данных и сохранить заказ.

Содержимое файла `create-order.js` должно выглядеть, как показано в листинге 2.8.

Листинг 2.8. Обработчик запроса на создание заказа

```
function createOrder(order) {
  if (!order || !order.pizzaId || !order.address)
    throw new Error('To order pizza please provide pizza type and address
      where pizza should be delivered')

  return {}
}

module.exports = createOrder
```

Функция createOrder принимает объект заказа.

Если объект order не имеет идентификатора пиццы или адреса клиента, сгенерировать ошибку.

Иначе вернуть пустой объект.

Экспортировать функцию-обработчик.

Теперь, когда у нас есть обработчик для создания заказа, определим маршрут. В отличие от предыдущих маршрутов, этот должен принимать запросы `POST`. Для этого вернемся в файл `api.js`. По аналогии с `api.get`, в Claudia API Builder есть метод `api.post`, принимающий три аргумента: путь, функцию-обработчик и параметры.

> **ПРИМЕЧАНИЕ.** Помимо `GET`, Claudia API Builder поддерживает также HTTP-запросы `POST`, `PUT` и `DELETE`.

Путь, определяющий маршрут для создания нового заказа, запишем как `/orders`. В качестве функции-обработчика импортируем файл `create-order.js`, который мы только что создали в папке `handlers`. Наконец, в аргументе с параметрами передадим настроенные коды, соответствующие успеху и неудаче: 201 и 400 соответственно. Используйте атрибут `success`, чтобы добавить свой код для случая успешного завершения.

Тело запроса `POST` будет извлечено автоматически и передано в атрибуте `request.body`, то есть вам не нужно предпринимать никаких дополнительных действий для парсинга полученных данных.

Парсинг тела запроса POST

Парсинг тела запроса `POST` автоматически выполняет API Gateway. Claudia проверяет тело и нормализует его. Например, если запрос имеет тип содержимого `application/json`, Claudia превратит пустое тело в пустой объект JSON.

После добавления нового маршрута содержимое файла `api.js` должно выглядеть, как показано в листинге 2.9.

Листинг 2.9. Главный файл API с новыми маршрутами

```
'use strict'

const Api = require('claudia-api-builder')
const api = new Api()

const getPizzas = require('./handlers/get-pizzas')
const createOrder = require('./handlers/create-order')

api.get('/', () => 'Welcome to Pizza API')

api.get('/pizzas', () => {
  return getPizzas()
})

api.get('/pizzas/{id}', (request) => {
  return getPizzas(request.pathParams.id)
}, {
  error: 404
})

api.post('/orders', (request) => {
  return createOrder(request.body)
}, {
  success: 201,
  error: 400
})
module.exports = api
```

Импортировать обработчик create-order из каталога handlers.

Добавить маршрут POST /orders для создания заказа и передать request.body в обработчик.

Вернуть код «201 Created» в случае успеха.

Вернуть код «400 Bad Request» в случае ошибки.

Снова разверните API командой `claudia update`.

Тестирование запроса `POST` выглядит немного сложнее, чем запроса `GET`. Его нельзя проверить, просто открыв URL маршрута в браузере. По этой причине для проверки маршрутов `POST` следует использовать один из бесплатных инструментов тестирования HTTP, например `curl` или Postman.

> **ПРИМЕЧАНИЕ.** Начиная с этого момента, вы будете видеть команды `curl` для всех примеров, где требуется опробовать конечные точки API. При желании вы можете использовать любой другой инструмент по своему выбору.

Протестируем конечную точку `POST /orders` с помощью команды `curl`. В этой команде мы отправим пустое тело запроса, чтобы проверить ошибку проверки. Помимо тела запроса `POST`, нужно указать метод, заголовок, сообщающий нашему API, что запрос содержит данные в формате JSON, и полный URL, куда отправляется запрос.

curl и Postman

`curl` – это инструмент, используемый в командной строке или сценариях для передачи данных. Он также применяется в автомобилях, телевизорах, маршрутизаторах, принтерах, аудиотехнике, мобильных телефонах, планшетах, телевизионных приставках и медиапроигрывателях и является главным инструментом передачи данных через интернет для тысяч программных приложений, которыми ежедневно пользуются миллиарды людей. `curl` предназначен для работы без непосредственного взаимодействия с пользователем.

Postman – это приложение с графическим интерфейсом, которое также может помочь вам протестировать ваши API. Его применение может ускорить разработку, поскольку дает вам возможность конструировать запросы API и документацию методом тестирования. Имеются версии Postman для Mac, Windows и Linux, а также в виде плагина для Chrome.

ПРИМЕЧАНИЕ. По умолчанию `curl` не выводит код HTTP-ответа. Чтобы проверить возвращаемый код, используйте флаг `-w` и добавьте код HTTP после ответа API.

Формат команды показан в листинге 2.10. Эта команда создает запрос с пустым телом и таким образом позволяет проверить реакцию API на ошибочную ситуацию.

Листинг 2.10. Команда `curl` для тестирования маршрута `POST /orders route` (ошибочная ситуация)

```
curl -i \                                  ← Потребовать от команды curl вывести ответ с HTTP-заголовками.
  -H "Content-Type: application/json" \    ← Установить заголовок, сообщающий, что параметры запроса передаются в формате JSON.
  -X POST \                                ← Указать метод POST.
  -d '{}' https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/
     chapter2_3/orders                     ← Послать пустой объект в API по адресу /orders.
```

После запуска команды `curl` из листинга 2.10 в терминале должен появиться следующий ответ с несколькими дополнительными заголовками:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 104
Date: Mon, 25 Sep 2017 06:53:36 GMT

{"errorMessage":"To order pizza please provide pizza type and address where
     pizza should be delivered"}
```

Теперь, после проверки ошибки в случае отсутствия данных заказа, мы должны проверить успешный ответ. Для этого выполним аналогичную команду `curl`, изменив только тело запроса, указав в нем идентификатор пиццы и адрес клиента. Соответствующая команда `curl` показана в листинге 2.11. Она имеет правильно сформированное тело, поэтому мы можем проверить успешный ответ.

Листинг 2.11. Команда `curl` для тестирования маршрута `POST /orders route` (успешная ситуация)

```
curl -i \
  -H "Content-Type: application/json" \
  -X POST \
  -d '{"pizzaId":1,"address":"221B Baker Street"}' \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_3/orders
```

Послать в теле POST-запроса идентификатор пиццы и адрес клиента.

Эта команда вернет следующий ответ:

```
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 2
Date: Mon, 25 Sep 2017 06:53:36 GMT

{}
```

Этот ответ подтверждает правильную работу API.

Теперь, познакомившись с основами конструирования бессерверных API, пришло время посмотреть, что делает библиотека Claudia после запуска команды `claudia create`.

2.2. Как Claudia развертывает API

Предыдущие примеры наглядно демонстрируют одну из главных целей Claudia: дать возможность развертывать приложения одной командой. В этом инструменте нет ничего волшебного, то есть принцип действия каждой его команды легко объяснить.

На рис. 2.4 изображен поток событий, происходящих после запуска команды `claudia create`. Эта упрощенная диаграмма показывает только наиболее важные этапы процесса, чтобы его проще было понять. Кроме того, некоторые этапы можно пропустить или изменить их работу, добавив дополнительные флаги в команду `create`. Например, Claudia может пропустить первый этап и скопировать ваш код со всеми локальными зависимостями, если передать флаг `--use-local-dependencies`. Полный список флагов и параметров можно найти на странице: https://github.com/claudiajs/claudia/blob/master/docs/create.md.

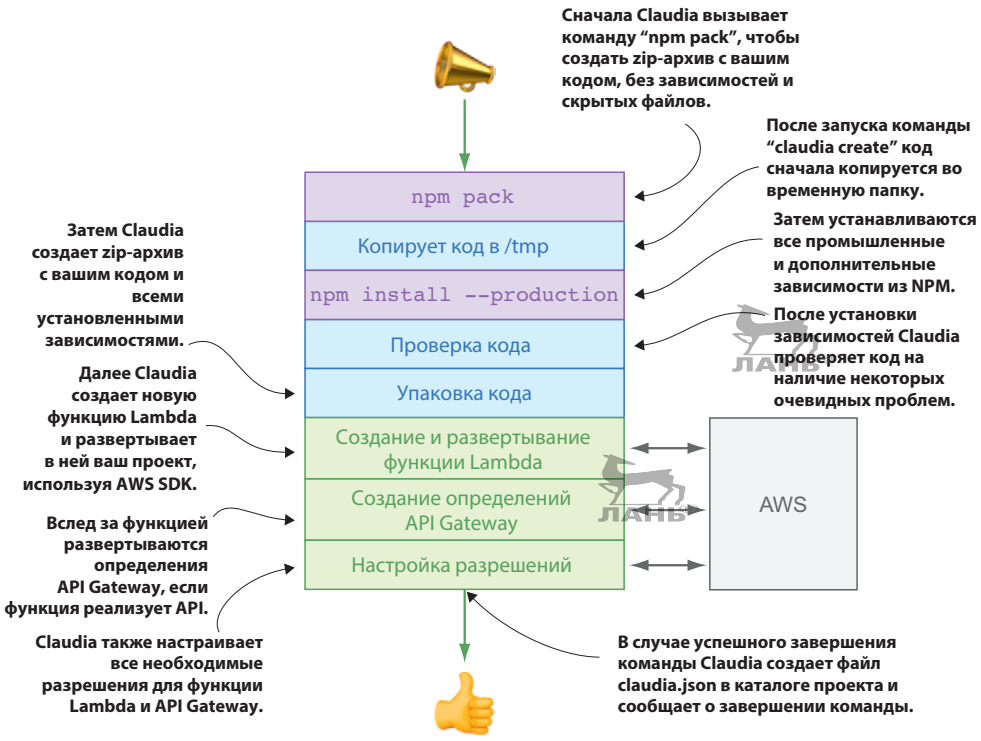


Рис. 2.4. Порядок работы команды claudia create

Сразу после запуска команды `claudia create` библиотека Claudia помещает ваш код без зависимостей и скрытых файлов в zip-архив, используя команду `npm pack`. Затем создает копию проекта во временной папке в локальной системе. Это гарантирует чистоту и воспроизводимость развертывания – развертывание всегда начинается с одной и той же известной точки, что предотвращает любые потенциальные проблемы, вызванные локальными зависимостями. На этом этапе Claudia игнорирует вашу папку `node_modules` и все файлы, которые игнорируются Git или NPM. Она также устанавливает промышленные и дополнительные зависимости командой `npm install --production`.

Так как для развертывания функции Lambda требуется, чтобы выгружаемый zip-архив содержал код со всеми зависимостями, перед упаковкой проекта в zip-файл Claudia устанавливает все промышленные и дополнительные зависимости NPM.

Так как отладка функций Lambda не самое простое дело, в чем вы убедитесь в главе 5, Claudia также проверяет ваш проект на наличие некоторых очевидных проблем, таких как опечатки или ссылки на неопределенные модули. Не принимайте этот этап всерьез, потому что он выполняет лишь самые поверхностные проверки. Если вы допустите опечатку или попытаетесь вызвать неопределенную функцию либо модуль внутри функции-обработчика, эта проверка не заметит ошибки. На следующем этапе Claudia создает zip-файл с вашим кодом и всеми зависимостями, установленными на первом этапе.

Последние три этапа, изображенных на рис. 2.4, выполняются параллельно. После создания zip-файла Claudia вызывает AWS API для создания функции Lambda и выгружает архив. Взаимодействие с платформой AWS осуществляется посредством модуля AWS SDK для Node.js. Прежде чем выгрузить код, Claudia создает нового пользователя с помощью IAM и настраивает его привилегии, необходимые для взаимодействий с AWS Lambda и API Gateway.

Пользователи IAM в AWS, роли и разрешения

Механизм идентификации и управления доступом в AWS (Identity and Access Management, IAM) помогает обеспечить безопасный доступ к службам и ресурсам AWS для ваших пользователей. С помощью IAM можно создавать учетные записи и группы пользователей AWS и управлять ими, а также определять привилегии, разрешающие или запрещающие пользователям или группам доступ к ресурсам AWS.

Подробное описание IAM выходит далеко за рамки этой книги, но мы настоятельно советуем познакомиться с этим механизмом, прежде чем вы перейдете к следующим главам. Для начала можно ознакомиться с официальной документацией: https://aws.amazon.com/iam/.

После установки и настройки функции Lambda Claudia настраивает экземпляр API Gateway, определяя маршруты и назначая необходимые разрешения.

Команда `claudia update` действует почти так же, как `claudia create`, но пропускает некоторые этапы, такие как создание роли и настройка разрешений.

Желающие глубже изучить работу Claudia и ее команд могут заглянуть в исходный код библиотеки: https://github.com/claudiajs/claudia.

Теперь, узнав, как действует Claudia, нам осталось разобраться с последним элементом мозаики – особенностями маршрутизации в API Gateway.

2.3. Управление трафиком: как работает API Gateway

В главе 1 вы узнали, что пользователи не могут взаимодействовать с AWS Lambda из-за пределов платформы AWS иначе, обратившись к некоторому механизму, который вызовет функцию. Одним из важнейших таких механизмов в Lambda является API Gateway.

Как показано на рис. 2.5, API Gateway действует подобно маршрутизатору или регулировщику дорожного движения. Он принимает HTTP-запросы (например, запрос к Pizza API от мобильного или веб-приложения), преобразует их в обобщенный формат и направляет в одну из ваших служб в AWS.

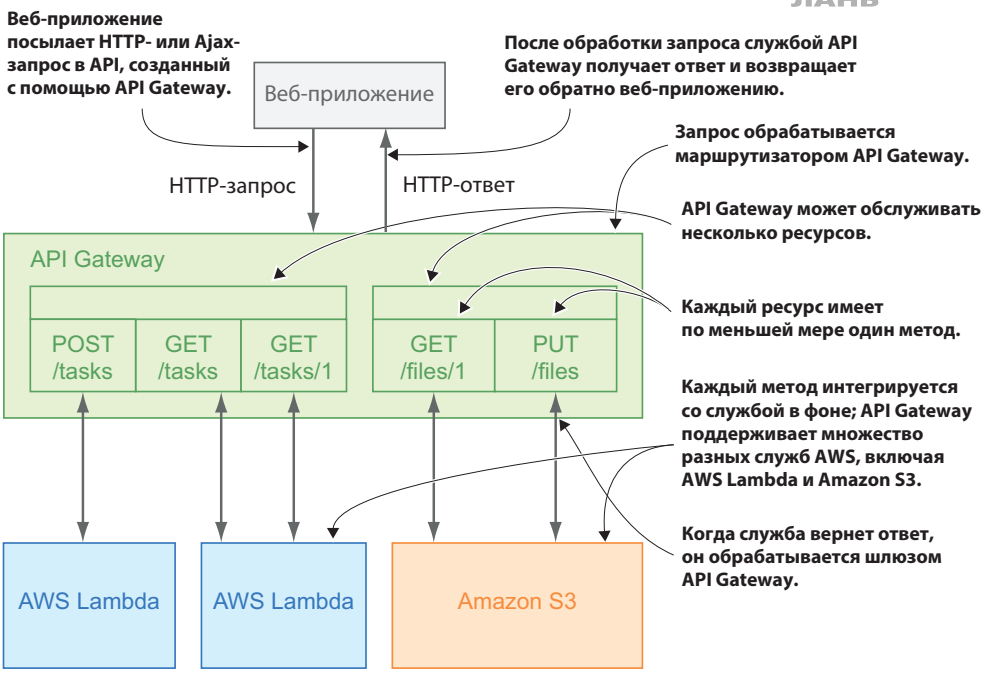


Рис. 2.5. API Gateway пересылает запросы вашим службам AWS

API Gateway может интегрироваться со множеством служб AWS, включая AWS Lambda и Amazon S3. Каждый API в API Gateway может подключаться к нескольким службам. Например, один маршрут может приводить к вызову функции Lambda, а другой – осуществлять взаимодействия с некоторыми другими службами.

API Gateway предлагает еще один подход к маршрутизации HTTP-запросов, который называется *прокси-маршрутизацией* (proxy router). Вместо создания отдельных маршрутов прокси-маршрутизатор пересылает все запросы единственной функции AWS Lambda. Этот подход удобно использовать для создания небольших API или когда желательно увеличить скорость развертывания, потому что создание и обновление большого числа маршрутов в API Gateway может длиться до нескольких минут, в зависимости от быстродействия вашего подключения к интернету и количества маршрутов.

2.4. Когда бессерверный API не является решением

Несмотря на довольно поверхностное знакомство, мы уже имели возможность убедиться, насколько просто создать бессерверный API с помощью Claudia.js и Claudia API Builder. Бессерверные API могут быть очень мощными и отличаются невероятной способностью к масштабированию, но иногда традиционные API оказываются более удачным решением, например:

- когда никакая дополнительная задержка в обработке запроса недопустима. Бессерверные приложения не могут гарантировать минимальную задержку;
- когда важно гарантировать определенный уровень доступности. В большинстве случаев AWS обеспечивает довольно высокий уровень доступности, но иногда этого бывает недостаточно;
- когда приложение выполняет тяжелые и продолжительные вычисления;
- когда API должен соответствовать специальным стандартам. AWS Lambda и API Gateway могут оказаться недостаточно гибкими для этого.

2.5. Опробование!

В конце каждой главы присутствует раздел «сделай сам». В большинстве глав вам будет предложено решить некоторую задачу, и вы должны попробовать реализовать ее самостоятельно. В этих разделах мы также будем представлять некоторые подсказки и возможные варианты решения.

2.5.1. Упражнение

В этой главе мы реализовали маршруты `GET /pizzas` и `POST /orders`. Чтобы API получился более функциональным, в него нужно добавить еще два маршрута: `PUT /orders` и `DELETE /orders`.

В этом первом упражнении попробуйте:

- создать обработчик, изменяющий заказ, и добавить маршрут к нему;
- создать обработчик, удаляющий заказ, и добавить маршрут к нему.

Вот несколько подсказок:

- для добавления маршрута `PUT` используйте метод `api.put` экземпляра Claudia API Builder;
- для добавления маршрута `DELETE` используйте метод `api.delete` экземпляра Claudia API Builder;
- оба метода принимают три аргумента: путь маршрута, функцию-обработчик и объект с параметрами;
- в обоих случаях пути содержат динамический параметр: числовой идентификатор заказа;
- обработчик `updateOrder` также требует передачи тела запроса с новыми параметрами заказа;
- так как пока мы не сохраняем заказы в базе данных, просто возвращайте пустой объект или обычное текстовое сообщение.

После выполнения упражнения структура файлов в Pizza API должна выглядеть, как показано на рис. 2.6.

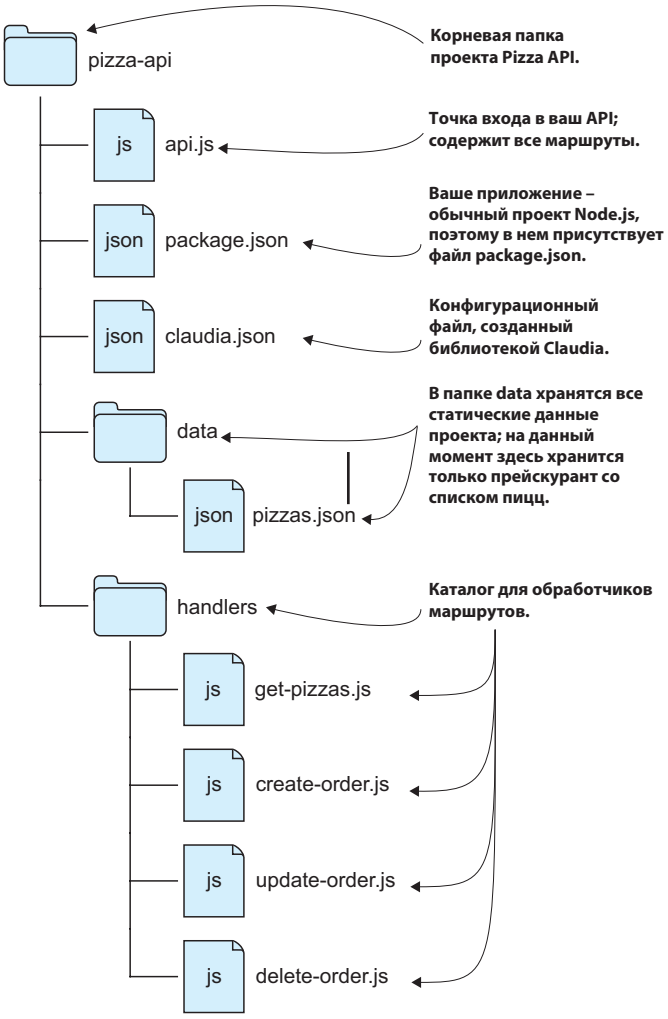


Рис. 2.6. Измененная структура файлов и каталогов проекта Pizza API

Если упражнение покажется вам слишком простым и вы захотите усложнить его, попробуйте добавить маршрут для получения списка заказов. Вариант решения этого задания мы не приводим, но вы можете заглянуть в исходный код примера проекта Pizza API, распространяемого вместе с книгой, чтобы сравнить свое решение с нашим.

2.5.2. Решение

Надеемся, что у вас получилось самим реализовать решение предложенного упражнения. Ниже приводится наше решение, чтобы вы могли сравнить его со своим.

В упражнении выше вам было предложено создать обработчик для изменения существующего заказа. Для этого сначала нужно создать в папке `handlers`

новый файл с именем `update-order.js`. В этом файле вы должны экспортировать функцию `updateOrder`, принимающую идентификатор заказа и параметры измененного заказа. Функция должна сгенерировать ошибку, если идентификатор или параметры заказа не указаны, иначе – вернуть признак успеха. В листинге 2.12 показано, как примерно должен выглядеть код.

Листинг 2.12. Обработчик изменения заказа

```
function updateOrder(id, updates) {
  if (!id || !updates)
    throw new Error('Order ID and updates object are required for updating
       the order')
  return {
    message: `Order ${id} was successfully updated`
  }
}

module.exports = updateOrder
```

Обработчик принимает идентификатор и параметры измененного заказа.

Если идентификатор или объект updates отсутствует, сгенерировать ошибку.

Иначе вернуть сообщение об успешном выполнении операции.

Экспортировать функцию-обработчик.

Создав функцию `updateOrder`, проделайте те же шаги, чтобы создать обработчик удаления заказа. Сначала создайте файл `delete-order.js` в папке `handlers`. Затем определите в нем экспортируемую функцию `deleteOrder`. Эта функция должна принимать идентификатор заказа. Если идентификатор не указан, функция должна генерировать ошибку; иначе вернуть пустой объект. В листинге 2.13 показано, как примерно должен выглядеть код.

Листинг 2.13. Обработчик удаления заказа

```
function deleteOrder(id) {
  if (!id)
    throw new Error('Order ID is required for deleting the order')
  return {}
}

module.exports = deleteOrder
```

Обработчик принимает идентификатор заказа.

Если идентификатор отсутствует, сгенерировать ошибку.

Иначе вернуть пустой объект.

Экспортировать функцию-обработчик.

Следующий шаг после реализации обработчиков – импортировать их в `api.js` и создать маршруты для изменения и удаления заказов.

Чтобы определить маршрут изменения заказа, используйте метод `api.put` и путь `/orders/{id}`; затем назначьте функцию-обработчик и верните код 400 в случае ошибки. Вы не сможете просто передать функцию-обработчик, созданную на предыдущем шаге, потому что она не принимает полный объект запроса; вместо этого нужно передать анонимную функцию, которая вызывает `updateOrder` с идентификатором заказа, извлеченным из тела запроса. Марш-

рут `DELETE /orders` оформляется так же, но с двумя отличиями: для этого используется метод `api.delete` и не требуется передавать тело запроса в функцию `deleteOrder`.

После выполнения этого шага содержимое файла `api.js` должно выглядеть как в листинге 2.14.

Листинг 2.14. Pizza API с маршрутами `PUT/orders` и `DELETE/orders`

```
'use strict'

const Api = require('claudia-api-builder')
const api = new Api()

const getPizzas = require('./handlers/get-pizzas')
const createOrder = require('./handlers/create-order')
const updateOrder = require('./handlers/update-order')
const deleteOrder = require('./handlers/delete-order')

// Определения маршрутов
api.get('/', () => 'Welcome to Pizza API')

api.get('/pizzas', () => {
  return getPizzas()
})

api.get('/pizzas/{id}', (request) => {
  return getPizzas(request.pathParams.id)
}, {
  error: 404
})

api.post('/orders', (request) => {
  return createOrder(request.body)
}, {
  success: 201,
  error: 400
})

api.put('/orders/{id}', (request) => {
  return updateOrder(request.pathParams.id, request.body)
}, {
  error: 400
})

api.delete('/orders/{id}', (request) => {
```

Импортировать обработчик update-order из каталога handlers.

Импортировать обработчик delete-order из каталога handlers.

Добавить маршрут PUT /orders и подключить обработчик.

Маршрут возвращает код 400 в случае ошибки.

Добавить маршрут DELETE /orders и подключить обработчик.

```
  return deleteOrder(request.pathParams.id)
}, {
  error: 400
})

module.exports = api
```

Маршрут возвращает код 400 в случае ошибки.

Как обычно, откройте терминал, перейдите в папку `pizza-api` и запустите команду `claudia update`, чтобы обновить функцию Lambda и определение API Gateway.

По завершении обновления Pizza API вы можете использовать команды `curl` из листингов 2.15 и 2.16, чтобы протестировать новые конечные точки. Эти команды очень похожи на команду, использовавшуюся для отправки запроса `POST`, но имеют следующие отличия:

- отличается HTTP-метод: для изменения заказа используется метод `PUT`, для удаления – метод `DELETE`;
- для изменения заказа требуется передать непустое тело запроса с измененным заказом;
- для удаления заказа передавать что-либо в теле запроса не требуется.

Обе команды должны завершаться успехом.

Листинг 2.15. Команда `curl` для проверки маршрута `PUT/orders/{id}`

```
curl -i \
  -H "Content-Type: application/json" \
  -X PUT \
  -d '{"pizzaId":2}' \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_4/orders/42
```

Посылает запрос PUT.

Добавляет дополнительные ингредиенты.

Идентификатор заказа как параметр пути.

Листинг 2.16. Команда `curl` для проверки маршрута `DELETE/orders/{id}`

```
curl -i \
  -H "Content-Type: application/json" \
  -X DELETE \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter2_4/orders/42
```

Посылает запрос DELETE.

Идентификатор заказа как параметр пути.

Команды должны вывести в терминале ответы `{"message":"Order 42 was successfully updated"}`, `{}` соответственно, оба с кодом 200.

В заключение

- Библиотека Claudia позволяет развернуть API в API Gateway и AWS Lambda одной командой.
- Для обновления API с помощью Claudia также достаточно одной команды.
- Бессерверный API в AWS Lambda не требует какой-то особой структуры каталогов.
- API Gateway действует подобно регулировщику и может вызывать разные службы.
- Чтобы связать несколько маршрутов с единственной функцией AWS Lambda, необходимо организовать внутреннюю маршрутизацию.
- Маршрутизатор в Claudia API Builder идентичен маршрутизаторам в других популярных веб-библиотеках Node.js.
- Бессерверные API обладают широкими возможностями, но они не панацея, поэтому иногда традиционные API могут оказаться лучшим решением.

Глава 3

Простота асинхронных операций с Promise()

Эта глава охватывает следующие темы:

- выполнение асинхронных операций с помощью Claudia;
- основы асинхронных вычислений в JavaScript;
- подключение к DynamoDB из Claudia и AWS Lambda.

В предыдущей главе мы создали простой API для обработки прейскуранта и заказов. Вы узнали, что, в отличие от традиционного сервера Node.js, функции AWS Lambda не сохраняют состояния между вызовами. Следовательно, для хранения заказов на доставку пиццы и любых других данных требуется база данных или внешняя служба.

Node.js действует асинхронно, поэтому сначала мы посмотрим, как бессерверные вычисления влияют на асинхронные взаимодействия: как они выполняются в Claudia, и, что более важно, познакомимся с рекомендуемым способом разработки бессерверных приложений. Освоив эти идеи, вы увидите, насколько просто подключить AWS Lambda к внешней службе и как использовать эту возможность для организации хранения заказов в AWS DynamoDB.

Так как наш мозг плохо справляется с асинхронными операциями, да и книги пишутся в синхронной манере, будем двигаться вперед постепенно и последовательно.

3.1. Хранение заказов

Дзынь-дзынь! Мы только что позвонили тетушке Марии. Она впечатлена нашей скоростью, хотя по-прежнему не может использовать приложение, поскольку мы не реализовали сохранение заказов на пиццу. Она еще должна использовать старый метод с ручкой и бумагой. Чтобы закончить создание базовой версии Pizza API, мы должны добавить хранение заказов.

Перед началом разработки всегда следует выяснить, какие данные должны храниться. В нашем случае элементарный заказ пиццы определяется выбранной пиццей, адресом доставки и состоянием заказа. Для наглядности в таких случаях информацию обычно изображают в виде диаграммы. Итак, в качестве небольшого упражнения отвлекитесь на минутку и попытайтесь нарисовать ее самостоятельно.

Ваша диаграмма должна быть похожа на рис. 3.1.

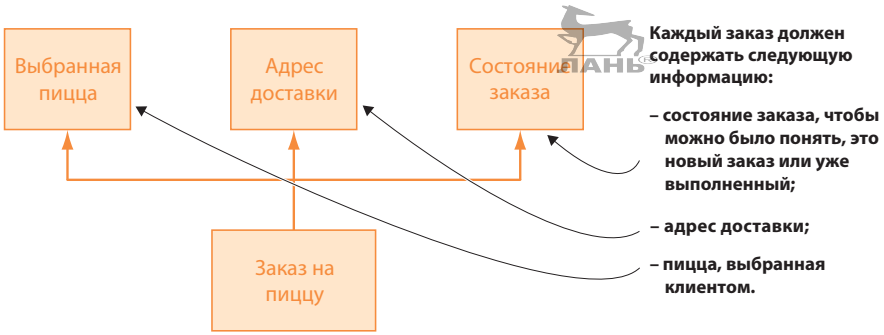


Рис. 3.1. Элементарный заказ на доставку пиццы

Теперь, получив представление о том, какие данные должны сохраняться, посмотрим, как их следует организовать для хранения в базе данных. Как вы уже знаете, мы не можем полагаться на AWS Lambda для сохранения состояния, а это означает, что хранить информацию о заказе в файловой системе Lambda не получится.

В традиционном приложении Node.js мы использовали бы одну из популярных баз данных, таких как MongoDB, MySQL или PostgreSQL. В мире бессерверных вычислений каждый из поставщиков услуг имеет свою комбинацию систем хранения данных. AWS не имеет готового решения ни для одной из перечисленных баз данных.

В качестве простейшей альтернативы можно использовать Amazon DynamoDB – популярную базу данных NoSQL, которая легко подключается к AWS Lambda.

> **ПРИМЕЧАНИЕ.** AWS Lambda не ограничивается поддержкой только DynamoDB, вы можете использовать также другие базы данных, но обсуждение этого вопроса выходит за рамки данной книги.

Что такое DynamoDB?

DynamoDB – это полноценная, проприетарная служба баз данных NoSQL, предлагаемая Amazon в рамках портфеля услуг AWS. DynamoDB предоставляет аналогичную модель данных и получила свое название от Dynamo – высокодоступной структурированной системы хранения пар ключ/значение.

Проще говоря, DynamoDB – это еще один строительный блок с базой данных для бессерверных приложений. DynamoDB в мире баз данных NoSQL – это то же самое, что AWS Lambda в мире вычислительных функций: полноценно, автоматически масштабируемое и относительно дешевое решение для облачных баз данных.

DynamoDB хранит данные в таблицах. Таблица – это набор данных. Каждая таблица содержит несколько элементов. Элемент представляет нечто единое целое и описывается группой атрибутов. Элемент можно рассматривать как объект JSON, поскольку он имеет следующие сходные характеристики:

- его ключи уникальны;
- число атрибутов не ограничивается;
- значения могут быть разных типов, в том числе числами, строками и объектами.

Таблица – это просто хранилище модели, представленной на рис. 3.1.

Теперь нам нужно преобразовать эту модель в структуру, понятную базе данных: в таблицу базы данных. При этом нужно иметь в виду, что DynamoDB практически не имеет схемы, а это означает, что достаточно определить только первичный ключ, а все остальное можно будет добавить позже. В качестве первого шага создадим минимальную жизнеспособную таблицу для наших заказов.

Готовы?

Итак, нам нужно сохранить каждый заказ как отдельный элемент таблицы. Для хранения заказов мы используем одну таблицу DynamoDB, служащую коллекцией наших заказов. Мы будем получать заказы через API и сохранять их в таблице DynamoDB. Каждый заказ можно описать следующим набором характеристик:

- уникальный номер заказа;
- выбранная пицца;
- адрес доставки;
- состояние заказа.

Эти характеристики можно использовать как ключи таблицы. Таблица с заказами должна выглядеть, как показано в табл. 3.1.

Таблица 3.1. Структура заказов в таблице в DynamoDB

Номер заказа	Состояние заказа	Пицца	Адрес
1	ожидает	Капричоза	221Б Бейкер-стрит
2	ожидает	Наполетана	29 Акация-роуд

Следующий шаг – создание таблицы. Назовем ее `pizza-orders`. Как и в большинстве других случаев в AWS, сделать это можно несколькими способами;

мы предпочитаем интерфейс командной строки. Чтобы создать таблицу для заказов, можно выполнить команду `aws dynamicodb create-table`, как показано в листинге 3.1.

Мы должны добавить в команду несколько обязательных параметров. Во-первых, мы должны указать имя таблицы; в нашем случае это будет `pizza-orders`. Затем нужно определить атрибуты. Как уже упоминалось выше, DynamoDB требует определить только первичный ключ, поэтому мы можем указать лишь атрибут `orderId` и сообщить DynamoDB, что он будет иметь строковый тип. Мы также должны сообщить DynamoDB, что `orderId` будет вашим первичным ключом (или, выражаясь терминологией DynamoDB, *хеш-ключом*).

Далее мы должны сообщить DynamoDB, какую пропускную способность для операций чтения и записи она должна зарезервировать для вашего приложения. Поскольку мы пока еще только разрабатываем приложения, пропускной способности 1 будет более чем достаточно, а кроме того, ее легко изменить позже, воспользовавшись клиентом командной строки AWS CLI. DynamoDB поддерживает автоматическое масштабирование, но требует определить минимальную и максимальную пропускную способность. На данный момент нам не нужно автоматическое масштабирование, но если вы захотите узнать больше об этом, посетите страницу http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/AutoScaling.html.

Наконец, вам нужно выбрать регион, где будет создана таблица. Выберите тот же регион, что и для приложения с функцией Lambda, чтобы уменьшить задержку при обмене данными с таблицей. Полная команда приводится в листинге 3.1.

Листинг 3.1. Создание таблицы DynamoDB с помощью AWS CLI

```
aws dynamodb create-table --table-name pizza-orders \
  --attribute-definitions AttributeName=orderId,AttributeType=S \
  --key-schema AttributeName=orderId,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1 \
  --region eu-central-1 \
  --query TableDescription.TableArn --output text
```

Создание таблицы pizza-orders с помощью AWS CLI.

Определение схемы ключа.

Выбор региона для таблицы DynamoDB.

По результатам вернуть имя ресурса таблицы (Amazon Resource Name, ARN), чтобы убедиться, что таблица создана.

Определение пропускной способности (чтения и записи) для таблицы DynamoDB.

Определение атрибута, сообщающее базе данных DynamoDB, что первичный ключ имеет строковый тип (S).

СОВЕТ. Атрибут `--query` в команде AWS CLI отфильтрует выходные данные и вернет только те значения, которые вам нужны. Например, `--query TableDescription.TableArn` вернет лишь имя ресурса Amazon (Amazon Resource Name, ARN) таблицы.

Также можно определить тип вывода, добавив атрибут `--output` со значением. Например, `--output text` вернет результат в виде простого текста.

После запуска команда из листинга 3.1 выведет ARN таблицы DynamoDB, который выглядит примерно так:

```
arn:aws:dynamodb:eu-central-1:123456789101:table/pizza-orders
```

Вот и все! Теперь у нас есть таблица DynamoDB для хранения заказов. Давайте посмотрим, как подключить ее к обработчикам маршрутов нашего API.

Чтобы подключиться к таблице DynamoDB из Node.js, необходимо установить AWS SDK для Node.js. Для этого установите модуль `aws-sdk` с помощью NPM. Если вы не знаете, как это делается, загляните в приложение A.

Теперь у вас есть все ингредиенты, и пришло время сделать самый важный шаг: приготовить из них пиццу. (К счастью, у нас есть рецепт пиццы в приложении D.)

Самый простой способ взаимодействий с DynamoDB из приложения Node.js – использовать класс `DocumentClient`, который действует асинхронно. Как и любая часть AWS SDK, он отлично работает с Claudia, поэтому используем его в обработчиках маршрутов API, реализованных в главе 2.

DynamoDB DocumentClient

`DocumentClient` – это один из классов DynamoDB в AWS SDK. Его цель – упростить работу с элементами таблицы путем абстрагирования операций. Он имеет простой API, но мы рассмотрим только те его части, которые вам понадобятся в этой главе. Желающие ознакомиться с документацией могут найти ее по адресу: http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB/DocumentClient.html.

Подключить Pizza API к вновь созданной базе данных не составляет труда. Чтобы сохранить заказ в таблице DynamoDB, нужно выполнить два шага:

1) импортировать AWS SDK и инициализировать экземпляр `DocumentClient`;
2) дополнить обработчик метода `POST`, добавив операцию сохранения заказа.

Поскольку в главе 2 мы разбили свой код на отдельные файлы, начнем с файла `create-order.js` в папке `handlers`. В листинге 3.2 показано, как изменить `create-order.js`, чтобы добавить сохранение нового заказа в таблице `pizza-orders`.

Листинг 3.2. Сохранение заказа в таблице DynamoDB

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function createOrder(request) {
```

Импортировать AWS SDK и инициализировать DocumentClient.

```
  if (!request || !request.pizza || !request.address)
    throw new Error('To order pizza please provide pizza type and address
      where pizza should be delivered')

  return docClient.put({
    TableName: 'pizza-orders',
    Item: {
      orderId: 'some-id',
      pizza: request.pizza,
      address: request.address,
      orderStatus: 'pending'
    }
  }).promise()
    .then((res) => {
      console.log('Order is saved!', res)
      return res
    })
    .catch((saveError) => {
      console.log(`Oops, order is not saved :(`, saveError)
      throw saveError
    })
}

module.exports = createOrder
```

Сохранить новый заказ в таблице DynamoDB.

Номер заказа может быть любой строкой – пока он жестко зашит в код.

Экземпляр DocumentClient имеет метод .promise, возвращающий объект Promise.

Вывести ответ в консоль и вернуть данные, когда объект Promise завершит выполнение.

Если Promise завершился с ошибкой, вывести сообщение об ошибке и снова сгенерировать ее, чтобы дать возможность обработать в файле api.js.

Экспортировать функцию-обработчик.

По завершении этого шага метод `POST /orders` нашего Pizza API должен работать, как показано на рис. 3.2.

Рассмотрим происходящее поближе. После импорта AWS SDK мы должны инициализировать экземпляр класса `DocumentClient`. Также мы заместили возврат пустого объекта в строке 7 предыдущей версии обработчика `create-order.js` кодом, который сохраняет заказ в таблице с помощью `DocumentClient`.

Чтобы сохранить заказ в DynamoDB, мы вызвали метод `DocumentClient.put`, который помещает новый элемент в базу данных, создавая новый или заменяя существующий элемент с тем же идентификатором. Метод `put` принимает объект с атрибутом `TableName`, описывающим таблицу, и с атрибутом `Item`, представляющим элемент. Проектируя таблицу в базе данных, мы решили, что элемент должен иметь четыре атрибута: номер заказа, вид пиццы, адрес и состояние заказа. Именно эти данные добавляются в атрибут `Item` объекта, который передается методу `DocumentClient.put`.

Поскольку Claudia API Builder поддерживает асинхронные операции, мы использовали метод `.promise` для `DocumentClient.put`. Метод `.promise` преобразует ответ в JavaScript-объект `Promise`[1]. Возможно, вам интересно узнать, существу-

[1] Promise в переводе с англ. – «обещание». Возвращая объект `Promise`, класс `DocumentClient` обещает выполнить запрошенную операцию в какой-то момент в будущем. – *Прим. перев.*

ют ли какие-либо отличия в работе объектов `Promise` в бессерверных приложениях и как Claudia обрабатывает асинхронный обмен данными. В следующем разделе вы найдете краткое описание объектов `Promise` и как они работают с Claudia и Claudia API Builder. Если вы уже знакомы с этими понятиями, можете смело переходить к разделу 3.3.

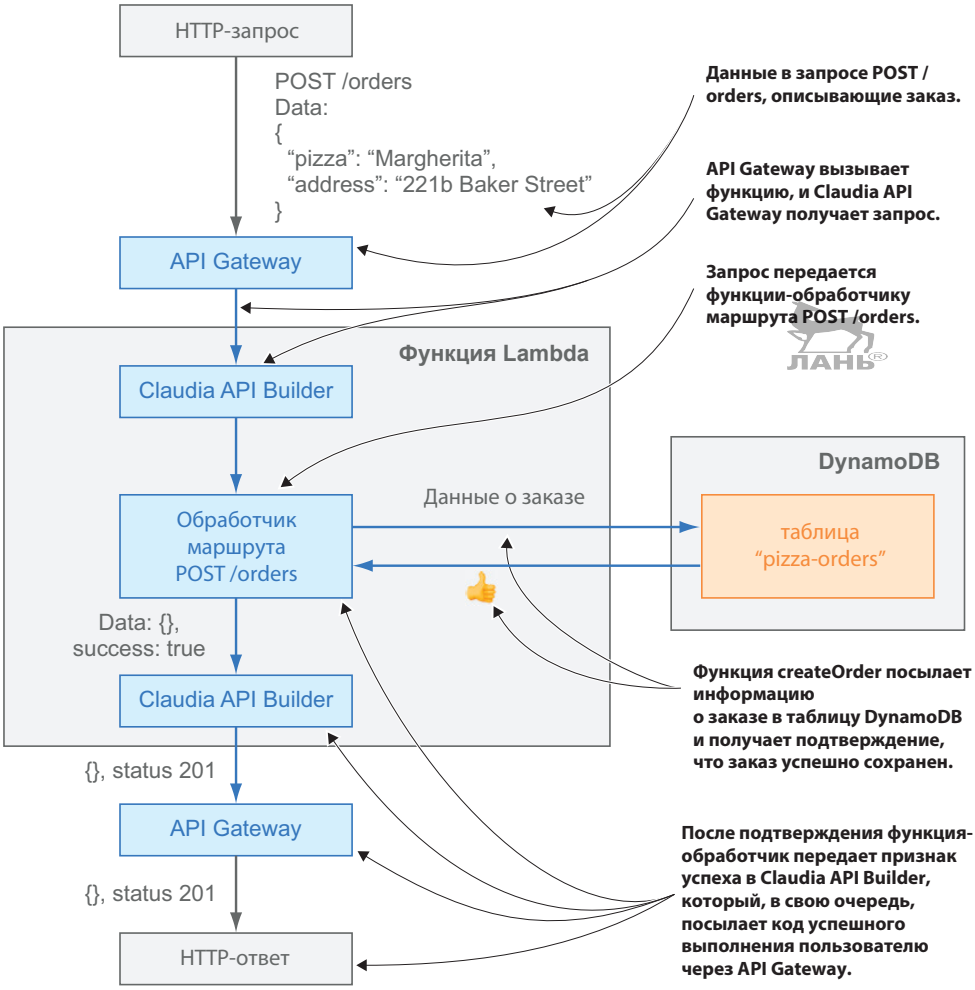


Рис. 3.2. Поток обработки запроса POST /orders в Pizza API с использованием базы данных DynamoDB

3.2. Обещание доставить меньше чем за 30 минут!

Процесс приготовления пиццы включает замешивание теста, выпекание, добавление ингредиентов на основу и т. д. Все это асинхронные операции. Если бы они были синхронными, пиццерия тетушки Марии застопорилась бы и

перестала выполнять любые другие операции до окончания текущей. Например, вам пришлось бы ждать, пока тесто поднимется, и только потом делать что-то еще. И за такую трату времени тетушка Мария уволит кого угодно, даже вас! Поскольку большинство сред выполнения JavaScript является однопоточными, многие длительные операции, такие как сетевые запросы, выполняются асинхронно. Асинхронное выполнение реализуется двумя известными способами: с помощью обратных вызовов и обещаний (объектов `Promise`). На момент написания этих строк для всех приложений Node.js рекомендовался способ с использованием объектов `Promise`. Мы не будем описывать обратные вызовы, так как вы, скорее всего, уже знакомы с ними.

Асинхронные обещания

Обещание, или объект *Promise*, представляет результат асинхронной операции, который будет получен в будущем.

Объект обещания (`Promise`) сродни обычным обещаниям, которые мы часто даем партнерам, друзьям, родителям и детям:

- «Дорогой, ты вынесешь мусор?»
- «Да, дорогая, я обещаю сделать это!»

Угадайте, кто через пару часов вынесет мусор?

По большому счету объекты `Promise` – это всего лишь удобные обертки вокруг обратных вызовов. Объекты `Promise` используются для обертывания некоторого действия или операции и, подобно обычным обещаниям, могут оказываться в двух состояниях: *выполнено* или *отвергнуто* (не выполнено).

Объекты `Promise` могут иметь связанные с ними условия, и тогда мощь асинхронных вычислений проявляется особенно ярко:

- «Джонни, когда закончишь убирать свою комнату, сможешь пойти погулять!»

Этот пример демонстрирует возможность организовать выполнение определенных действий только после успешного выполнения некоторой асинхронной операции. Аналогично, есть возможность приостановить выполнение определенных блоков кода в ожидании завершения асинхронной операции.

Листинг 3.3 реализует на JavaScript обещание, данное в предложении выше.

Листинг 3.3. Отпускаем Джонни погулять, используя подход с обещаниями

```
function tellJohhny(homework) {
  return finish(homework)
    .then(finishedHomework => {
```

finish, getOut и play – это асинхронные функции, но все они возвращают обещания, которые можно объединить в цепочку.

```
    return getOut(finishedHomework);
  })
  .then(result => {
    return play();
  })
  .catch(error => {
    console.log(error);
  });
}
```

getOut вызывается только после того, как Джонни закончит уборку в своей комнате.

Перехватывает ошибки, возникшие при выполнении любых функций в цепочке обещаний.

Объекты `Promise` обладают следующими особенностями:

- *могут объединяться в цепочки* – как показано в листинге 3.3, объекты обещаний можно объединять в цепочки и передавать результаты из одного блока кода в другой;
- *выполняются параллельно* – можно одновременно выполнить две функции и получить результаты обеих одновременно;
- *позволяют прерывать асинхронные операции* – если функция вернула ошибку или ее результат стал не нужен, вы можете отвергнуть полученный результат или в любой момент прервать выполнение функции. В отличие от подхода на основе обратных вызовов, прерывание работы объекта `Promise` останавливает выполнение всей цепочки обещаний;
- *восстановление после ошибки* – используя блок `catch`, легко можно перехватить и обработать возникшую ошибку.

Некоторые клиенты заказывают несколько пицц в одном заказе, и все они доставляются вместе, а не по одной. Если бы доставка осуществлялась иначе, клиенты были бы крайне недовольны такой неэффективностью. Поэтому повар обычно печет все пиццы одновременно, а доставщик ждет, пока все они приготовятся.

Этот процесс реализует код в листинге 3.4.

Листинг 3.4. Одновременное выпекание нескольких пицц

```
function preparePizza(pizzaName) {
  return new Promise((resolve, reject) => {
    // Выпекание пиццы
    resolve(bakedPizza);
  });
}

function processOrder(pizzas) {
  return Promise.all([
    preparePizza('extra-cheese'),
```

```
    preparePizza('anchovies')
  ]);
}

return processOrder(pizzas)
  .then((readyPizzas) => {
    // Вывести результат приготовления пиццы с двойным сыром
    console.log(readyPizzas[0]);
    // Вывести результат приготовления пиццы с анчоусами
    console.log(readyPizzas[1]);
    return readyPizzas;
})
```

Как демонстрируют листинги 3.3 и 3.4, объекты обещаний `Promise` здорово помогают организовать процесс. Они позволяют справиться с любой ситуацией в пиццерии тетушки Марии и помогают правильно описать все процессы. Библиотека Claudia поддерживает все возможности `Promise`, так что вы без труда сможете использовать их. В листинге 3.5 показан простой пример, когда обработчик задерживает ответ на одну секунду. Поскольку функция `setTimeout` не возвращает экземпляр `Promise`, ее вызов нужно обернуть инструкцией `Promise()`.

Листинг 3.5. Обертывание асинхронной операции, не возвращающей экземпляр объекта обещания Promise

```
const Api = require('claudia-api-builder')
const api = new Api()

api.get('/', request => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Hello after 1 second')
    }, 1000)
  })
})

module.exports = api
```

Заключить асинхронную операцию в JavaScript-объект Promise.

Вызвать setTimeout для односекундной задержки.

Использовать метод resolve для отправки ответа в Claudia API Builder.

Как видите, в отличие от некоторых популярных фреймворков для Node.js, Claudia API Builder передает запрос в обработчик маршрута. В главе 2, чтобы отправить ответ, мы просто возвращали значение, но в случае асинхронной операции мы должны вернуть обещание – экземпляр `Promise`. Получив его, Claudia API Builder дождется выполнения операции и вернет полученное значение в ответе.

ПРИМЕЧАНИЕ. AWS SDK имеет встроенную поддержку JavaScript-объектов `Promise`. Все классы SDK имеют метод `promise`, возвращающий обещание.

3.3. Опробование API

После краткого знакомства с обещаниями снова выполните команду `claudia update` в папке `pizza-api` и разверните код. Менее чем через минуту вы сможете протестировать API и проверить его работу.

Для тестирования снова используем команду `curl` из главы 2:

```
curl -i \
  -H "Content-Type: application/json" \
  -X POST \
  -d '{"pizza":4,"address":"221b Baker Street"}'
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_1/orders
```

ПРИМЕЧАНИЕ. Не забудьте заменить URL в команде своим URL, который вернула команда `claudia update`.

Вот так так ! Команда `curl` вернула:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 219
Date: Mon, 25 Sep 2017 06:53:36 GMT

{"errorMessage":"User: arn:aws:sts::012345678910:assumed-role/pizza-api
     -executor/book-pizza-api
is not authorized to perform: dynamodb:PutItem on resource:
arn:aws:dynamodb:eu-central-1:012345678910:table/pizza-orders"}
```

В чем ошибка?

Команда сообщила, что роль, которой наделена наша функция Lambda (`arn:aws:sts::012345678910:assumed-role/pizza-api-executor/book-pizza-api`), не допускает выполнения команды `dynamodb:PutItem` в базе данных DynamoDB (`arn:aws:dynamodb:eu-central-1:012345678910:table/pizza-orders`).

Чтобы устранить проблему, нужно добавить IAM-политику, которая позволит нашей функции Lambda взаимодействовать с базой данных. Сделать это можно с помощью команды `claudia create` с ключом `--policies`. Но будьте внимательны: команда `claudia update` не поддерживает этот ключ – библиотека Claudia не дублирует возможности одной команды в другой.

ПРИМЕЧАНИЕ. В AWS все ограничивается политиками IAM, которые напоминают политики авторизации. Политика IAM аналогична визе в паспорте. Чтобы въехать в определенную страну, необходимо иметь действующую визу.

Сначала определим роль в файле JSON. Создайте новую папку `roles` в корневом каталоге проекта. Затем создайте файл роли для DynamoDB. Сохраните его с именем `dynamicodb.json` и добавьте в него код из листинга 3.6. Вам нужно разрешить вашей функции Lambda читать, удалять и помещать элементы в таблице. Поскольку в будущем у вас может появиться больше таблиц, применим это правило ко всем таблицам, а не только к той, которая есть сейчас.

Листинг 3.6. Файл JSON с описанием роли DynamoDB

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:Scan",
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

Определение версии.

Определение инструкции для этой роли.

Определение конкретных действий, допускаемых или отвергаемых этой ролью.

Разрешить ("Allow") описанные действия.

Это правило применяет роль ко всем таблицам в DynamoDB.

СОВЕТ. В промышленном приложении вам могут понадобиться более точные роли, запрещающие доступ функциям Lambda ко всем таблицам в DynamoDB. Узнать больше о ролях и политиках можно на странице http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.

Теперь можно использовать команду `put-role-policy`, чтобы добавить политику к своей роли, как показано в листинге 3.7. Для этого нужно указать роль, которую использует функция Lambda, имя политики и абсолютный путь к файлу `dynamodb.json`. Где сохраняется роль? Помните файл `claudia.json`, созданный библиотекой Claudia в корневой папке проекта? Откройте этот файл, и вы увидите атрибут `role` в разделе `lambda`.

Листинг 3.7. Добавление политики для роли Lambda, позволяющей взаимодействовать с таблицами в DynamoDB

```
aws iam put-role-policy \
  --role-name pizza-api-executor \
```

Команда put-role-policy добавляет политику.

Роль Lambda из файла claudia.json, для которой определяется политика.

```
  --policy-name PizzaApiDynamoDB \
  --policy-document file://./roles/dynamodb.json
```

Имя политики.

Использовать файл dynamodb.json как источник для создания политики.

ПРИМЕЧАНИЕ. Путь к `dynamicodb.json` должен указываться с префиксом `file://`. Если вы решите указать абсолютный путь, то должны после `file:` добавить три слеша. Первые два относятся к определению протокола `file://`, а третий обозначает начало абсолютного пути.

Команда из листинга 3.7 ничего не выводит. Это нормально, потому что отсутствие ответа означает, что все прошло благополучно.

Теперь снова выполните ту же команду `curl` и попробуйте добавить заказ:

```
curl -i \
  -H "Content-Type: application/json" \
  -X POST \
  -d '{"pizza":4,"address":"221b Baker Street"}'
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_1/orders
```

ПРИМЕЧАНИЕ. На этот раз не нужно повторно развертывать код, потому что он не изменился. Мы лишь изменили роль для функции Lambda.

Команда `curl` должна вернуть `{}` с кодом 201. Если у вас это получилось, поздравляем! Ваша база данных подключена и работает! Но как узнать, что заказ действительно сохранился в таблице?

У AWS CLI есть ответ на этот вопрос. Чтобы вывести список всех элементов в вашей таблице, выполните команду `scan`. Команда `scan` вернет все элементы, хранящиеся в таблице, если вызвать ее без фильтра. Чтобы вывести список всех элементов в таблице, выполните команду из листинга 3.8.

Листинг 3.8. Команда для вывода списка всех элементов из таблицы `pizza-orders`

```
aws dynamodb scan \
  --table-name pizza-orders \
  --region eu-central-1 \
  --output json
```

Команда scan выводит список всех элементов в таблице.

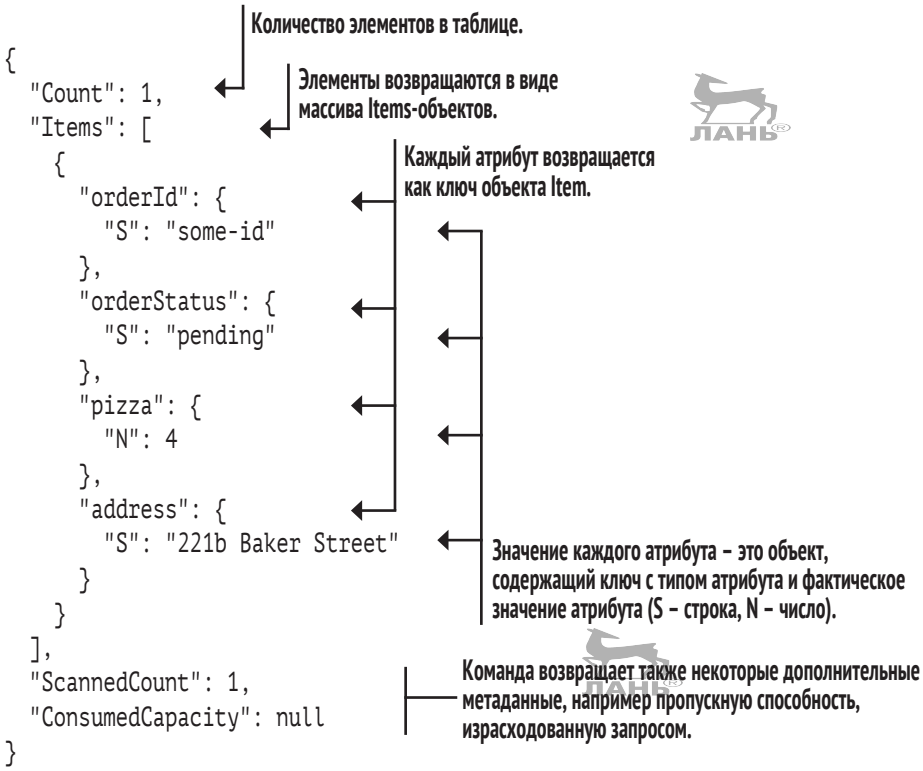
Команде нужно передать параметр с именем таблицы.

Также можно определить формат вывода результатов.

Эта команда «просканирует» таблицу `pizza-orders` и вернет результат в виде объекта JSON. Есть возможность изменить формат вывода, передав параметр `text`. В этом случае вы получите результат в виде обычного текста. Также поддерживаются и другие форматы вывода, включая XML.

Команда должна вернуть результат, как показано в листинге 3.9: ответ в формате JSON с количеством записей и массивом всех элементов из таблицы.

Листинг 3.9. Результат применения команды `scan` к таблице `pizza-orders`




```
{
  "Count": 1,
  "Items": [
    {
      "orderId": {
        "S": "some-id"
      },
      "orderStatus": {
        "S": "pending"
      },
      "pizza": {
        "N": 4
      },
      "address": {
        "S": "221b Baker Street"
      }
    }
  ],
  "ScannedCount": 1,
  "ConsumedCapacity": null
}
```

Отлично! Похоже, что наш API работает так, как ожидалось!

Попробуйте добавить еще один заказ с помощью той же команды `curl`, например пиццу `Napoletana` с адресом доставки `29 Acacia Road`. Если потом снова запустить команду из листинга 3.8, вы увидите только один элемент в вашей таблице – предыдущий таинственным образом исчез!

Почему так получилось?

Помните, что мы жестко зашили номер заказа `orderId` в код обработчика `create-order.js` (см. листинг 3.2)?

Каждый заказ должен иметь уникальный первичный ключ, а мы использовали один и тот же номер, поэтому новый заказ заменил старый.

Чтобы решить эту проблему, нужно установить модуль `uuid` из NPM и сохранить его как зависимость. `uuid` – это простой модуль, генерирующий универсально-уникальные идентификаторы (universally unique identifiers).

После установки модуля измените обработчик `create-order.js`, как показано в листинге 3.10. Чтобы получить уникальный идентификатор, достаточно импортировать и вызвать функцию `uuid`. Имейте в виду, в листинге 3.10 показана только часть обработчика `create-order.js`, затронутая изменениями; остальной код остался без изменений (см. листинг 3.2).

Универсально-уникальные идентификаторы

Универсально-уникальный идентификатор – это 128-битное значение, используемое для идентификации информации в компьютерных системах. Он более широко известен под аббревиатурой UUID. Иногда его называют *глобально-уникальным идентификатором* (Globally Unique IDentifier, GUID).

Универсально-уникальные идентификаторы стандартизованы фондом свободного программного обеспечения Open Software Foundation (OSF) как часть среды распределённых вычислений (Distributed Computing Environment, DCE). Узнать больше о стандарте UUID можно в документе RFC 4122 (описывающем этот стандарт), доступном по адресу: http://www.ietf.org/rfc/rfc4122.txt.

Листинг 3.10. Добавление универсально-уникального идентификатора UUID в заказ при его создании

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const uuid = require('uuid')

function createOrder(request) {
  return docClient.put({
    TableName: 'pizza-orders',
    Item: {
      orderId: uuid(),
      pizza: request.pizza,
      address: request.address,
      status: 'pending'
    }
  }).promise()

// Остальной код не изменился
```

Импортировать модуль uuid, установленный с помощью NPM.

Вызвать функцию uuid, чтобы получить уникальный идентификатор для заказа.

Остальной код в файле остался прежним, как в листинге 3.2.

Повторно разверните код командой `claudia update`, выполните ту же команду `curl`, что использовалась для тестирования API, и затем снова вызовите команду `scan` из листинга 3.8. Вы увидите, что появился новый заказ с уникальным идентификатором, который выглядит примерно так: `8c499027-a2d7-4ad9-8360-a49355021adc`. Попробовав добавить еще заказы, вы увидите, что теперь все они сохраняются в таблице.

3.4. Извлечение заказов из базы данных

Извлечь заказ, сохраненный в таблице, намного проще. Для этого можно использовать метод `scan` класса `DocumentClient`.

Метод `scan` действует подобно одноименной команде в AWS CLI, отличаясь лишь тем, что требует передать параметр с объектом, содержащим атрибуты, которые описывают операцию сканирования. Единственным обязательным атрибутом в этом объекте является атрибут с именем таблицы.

Обработчик `get-orders.js` может не только сканировать таблицу, но и извлекать элементы по их идентификаторам. То же самое можно сделать, отфильтровав результаты сканирования, но это неэффективное решение. Более эффективный подход заключается в использовании метода `get`, который действует почти так же, но дополнительно требует указать ключ извлекаемого элемента.

Изменим наш файл `get-orders.js` в папке `handlers`, добавив в него сканирование таблицы с заказами или извлечение единственного элемента, если был указан его ключ. Код должен выглядеть, как показано в листинге 3.11. После внесения этих изменений снова разверните функцию Lambda командой `claudia update`.

Листинг 3.11. Обработчик `get-orders.js` читает данные из таблицы `pizza-orders`

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function getOrders(orderId) {
  if (typeof orderId === 'undefined')
    return docClient.scan({
      TableName: 'pizza-orders'
    }).promise()
      .then(result => result.Items)

  return docClient.get({
    TableName: 'pizza-orders',
      Key: {
        orderId: orderId
      }
    }).promise()
      .then(result => result.Item)
}

module.exports = getOrders
```

Импортировать и инициализировать DocumentClient.

Сканировать таблицу pizza-orders.

Метаданные нам не интересны, поэтому вернуть только элементы.

Если указан идентификатор заказа, использовать метод get, чтобы получить только требуемый элемент.

Метод get требует указать первичный ключ – в данном случае orderId.

И снова, метаданные нам не интересны, поэтому возвращаем только элемент.

Давайте проверим его! Сначала извлечем все заказы следующей командой `curl`:

```
curl -i \
  -H "Content-Type: application/json" \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_2/orders
```

Она должна вывести примерно такой список:

```
HTTP/1.1 200 OK
[{
  "address": "29 Acacia Road",
  "orderId": "629d4ab3-f25e-4110-8b76-aa6d458b1fce",
  "pizza": 4,
  "orderStatus":"pending"
}, {
  "address": "29 Acacia Road",
  "orderId": "some-id",
  "pizza": 4,
  "status": "pending"
}]
```

Не волнуйтесь, если увидите у себя другие идентификаторы заказов; так и должно быть, потому что они *уникальные*.

Теперь попробуйте передать один из идентификаторов, чтобы получить только один заказ, выполнив следующую команду `curl`:

```
curl -i \
  -H "Content-Type: application/json" \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_2/
     orders/629d4ab3-f25e-4110-8b76-aa6d458b1fce
```

Результат должен выглядеть примерно так:

```
HTTP/1.1 200 OK
{
  "address": "29 Acacia Road",
  "orderId": "629d4ab3-f25e-4110-8b76-aa6d458b1fce",
  "pizza": 4,
  "status": "pending"
}
```

Все работает! Это было легко и просто, верно?

3.5. Опробование!

Как вы уже видели, сохранение заказов в базе данных и их извлечение реализуется довольно просто. Но тетушка Мария сказала нам, что иногда клиенты совершают ошибки и заказывают не ту пиццу, поэтому ей нужна возможность изменить или отменить заказ пиццы.

3.5.1. Упражнение

Чтобы выполнить просьбу тетушки Марии, подключите к базе данных еще две конечные точки:

1) добавьте в обработчик `update-order.js` возможность изменения существующего заказа в таблице `pizza-orders`;
2) добавьте в обработчик `delete-order.js` возможность удаления заказа из таблицы `pizza-orders`.

После выполнения упражнения структура файлов в Pizza API должна выглядеть, как показано на рис. 3.3.

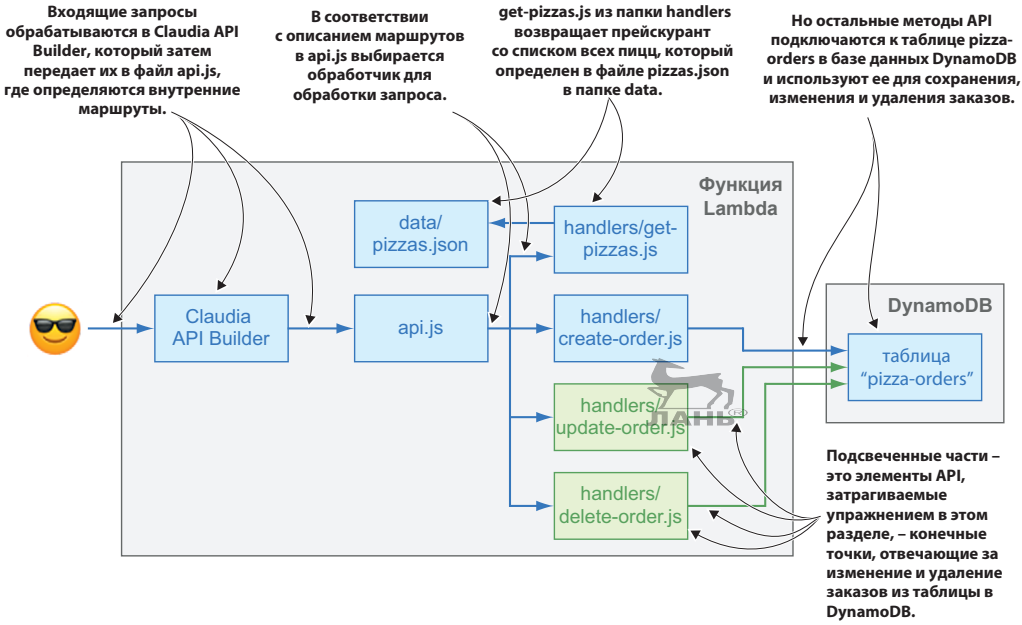


Рис. 3.3. Структура файлов и каталогов проекта Pizza API после подключения всех конечных точек к таблице DynamoDB. Элементы, затронутые изменениями, выделены другим цветом

Код решения приводится в следующем разделе. Но, прежде чем перейти к нему, попробуйте выполнить упражнение самостоятельно, но если возникнут затруднения, можете подглядеть одним глазком.

Несколько подсказок:

- для изменения и удаления заказов используйте класс `DocumentClient`;
- для изменения существующего заказа используйте метод `DocumentClient.update`. Кроме `TableName`, вы должны будете передать в вызов этого метода дополнительные параметры: `Key`, `UpdateExpression` и др. За более полной

информацией обращайтесь к официальной документации для Claudia API Builder: http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB/DocumentClient.html#update-property;

- если метод `update` покажется вам слишком сложным, вспомните, что метод `DocumentClient.put` заменяет существующий заказ новым, если их первичные ключи совпадают, поэтому можете попробовать использовать его;
- для удаления существующего заказа используйте метод `DocumentClient.delete`. Вы должны передать в этот метод объект с атрибутами `TableName` и `Key`. За более полной информацией обращайтесь к официальной документации: http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB/DocumentClient.html#delete-property;
- не забудьте вернуть `Promise` и передать значение.

Если упражнение покажется вам слишком простым и вы захотите усложнить его, попробуйте также реализовать:

- измените обработчики `update-order.js` и `delete-order.js`, чтобы можно было изменять или удалять только невыполненные заказы, потому что клиенты не должны иметь возможности изменять заказы, которые уже были доставлены;
- измените обработчик `get-orders.js`, добавив фильтрацию по состоянию заказа. По умолчанию он должен возвращать только заказы, находящиеся в состоянии ожидания выполнения.

Решение этих дополнительных заданий вместе с комментариями можно найти в исходном коде примера.

3.5.2. Решение

Вы уже закончили? Или решили подглядеть? Если закончили, отлично! Но даже если вы не справились с упражнением, ничего страшного. База данных DynamoDB немного отличается от других популярных баз данных noSQL, и вам может понадобиться время, чтобы освоить ее.

А теперь рассмотрим решение. В листинге 3.12 показаны изменения в файле `update-order.js`.

Листинг 3.12. Изменение заказа в таблице `pizza-orders`

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function updateOrder(orderId, options) {
  if (!options || !options.pizza || !options.address)
```

Импортировать и инициализировать DocumentClient.

```
    throw new Error('Both pizza and address are required to update an order')

  return docClient.update({
    TableName: 'pizza-orders',
    Key: {
      orderId: orderId
    },
    UpdateExpression: 'set pizza = :p, address=:a',
    ExpressionAttributeValues: {
      ':p': options.pizza,
      ':a': options.address
    },
    ReturnValues: 'ALL_NEW'
  }).promise()
    .then((result) => {
      console.log('Order is updated!', result)
      return result.Attributes
    })
    .catch((updateError) => {
      console.log(`Oops, order is not updated :(`, updateError)
      throw updateError
    })
}

module.exports = updateOrder
```

Передать идентификатор и объект с атрибутами для изменения заказа.

Определение ключа заказа.

Описание изменения атрибутов заказа.

Передать значения в выражение UpdateExpression.

Сообщить DynamoDB, что требуется вернуть новый элемент.

Просто вывести ответ или ошибку и передать значение – мы используем его в главе 5 для отладки.

Экспортировать обработчик.

Этот код несильно отличается от кода в обработчике `create-order.js`. Вот два основных отличия:

- вызывается метод `DocumentClient.update` с параметром `Key`, представляющим идентификатор заказа `orderId`;
- в метод передается больше параметров – `orderId` и новые значения для атрибутов (`pizza` и `address`).

> **СОВЕТ.** Синтаксис метода `update` может показаться немного запутанным из-за атрибутов `UpdateExpression`, `ExpressionAttributeValues` и `ReturnValues`. Но эти атрибуты довольно просты. Комментарии в листинге 3.12 объясняют их. За более полной информацией обращайтесь к официальной документации: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.UpdateExpressions.html.

В листинге 3.13 показаны изменения в файле `delete-order.js`. Они напоминают изменения в файлах `create-order.js` и `update-order.js`; единственное отличие – здесь используется метод `DocumentClient.delete`.

Листинг 3.13. Удаление заказа из таблицы `pizza-orders`

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function deleteOrder(orderId) {
  return docClient.delete({
    TableName: 'pizza-orders',
    Key: {
      orderId: orderId
    }
  }).promise()
    .then((result) => {
      console.log('Order is deleted!', result)
      return result
    })
    .catch((deleteError) => {
      console.log(`Oops, order is not deleted :(`, deleteError)
      throw deleteError
    })
}

module.exports = deleteOrder
```

Импортировать и инициализировать DocumentClient.

Передать идентификатор заказа.

Использовать метод DocumentClient.delete для удаления заказа.

Передать orderId, первичный ключ в таблице.

Не забудьте вызвать метод .promise, чтобы вернуть экземпляр Promise.

Вывести ответ или ошибку и передать значение.

Экспортировать обработчик.

Это было совсем несложно, верно?

Теперь снова запустите команду `claudia update` из папки `pizza-api`, чтобы повторно развернуть код. Чтобы убедиться, что все работает правильно, используйте те же команды `curl`, которые применялись в главе 2. Скопируйте их из листингов 3.14 и 3.15 и вставьте в окно терминала. Не забудьте заменить значение `orderId`. Без этого команды не будут работать, потому что это просто заполнитель.

Листинг 3.14. Команда `curl` для тестирования маршрута `PUT /orders/{orderId}`

```
curl -i \
  -H "Content-Type: application/json" \
  -X PUT \
  -d '{"pizza": 3, "address": "221b Baker Street"}'
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_3/
     orders/some-id
```

Не забудьте заменить some-id действительным идентификатором заказа.

Эта команда должна вернуть:

```
HTTP/1.1 200 OK
{
```

```
  "address": "221b Baker Street",
  "orderId": "some-id",
  "pizza": 3
  "status": "pending"
}
```

Листинг 3.15. Команда `curl` для тестирования маршрута `DELETE /orders/{orderId}`

```
curl -i \
  -H "Content-Type: application/json" \
  -X DELETE \
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter3_3/
    orders/some-id
```

Не забудьте заменить some-id действительным идентификатором заказа.

Эта команда должна вернуть:

```
HTTP/1.1 200 OK
{}
```

В заключение

- Для создания полезного бессерверного приложения часто требуется использовать сторонние службы, например для хранения информации в базе данных или для получения данных из других API.
- Взаимодействия с внешними службами выполняются асинхронно.
- Claudia позволяет вызывать асинхронные функции с помощью JavaScript-объектов `Promise`.
- JavaScript-объекты `Promise` упрощают выполнение асинхронных операций. Они также позволяют решить проблему, известную под названием «ад обратных вызовов», за счет возможности объединять асинхронные операции в цепочки, передавать значения между ними и возвращать ошибки.
- Простейший способ организовать хранение данных в функциях AWS Lambda – использовать DynamoDB, базу данных NoSQL, которая является частью экосистемы AWS.
- Для использования DynamoDB из Node.js требуется установить модуль `aws-sdk`. Кроме того, AWS SDK предлагает также класс `DocumentClient`, помогающий сохранять, извлекать, изменять и удалять элементы в таблицах DynamoDB.
- Таблицы DynamoDB напоминают коллекции в традиционных базах данных NoSQL. К сожалению, DynamoDB поддерживает только запросы по первичному ключу, который может быть комбинацией хеш-ключа и диапазона.

Глава 4

Доставка пиццы: подключение к внешней службе

Эта глава охватывает следующие темы:

- подключение бессерверной функции к внешней службе через HTTP API;
- решение распространенных проблем асинхронных взаимодействий с помощью Claudia API Builder.

Как мы уже говорили в предыдущей главе, асинхронные операции в AWS Lambda выполняются с помощью Claudia API Builder. Там же мы рассказали, как создать базу данных для заказов на пиццу, и написали функции для их хранения, поиска, обновления и удаления. Но наше приложение способно на гораздо большее.

В этой главе мы покажем, как подключить бессерверное приложение к внешней службе HTTP, добавив в него возможность использовать API компании Some Like It Hot и предлагать больше услуг по доставке на дом. Вы узнаете, как сформировать HTTP-запрос в AWS Lambda, как обрабатывать ответы с сообщениями об ошибках и как определить точку входа с помощью Claudia API Builder. Вы также познакомитесь с самыми распространенными проблемами и подводными камнями асинхронных операций и узнаете, как их решать, а также как избежать их появления.

4.1. Подключение к внешней службе

Дзынь-дзынь! Снова звонок от тетушки Марии. Она довольна и благодарит вас за проделанную работу, но в ее голосе чувствуется легкое беспокойство. После секундной паузы она просит об одолжении.

Просьба касается организации доставки. Каждый раз, когда пиццерия выполняет заказ и хочет доставить его клиенту, сотруднику приходится звонить

в компанию Some Like It Hot, осуществляющую доставку. Это не было проблемой, пока количество заказов на пиццу не начало расти с недавнего времени (спасибо вам!). Но теперь это отнимает слишком много времени, поэтому тетушка Мария хочет, чтобы вы нашли альтернативное решение. К счастью, компания Some Like It Hot имеет свою веб-службу. Можно ли подключиться к ней?

Как мы уже говорили, бессерверное приложение может подключиться к:

- базе данных (DynamoDB, Amazon RDS);
- другой функции Lambda;
- другой службе AWS (SQS, S3 и многим другим);
- внешнему API.

Веб-служба Like It Hot как раз принадлежит к последней категории.

Подключение бессерверных приложений

- *Подключение к базе данных*. Как упоминалось в предыдущей главе, некоторым приложениям необходима более структурированная база данных и DynamoDB не подходит для их задач. AWS Lambda предлагает множество других возможностей, и вы можете подключиться практически к любой другой базе данных, включая MySQL или PostgreSQL, воспользовавшись службой Amazon Relational Database Service (RDS).

  Amazon RDS – это веб-служба, упрощающая настройку, использование и масштабирование реляционной базы данных в облаке. Она обеспечивает экономически эффективную модель размещения стандартной реляционной базы данных и управляет общими задачами администрирования. Узнать больше о RDS можно на странице: https://aws.amazon.com/rds/.

- *Подключение к функции Lambda*. Иногда требуется подключить функцию Lambda к другой функции Lambda или вызвать саму себя. Это можно сделать с помощью механизма асинхронных вызовов в AWS SDK. Данный метод имеет широкое применение, например Claudia Bot Builder использует его для доставки отложенных сообщений Slack. Подробнее о Claudia Bot Builder мы поговорим во второй части этой книги.

- *Подключение к другой службе AWS*. AWS предлагает широкий спектр различных услуг, включая услугу простой очереди Simple Queue Service (SQS), услугу простого хранилища Simple Storage Service (S3) и многие другие. Подключения к другим службам AWS (например, SQS или S3) часто используются на практике, но точно так же можно подключаться к сторонним службам, используя AWS SDK. Некоторые из этих служб описаны в последующих главах данной книги.

Все перечисленные варианты подключения поддерживаются библиотекой Claudia и описаны в этой книге. Первый мы рассмотрели в предыдущей главе, а последний рассмотрим в этой. Главы с 8 по 10 (описывающие создание чат-ботов) будут посвящены подключению к функциям Lambda.

4.2. Подключение к API компании доставки

Начнем с обработчика `createOrder` в файле `create-order.js`, находящемся в папке `handlers` внутри проекта. После того как обработчик `createOrder` сохранит заказ в базе данных, мы должны подключиться к API компании Some Like It Hot, чтобы запланировать доставку. Алгоритм работы приложения в этой ситуации показан на рис. 4.1.

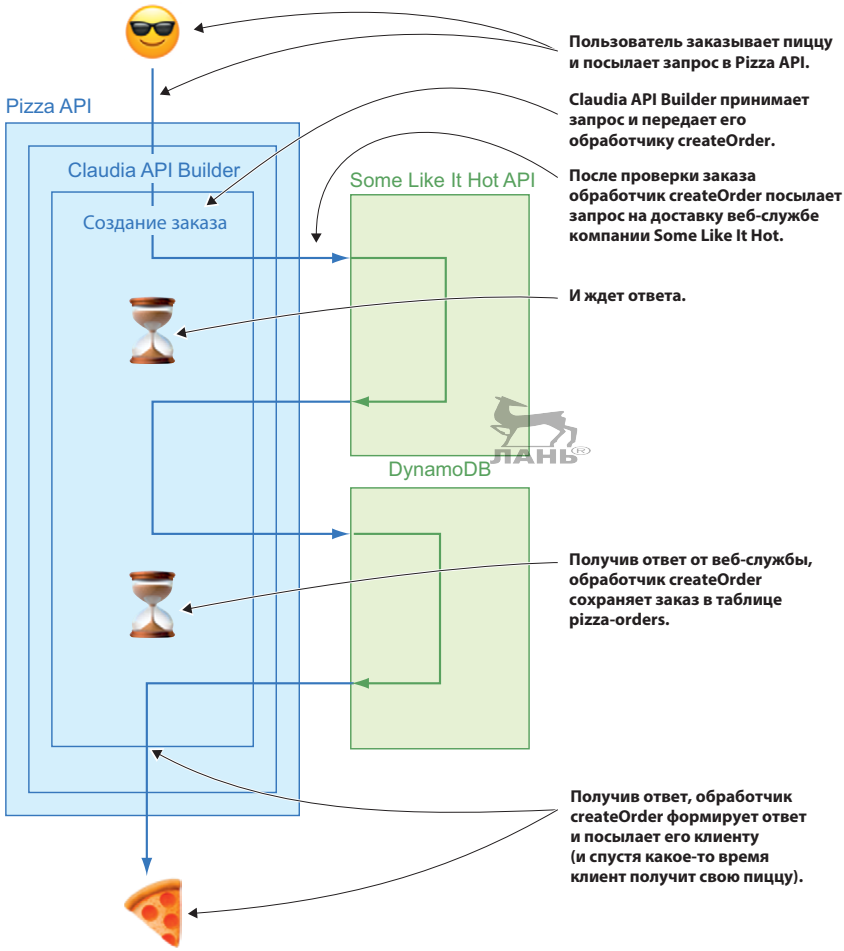


Рис. 4.1. Подключение обработчика createOrder к веб-службе компании доставки Some Like It Hot

Прежде чем приступить к подключению, рассмотрим API компании доставки Some Like It Hot, описанный в следующем разделе.

4.2.1. API компании доставки Some Like It Hot

Тетушка Мария довольна услугами компании доставки Some Like It Hot. По разумной цене они забирают и доставляют пиццы, пока они еще горячие.

И у них хорошо отлажена работа отдела приема заказов по телефону; сотрудники вежливы и быстро принимают заказы. Но это все еще узкое место – у них не так много сотрудников, сидящих на телефоне, и, несмотря на высокую скорость их работы, иногда приходится ждать какое-то время, пока сотрудник освободится и сможет ответить на ваш звонок, что является проблемой, когда нужно организовать доставку большого числа заказов каждый день.

Давайте заглянем на их веб-сайт и посмотрим, есть ли простой способ отправить им запрос. Даже простенькая веб-форма много лучше телефонного звонка. И тут нас поджидает приятный сюрприз! Компания доставки не только предлагает хорошее решение, но и полноценный API со следующими конечными точками:

- `POST /delivery` создает новую заявку на доставку и возвращает идентификатор заявки и примерное время ее выполнения;
- `GET /delivery` возвращает список запланированных к выполнению заявок, поданных вашим рестораном;
- `GET /delivery/{id}` возвращает код состояния указанной заявки;
- `DELETE /delivery/{id}` отменяет заявку, но только в течение первых 10 минут после ее создания.

Это не самый лучший API, но он достаточно хорош, чтобы мы могли автоматизировать процесс.

> **API КОМПАНИИ SOME LIKE IT HOT НЕ НАСТОЯЩИЙ API.** Имейте в виду, что API компании Some Like It Hot на самом деле… не настоящий. Мы создали фиктивный API с использованием Claudia и AWS Lambda, чтобы вы могли подключить свое тестовое приложение. Как вы увидите далее, он возвращает фиктивные данные о времени и расстоянии, не связанных с введенным вами адресом.
>
> Исходный код API открыт, и вы сможете увидеть его и документацию к нему на сайте https://github.com/effortless-serverless/some-like-it-hot-delivery.
>
> Вы можете без опаски использовать его – на самом деле он не исполняет никаких заявок и создан только ради тестирования ваших приложений!

Мы не будем углубляться в описание Some Like It Hot API, но познакомим вас с наиболее важными особенностями каждой конечной точки этого API по мере подключения к ним.

4.2.2. Создание первой заявки на доставку

Как уже рассказывала тетушка Мария, чтобы организовать доставку заказа, она обычно звонит по телефону и оставляет заявку на доставку. Наша задача – автоматизировать этот процесс. Потратьте несколько секунд и попробуйте нарисовать диаграмму алгоритма.

Когда клиент заказывает пиццу, вы должны:

1) проверить заказ;
2) подключиться к Some Like It Hot API и узнать, когда компания Some Like It Hot сможет доставить заказ;
3) сохранить заказ в базе данных.

ПРИМЕЧАНИЕ. Имейте в виду, что мы создаем минимально пригодный к работе продукт, поэтому не будем усложнять логику приложения. В реальных приложениях эта логика должна учитывать время приготовления пиццы, часы работы и, может быть, что-то еще.

Весь процесс изображен на рис. 4.2.

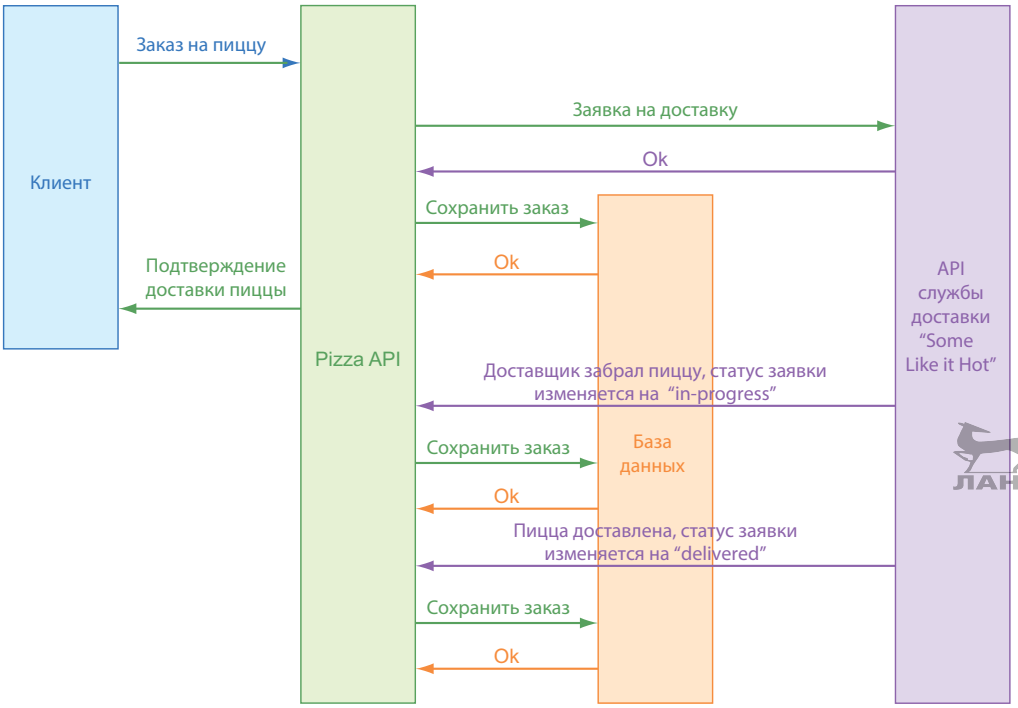


Рис. 4.2. Диаграмма, иллюстрирующая порядок взаимодействий обработчика createOrder с Some Like It Hot API и базой данных

Перед реализацией этого процесса нам нужно чуть больше узнать о создании заявки на доставку через Some Like It Hot API. Сделаем это прямо сейчас.

Наиболее важным в Some Like It Hot API является маршрут `POST /delivery`, который создает заявку. Эта конечная точка принимает следующие параметры:

- `pickupAddress` – адрес, откуда нужно забрать заказ. По умолчанию будет использоваться адрес из вашей учетной записи;
- `deliveryAddress` – адрес доставки заказа;

- `pickupTime` – время, когда можно забрать заказ. Если время не указано, заказ заберут при первой же возможности;
- `webhookUrl` – адрес URL точки входа, куда следует отправить уведомление об изменении состояния заявки.

Some Like It Hot API возвращает идентификатор заявки, время получения заказа и начальный статус заявки – «pending» («ожидает»). Когда доставщик заберет заказ из ресторана, Some Like It Hot API пошлет запрос `POST` вашей конечной точке с идентификатором и новым статусом заявки «in-progress» («выполняется»).

Точки входа

Точка входа – это просто конечная точка в вашем API. Проще говоря, это обратный вызов HTTP: HTTP-запрос `POST`, отправляемый вам, когда происходит какое-то событие. Считайте этот HTTP-запрос `POST` уведомлением о некотором событии. Веб-приложение, использующее точки входа, будет отправлять запросы `POST` на указанный URL-адрес при появлении определенных событий.

Настал момент обновить обработчик `create-order.js`. Он должен послать `POST`-запрос в Some Like It Hot API, дождаться ответа и сохранить заказ на пиццу в базе данных. Вам также нужно добавить идентификатор заявки на доставку в базу данных, чтобы получить возможность обновлять статус заказа, когда точка входа получит данные.

Дополненная версия обработчика `create-order.js` показана в листинге 4.1.

Листинг 4.1. Дополненная версия обработчика create-order.js, создающая заявку на доставку перед сохранением заказа в базе данных

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const rp = require('minimal-request-promise')

module.exports = function createOrder(request) {
  if (!request || !request.pizza || !request.address)
    throw new Error('To order pizza please provide pizza type and address
      where pizza should be delivered')

  return rp.post('https://some-like-it-hot.effortless-serverless.com/
    delivery', {
```

Послать запрос POST в Some Like It Hot API.

```
      headers: {
        "Authorization": "aunt-marias-pizzeria-1234567890",
        "Content-type": "application/json"
      },
      body: JSON.stringify({
        pickupTime: '15.34pm',
        pickupAddress: 'Aunt Maria Pizzeria',
        deliveryAddress: request.address,
        webhookUrl: 'https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/
chapter4_1/delivery',
      })
    })
    .then(rawResponse => JSON.parse(rawResponse.body))
    .then(response => {
      return docClient.put({
        TableName: 'pizza-orders',
        Item: {
          orderId: response.deliveryId,
          pizza: request.pizza,
          address: request.address,
          orderStatus: 'pending'
        }
      }).promise()
    })
    .then(res => {
      console.log('Order is saved!', res)
      return res
    })
    .catch(saveError => {
      console.log(`Oops, order is not saved :(`, saveError)
      throw saveError
    })
}
```

Добавить заголовки в запрос, включая заголовок Authorization с токеном авторизации.

Тело запроса нужно преобразовать в строку, потому что модуль minimal-request-promise ожидает получить строку.

Послать в теле параметры pickupTime, pickupAddress и deliveryAddress.

Послать URL точки входа в запросе.

Выполнить парсинг тела ответа, которое является строкой, – обратите внимание, что здесь объекты Promise объединяются в цепочку.

Сохранить данные в таблицу в DynamoDB.

Так как идентификатор заявки на доставку уникален, можно использовать его вместо создания своего идентификатора с помощью модуля uuid.

Обратите внимание на следующее:

- `minimal-request-promise` – как можно догадаться по имени, это минимально возможный API на основе `Promise` для отправки HTTP-запросов. Вы можете выбрать другой аналогичный модуль. Но мы советуем использовать `minimal-request-promise`, потому что он включает минимально необходимую реализацию. Дополнительные подробности можно узнать, заглянув в его исходный код на GitHub: https://github.com/gojko/minimal-request-promise;

- `Authorization` – чтобы послать запрос внешней службе, часто требуется пройти этап авторизации, но, так как данный пример Some Like It Hot API не является настоящим API, в заголовке `Authorization` можно послать что угодно;
- `webhookURL` – точка входа в ваш API, которая будет использоваться Some Like It Hot API для отправки уведомлений об изменении статуса заявки.

Как уже отмечалось выше, точка входа – это обычная конечная точка, принимающая запросы `POST`. Для ее реализации вы должны:

1) создать обработчик маршрута для точки входа;
2) создать маршрут `/delivery` для запросов `POST`.

Начнем с первого пункта. Перейдите в каталог `handlers` в своем проекте Pizza API и создайте новый файл `update-delivery-status.js`.

Алгоритм работы точки входа в общих чертах можно описать так:

1) точка входа должна принять запрос `POST` с идентификатором заявки на доставку и кодом состояния заявки;
2) найти заказ в таблице по идентификатору заявки, полученной от Some Like It Hot API;
3) изменить статус заявки.

Но здесь есть одна сложность. DynamoDB поддерживает две операции: `get` и `scan`. Команда `get` извлекает элементы по ключевым столбцам, а `scan` способна вернуть все элементы. Другая важная особенность состоит в том, что `scan` загружает всю таблицу, а затем применяет указанный фильтр к коллекции; команда `get` выполняет прямой запрос к таблице.

Эти различия кажутся существенными, но в действительности требуют выполнить лишь пару дополнительных шагов. Помимо единственного первичного ключа, DynamoDB поддерживает также составные ключи, состоящие из первичного, или хеш-ключа, и ключа сортировки, или диапазона, и требует, чтобы комбинация этих двух ключей была уникальной. Другой способ справиться с подобными проблемами – добавить вторичный индекс. Узнать больше об обоих подходах можно в официальной документации: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html.

В вашем случае есть еще более простое решение – идентификатор заявки на доставку уникален, и он известен до сохранения заказа в таблице `pizza-orders`, поэтому мы можем использовать его в роли идентификатора заказа. Это позволит нам запрашивать базу данных как по номеру заказа, так и по идентификатору заявки, поскольку они совпадают, а также избавиться от модуля `uuid`, потому что он вам больше не нужен.

Попробуем реализовать задуманное. Код представлен в листинге 4.2.

Листинг 4.2. Обработчик изменения статуса заявки на доставку по запросу от службы Some Like It Hot в таблице заказов

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

module.exports = function updateDeliveryStatus(request) {
  if (!request.deliveryId || !request.status)
    throw new Error('Status and delivery ID are required')

  return docClient.update({
    TableName: 'pizza-orders',
    Key: {
      orderId: request.deliveryId
    },
    AttributeUpdates: {
      deliveryStatus: {
        Action: 'PUT',
        Value: request.status
      }
    }
  }).promise()
  .then(() => {
    return {}
  })
}
```

Проверка.

Использовать DocumentClient для изменения значения в таблице.

Использовать deliveryId как первичный ключ для заказа, потому что он совпадает с идентификатором заказа.

Изменить deliveryStatus в выбранном заказе.

Вернуть пустой объект службе Some Like It Hot.

Прежде чем протестировать точку входа, нужно добавить маршрут в файл `api.js`. Для этого импортируем новый обработчик в начале файла, добавив строку `const updateDeliveryStatus = require('./handlers/update-delivery-status')`. Затем добавим еще один маршрут `POST`, как мы уже делали это в главе 2. В листинге 4.3 показаны последние несколько строк измененного файла `api.js`.

Листинг 4.3. Последние несколько строк из измененного файла api.js с новым маршрутом для точки входа `/delivery`

```
// Прежний код в файле

api.delete('/orders/{id}', request => deleteOrder(request.pathParams.id), {
  success: 200,
  error: 400
```

Прежний код в файле остался без изменений, кроме дополнительной инструкции импорта обработчика изменения статуса доставки.

```
})

api.post('/delivery', request => updateDeliveryStatus(request.body), {
  success: 200,
  error: 400
})

// Экспортировать экземпляр Claudia API Builder
module.exports = api
```

Установить признак успеха – код 200.

Установить признак ошибки – код 400.

Добавить маршрут, принимающий запросы POST и использующий обработчик updateDeliveryStatus, импортированный в начале файла.

Отлично! Теперь у нас есть точка входа и все необходимое наконец-то на месте. Опробуем точку входа. Для этого развернем наш API с помощью команды `claudia update`. Когда развертывание завершится, используйте ту же команду `curl`, которую мы использовали в главах 2 и 3 для проверки создания заказа:

```
curl -i \
  -H "Content-Type: application/json" \
  -X POST \
  -d '{"pizza":4,"address":"221b Baker Street"}'
  https://whpcvzntil.execute-api.eu-central-1.amazonaws.com/chapter4_1/orders
```

> **ПРИМЕЧАНИЕ.** Не забудьте заменить URL в команде `curl`, подставив вместо него URL, указанный в команде `claudia update`.

Команда `curl` должна вернуть `{}`, код 200, сообщающий об успехе. А теперь посмотрим, что происходит за кулисами.

> **ВРЕМЯ В SOME LIKE IT HOT API.** Чтобы упростить тестирование, через одну минуту Some Like It Hot API устанавливает для каждой заявки статус «in-progress» («выполняется»), а еще через минуту – статус «delivered» («доставлен»). То есть весь процесс, от заказа до доставки, занимает две минуты. Было бы замечательно, если бы так было в реальном мире!

Как можно видеть на рис. 4.3, наш Pizza API сначала подключается к Some Like It Hot API, а затем сохраняет заказ в таблице `pizza-orders`. Чуть позже Some Like It Hot API посылает запрос нашей точке входа и изменяет статус доставки на «in-progress» («выполняется»). И наконец, еще чуть позже снова посылает запрос точке входа, чтобы установить статус «delivered» («доставлен»).

Вот и все!

Необходимо ли подключаться к Some Like It Hot API для чего-то еще?

Имея точку входа, нам не нужно обращаться к Some Like It Hot API, чтобы получить статус доставки. Но нам нужно обратиться к API, если потребуется отменить заявку на доставку. Реализация отмены заявки послужит хорошим упражнением, и мы предложим вам реализовать ее самостоятельно в раз-

деле 4.4. Но прежде чем приступить к упражнению, рассмотрим некоторые типичные проблемы асинхронных запросов в AWS Lambda, выполняемых с использованием Claudia.

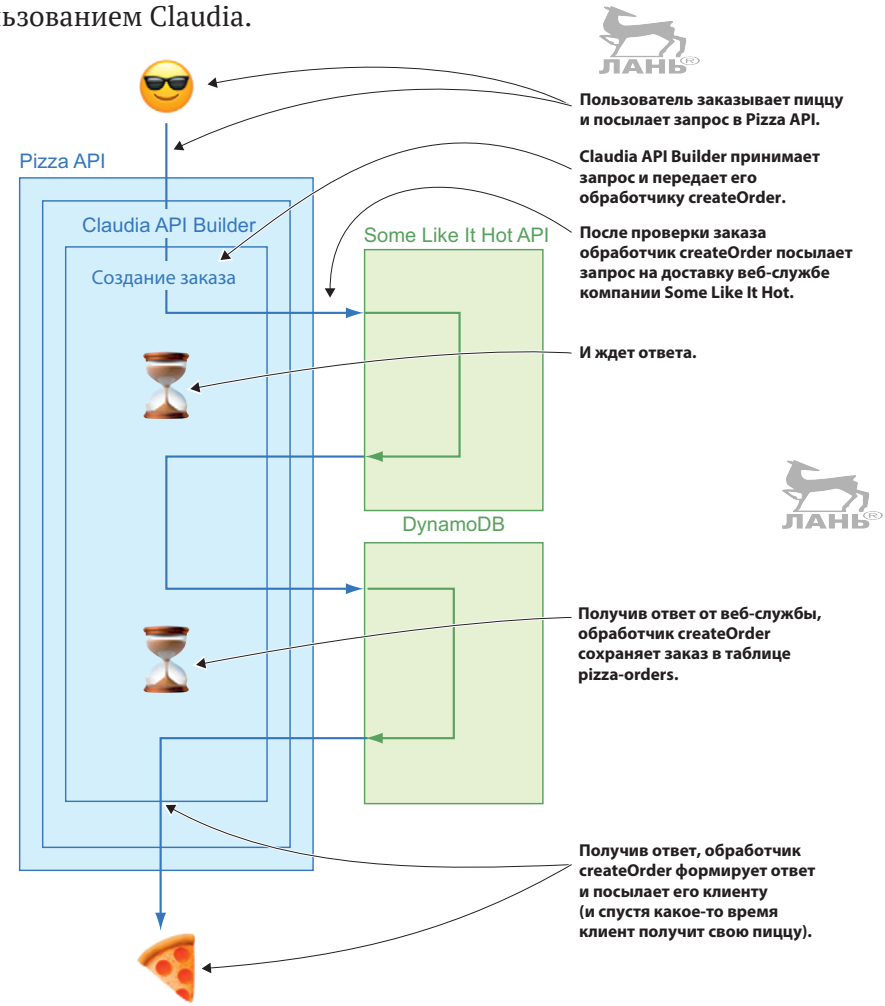


Рис. 4.3. Процесс от заказа до доставки

4.3. Типичные проблемы асинхронных взаимодействий

Как вы уже видели, AWS Lambda и Claudia упрощают работу с асинхронными запросами. Но иногда возникают проблемы, когда требуется подключиться к внешней службе или выполнить асинхронную операцию.

Перечислить все потенциальные проблемы невозможно, поэтому остановимся на наиболее распространенных ошибках, которые вы можете совершить:

- забыли вернуть `Promise`;
- не вернули значение из `.then` или `.catch`;
- не завернули вызов внешней службы в `Promise`, если она не поддерживает JavaScript-объекты `Promise`;
- превысили время ожидания до того, как асинхронная функция завершила выполнение.

Как видите, большинство проблем связано с объектами `Promise`. Давайте рассмотрим их по порядку.

4.3.1. Забыли вернуть Promise

Наиболее распространенная проблема интеграции с внешней службой или при выполнении асинхронной операции – отсутствие ключевого слова `return`. Пример этой ошибки показан в листинге 4.4. Эту проблему сложно отладить, поскольку код будет выполняться без ошибки, но выполнение остановится до выполнения асинхронной операции.

Листинг 4.4. Потеря работоспособности из-за отсутствия инструкции `return`

```
module.exports = function(pizza, address) {
  docClient.put({
    TableName: 'pizza-orders',
    Item: {
      orderId: uuid(),
      pizza: pizza,
      address: address,
      status: 'pending'
    }
  }).promise()
```

Эта строка больше не возвращает Promise.

В чем причина этой проблемы? Как показано на рис. 4.4, если асинхронная операция не возвращает `Promise`, Claudia API Builder не будет знать, что операция асинхронная, и сообщит AWS Lambda, что функция завершила свое выполнение. Он также отправит `undefined` как результат функции, потому что вы не вернули ничего значащего.

Решается эта проблема просто: убедитесь, что всегда возвращаете `Promise`, и когда код не работает, сначала убедитесь, что все экземпляры `Promise` возвращаются как нужно.

4.3.2. Отсутствие значения, возвращаемого из Promise

Эта проблема во многом похожа на предыдущую. Ее пример показан в листинге 4.5.

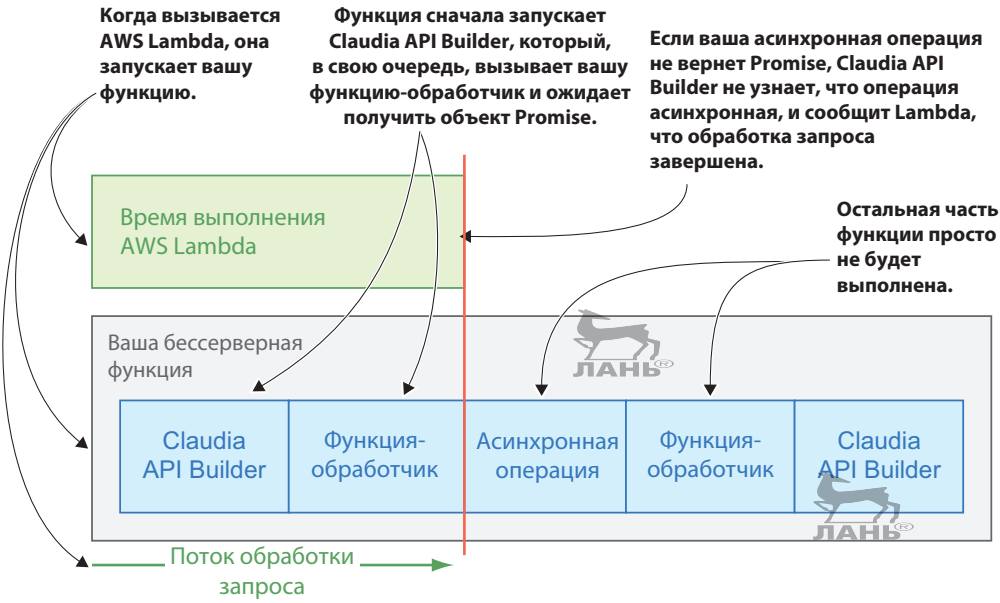


Рис. 4.4. Визуальное представление порядка выполнения функции Lambda, когда асинхронная операция не возвращает Promise

Листинг 4.5. Потеря работоспособности из-за отсутствия значения, возвращаемого из Promise

```
module.exports = function(pizza, address) {
  return docClient.put({
    TableName: 'pizza-orders',
    Item: {
      orderId: uuid(),
      pizza: pizza,
      address: address,
      status: 'pending'
    }
  }).promise()
  .then(result => {
    console.log('Result', result)
  })
```

Возвращается объект Promise, как и должно быть.

Но после журналирования запроса функция ничего не возвращает, поэтому вслед за .then ничего нельзя добавить в цепочку.

Как показано на рис. 4.5, основное отличие в том, что асинхронная операция в данном случае завершает свое выполнение, но результат никогда не возвращается обратно в вашу функцию-обработчик, и вся цепочка обещаний нарушается. И снова ваша бессерверная функция возвращает результат undefined.

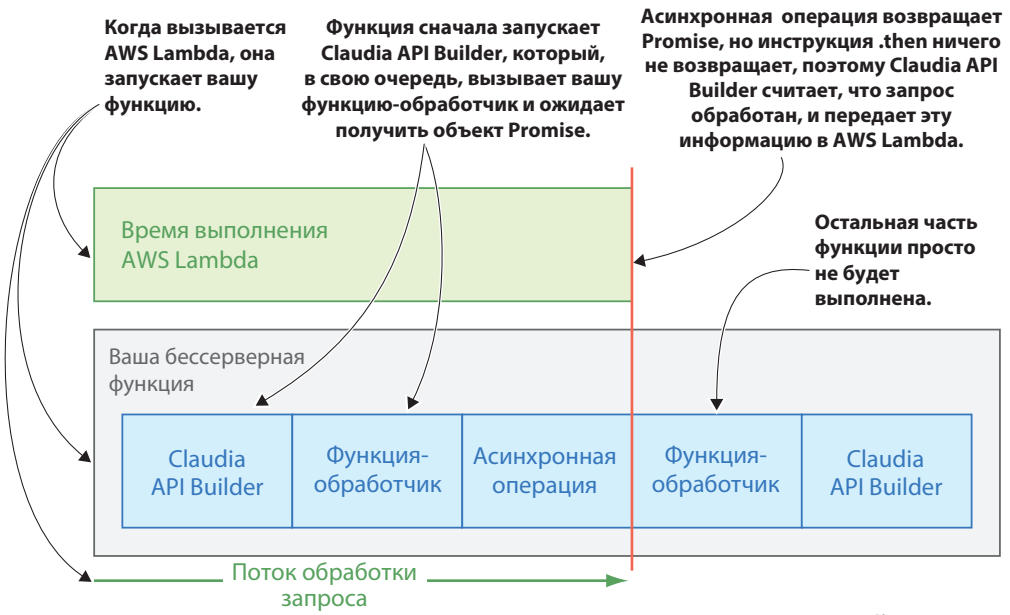


Рис. 4.5. Визуальное представление порядка выполнения функции Lambda, когда асинхронная операция не возвращает значения

Решается эта проблема так же, как предыдущая, – проверьте код, чтобы он всегда возвращал значение.

4.3.3. Вызов внешней службы не завернут в Promise

Иногда внешние и асинхронные службы не имеют встроенной поддержки `Promise`. В этом случае еще одной распространенной ошибкой является незаключение операции в экземпляр `Promise`, как показано в листинге 4.6.

Листинг 4.6. Потеря работоспособности из-за того, что асинхронная операция не завернута в Promise

```
module.exports = function(pizza, address) {
  return setTimeout(() => {
    return 'Are we there yet?'
  }, 500)
})
```

Значение возвращается, но setTimeout не возвращает Promise, поэтому данная строка разрушает цепочку обещаний.

Вы снова возвращаете значение, но это не Promise и возвращаемое значение из обратного вызова ничего не делает; эта часть кода так же никогда не выполняется.

Как показано на рис. 4.6, эта проблема в точности повторяет первую.

Но решается она немного иначе. Как показано в листинге 4.7, нужно вернуть новый, пустой объект `Promise`. Затем выполнить асинхронную операцию внутри него и, наконец, перевести объект `Promise` в состояние «выполнено», когда асинхронная операция завершится.

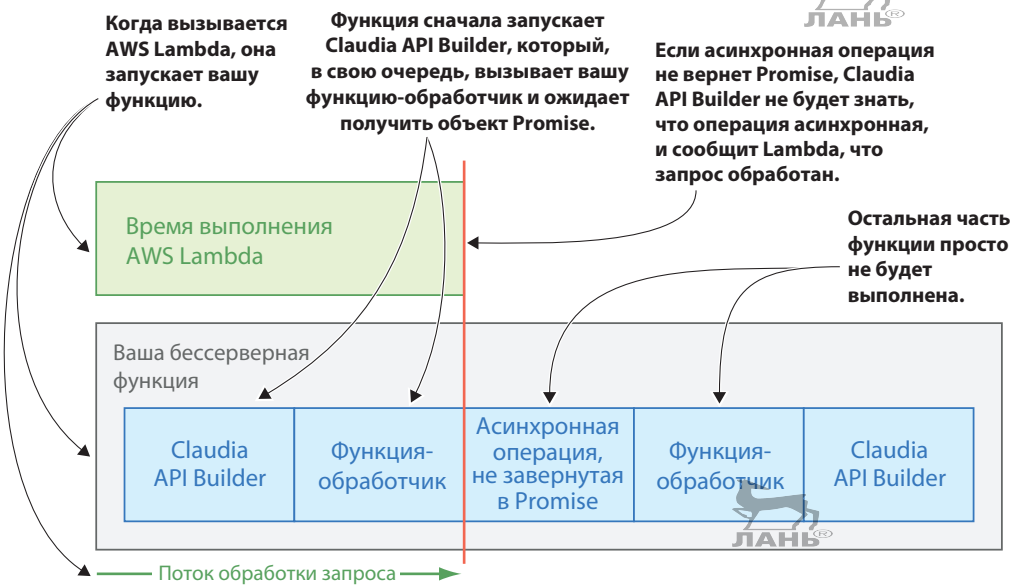


Рис. 4.6. Визуальное представление порядка выполнения функции Lambda, когда асинхронная операция не заключена в Promise

Листинг 4.7. Исправление ошибки заворачиванием асинхронного кода в объект Promise

```
module.exports = function(pizza, address) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Are we there yet?')
    }, 500)
  })
})
```

Создать и вернуть пустой объект Promise. Теперь есть возможность объявить об успехе или неудаче в функции обратного вызова.

Выполнить асинхронную операцию, которая поддерживает только функцию обратного вызова.

Когда она вернет значение, объявить об успешном выполнении Promise, указав нужное значение.

4.3.4. Превышение времени ожидания длительной асинхронной операцией

Эта последняя распространенная проблема связана с тайм-аутами в AWS Lambda. Как рассказывалось в главе 1, по умолчанию на выполнение дается три секунды. Что случится, когда асинхронная операция будет выполняться дольше трех секунд, как показано в листинге 4.8?

Листинг 4.8. Потеря работоспособности из-за превышения тайм-аута AWS Lambda

```
module.exports = function(pizza, address) {
  return new Promise((resolve, reject) => {
```

Операция setTimeout завернута в Promise и возвращает значение.

```
    setTimeout(() => {
      resolve('Are we there yet?')
    }, 3500)
  })
})
```

Операция setTimeout завернута в Promise и возвращает значение.

Но она выполняется 3.5 секунды, и если время выполнения AWS Lambda имеет значение по умолчанию, равное 3 секундам, эта асинхронная операция будет остановлена.

Итак, как показано на рис. 4.7, длительная операция, превысившая максимальное время ожидания, просто останавливается, и функция Lambda не возвращает никакого значения. Основное отличие здесь в том, что даже Claudia API Builder не получит управления. Представьте, что кто-то отключил ваш компьютер во время какой-либо операции, – эффект тот же.

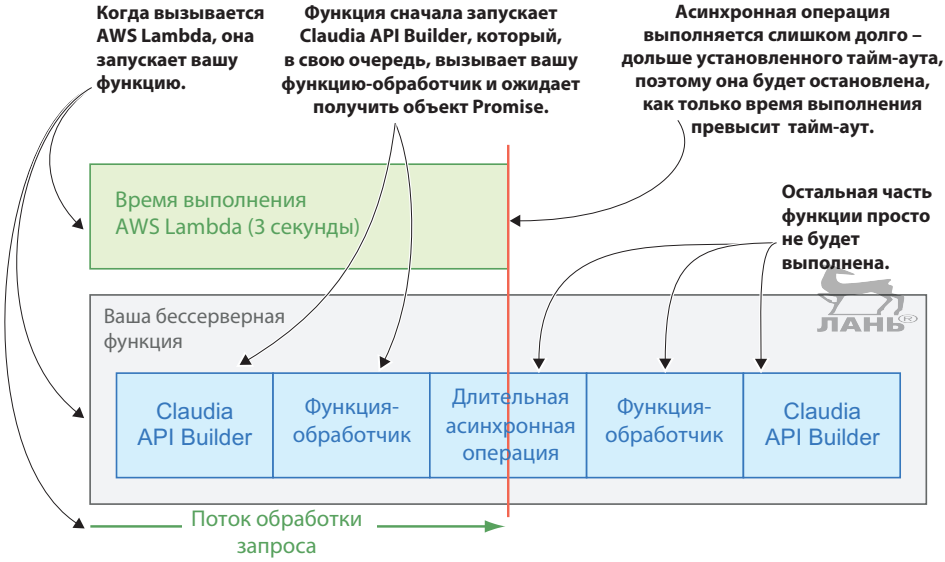


Рис. 4.7. Визуальное представление порядка выполнения функции Lambda, когда выполнение останавливается из-за превышения тайм-аута

Как исправить эту проблему?

Если нет возможности увеличить скорость асинхронной операции до уровня, когда функция уверенно будет выполняться меньше чем за три секунды, тогда остается только изменить величину тайм-аута для вашей функции.

Claudia позволяет изменить тайм-аут только во время создания функции. Для этого выполните команду `create` с параметром `--timeout`, например так:

```
claudia create --region eu-central-1 --api-module api --timeout 10
```

Значение этого параметра измеряется в секундах.

Если у вас уже есть функция, можно обновить ее, выполнив следующую команду AWS CLI:

```
claudia update --timeout 10
```

Дополнительную информацию об этой команде можно найти в официальной документации: http://docs.aws.amazon.com/cli/latest/reference/lambda/update-function-configuration.html.

После выполнения команды для вашей функции будет установлен новый 10-секундный тайм-аут. Если вы снова запустите пример из листинга 4.9, он должен работать без проблем.

Это далеко неполный список возможных проблем, но эти четыре проблемы охватывают подавляющее большинство случаев.

Теперь поиграйте с параметрами и попробуйте сломать свой бессерверный API более творческим способом!

4.4. Опробование!

Как вы уже видели, подключиться к внешним службам совсем несложно, поэтому теперь попробуйте сделать это самостоятельно.

4.4.1. Упражнение

Напомним, что нам еще нужно реализовать отмену заявки на доставку, используя Some Like It Hot API.

Ваша задача: добавьте в обработчик `delete-order.js` отмену заявки на доставку, использовав Some Like It Hot API перед удалением заказа из базы данных.

Вот некоторая информация о методе `DELETE` в Some Like It Hot API:

- чтобы удалить заявку на доставку, нужно послать в Some Like It Hot Delivery API запрос `DELETE` с маршрутом `/delivery/{deliveryId}`;
- в запросе следует указать идентификатор заявки как параметр пути в URL;
- полный URL для Some Like It Hot API имеет вид: `https://some-like-it-hot.effortless-serverless.com/delivery`;
- заказ можно удалить, только если он находится в состоянии «pending» («ожидает»).

Если этой информации вам будет достаточно, тогда идите и дерзайте!

Вот дополнительные подсказки для тех, кому перечисленной информации покажется недостаточно:

- сначала извлеките заказ из таблицы `pizza-orders`, чтобы узнать его состояние;
- если заказ находится не в состоянии «pending» («ожидает»), сгенерируйте ошибку;
- если заказ находится в состоянии «pending» («ожидает»), пошлите запрос в Some Like It Hot API; и, только получив положительный ответ, удалите заказ из таблицы `pizza-orders`.

Если и этого вам недостаточно или вы все сделали и хотите увидеть наше решение, переходите к следующему разделу.

Если упражнение покажется вам слишком простым и вы захотите усложнить его, попробуйте создать свою версию Some Like It Hot API, руководствуясь описанием в разделе 4.2.1. Вариант решения этого задания мы не приводим, но вы можете заглянуть в исходный код примера проекта по адресу: https://github.com/effortless-serverless/some-like-it-hot-delivery.

4.4.2. Решение

Начнем с алгоритма. Как мы уже говорили, сначала нужно прочитать заказ из таблицы `pizza-orders` в базе данных, чтобы убедиться, что заказ находится в состоянии «pending» («ожидает»). Затем отменить, послав запрос `DELETE` в Some Like It Hot Delivery API, и, наконец, удалить из таблицы `pizza-orders`. Этот алгоритм изображен на рис. 4.8.

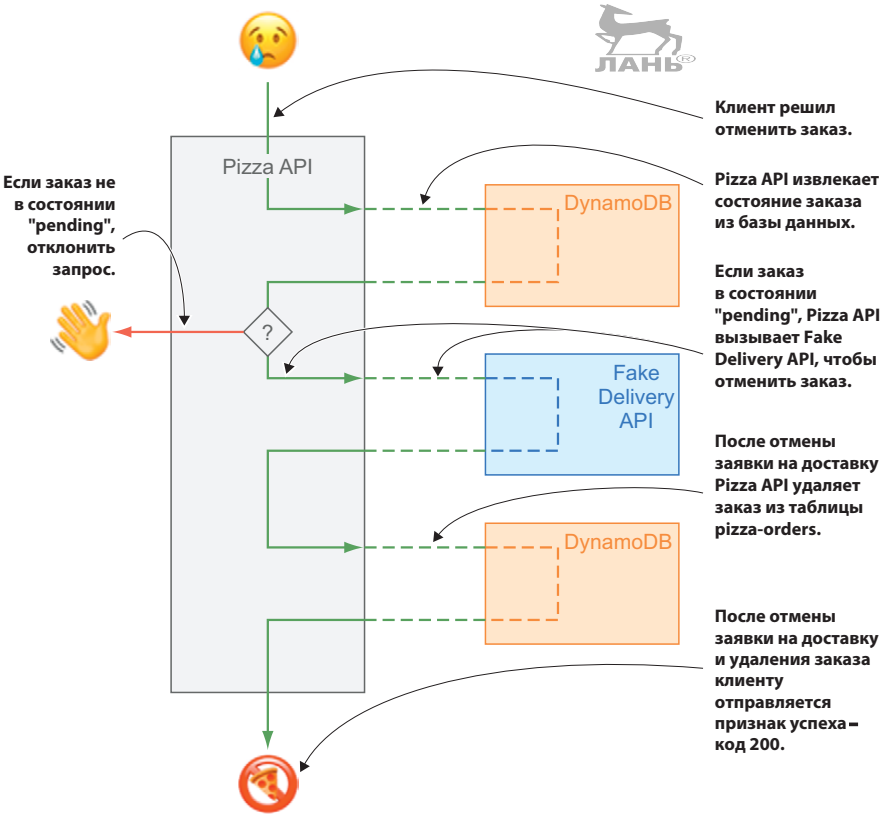


Рис. 4.8. Порядок обработки запроса на отмену заказа в Pizza API

Какие изменения следует внести в обработчик `delete-order.js`?

Здесь все просто. Во-первых, нужно импортировать модуль `minimal-request-promise`, потому что он будет использоваться для отправки запроса в Some Like It Hot API.

Затем в функции `deleteOrder` прочитать заказ из таблицы `pizza-orders`. Если заказа с указанным идентификатором не существует, функция автоматически вернет ошибку, и клиент получит код 400. Если заказ существует, нужно проверить его состояние; если состояние отличается от «pending» («ожидает»), сгенерируйте ошибку вручную.

Если заказ находится в состоянии «pending» («ожидает»), используйте модуль `minimal-request-promise`, чтобы послать запрос `DELETE` в Some Like It Hot API. Не забывайте, что идентификатор заказа совпадает с идентификатором заявки на доставку, поэтому для удаления заявки можно использовать этот идентификатор. Если Some Like It Hot API вернет ошибку, ваша функция `deleteOrder` автоматически вернет ее, поэтому клиент получит код 400, как и требуется.

Если API успешно удалит заявку на доставку, удалите заказ из таблицы `pizza-orders` – и все!

В листинге 4.9 приводится полный код обработчика `delete-order.js` после внесения всех изменений.

Листинг 4.9. Удаление заказа из таблицы `pizza-orders`

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const rp = require('minimal-request-promise')

module.exports = function deleteOrder(orderId) {
  return docClient.get({
    TableName: 'pizza-orders',
    Key: {
      orderId: orderId
    }
  }).promise()
  .then(result => result.Item)
  .then(item => {
    if (item.orderStatus !== 'pending')
      throw new Error('Order status is not pending')

    return rp.delete(`https://some-like-it-hot.effortless-serverless.com/
delivery/${orderId}`, {
      headers: {
        "Authorization": "aunt-marias-pizzeria-1234567890",
        "Content-type": "application/json"
      }
    })
  })
  .then(() => {
    return docClient.delete({
```

Импортировать модуль minimal-request-promise.

Извлечь заказ из таблицы pizza-orders.

Если заказ не в состоянии «pending», сгенерировать ошибку.

Удалить заявку на доставку, обратившись к Some Like It Hot API.

Удалить заказ из таблицы pizza-orders.

```
      TableName: 'pizza-orders',
      Key: {
        orderId: orderId
      }
    }).promise()
  })
}
```

Вызовы .then и .catch были удалены, потому что результат операции будет служить результатом вызова функции.

В заключение

- В AWS Lambda можно подключиться к любой внешней службе, как в любом приложении Node.js, если асинхронные операции выполняются правильно.
- Подключаясь к внешнему API, убедитесь, что ваша библиотека HTTP поддерживает механизм обещаний `Promise`, или заворачивайте асинхронные операции в этот объект.
- Подключение к внешним службам сопряжено с некоторыми потенциальными проблемами; чаще всего они связаны с разрушением цепочки обещаний (`Promise`).
- Еще одна распространенная проблема связана с превышением тайм-аута – если для выполнения функции Lambda требуется больше трех секунд, увеличьте тайм-аут для своей функции.

Глава 5

Хьюстон, у нас проблема!

Эта глава охватывает следующие темы:

- чтение вывода в консоль с помощью CloudWatch;
- проблемы отладки бессерверных приложений;
- отладка бессерверных API.

Мы – люди – склонны ошибаться. Что бы мы не делали, всегда есть шанс ошибиться, даже если мы сделаем все возможное, чтобы этого не допустить. Это особенно верно в отношении разработки программного обеспечения. Доводилось ли вам наблюдать, как зависает мобильное приложение или веб-сайт перестает отвечать? Скорее всего, вы наблюдали это совсем недавно, и вам приходилось обновлять страницу в браузере или перезапускать приложение.

Все мы допускаем ошибки, и приложения тоже зависают каждый день. Несмотря на то что ошибки в приложениях обычно безвредны, иногда они могут приводить к огромным потерям. Рассмотрим пример ошибки в приложении для пиццерии, которая не позволяет создавать заказы. Как найти ошибку? Как выполнить отладку бессерверного приложения?

В этой главе вы узнаете, как искать ошибки в бессерверных приложениях, как их отлаживать и какие инструменты отладки имеются в вашем распоряжении.

5.1. Отладка бессерверного приложения

Мы быстро движемся вперед, и тетушка Мария отправила вам сообщение, что наняла разработчика мобильных приложений, Пьера. Она хотела увеличить охват своих клиентов, и мобильное приложение для заказа пиццы показалось ей хорошим началом. Пьер решил опробовать ваше бессерверное приложение. К сожалению, когда он попытался создать заказ, приложение вернуло неверный ответ. Пьер пожаловался тетушке, и теперь тетушка звонит вам. Вы, вероятно, почесываете голову, думая «Где я мог ошибиться?» и «Как отладить ошибку?».

В традиционном серверном приложении Node.js можно просто добавить команду `console.log("некоторый текст")` в программу, чтобы вывести какой-либо текст или объекты в консоль, или даже использовать отладчик, чтобы расставить точки останова в коде для отладки, а потом запустить код локально и попробовать найти ошибку или зайти на сервер и просмотреть отладочный вывод.

Отладка бессерверных приложений отличается от традиционных. Бессерверное приложение часто состоит из совершенно отдельных модулей – API Gateway и функции Lambda, поэтому у вас не получится запустить его локально и отладить весь поток приложения. Кроме того, поскольку приложение не имеет сервера, нет никакого сервера, куда можно было бы зайти, чтобы исследовать журналы. Да, это звучит странно и расстраивает, но не волнуйтесь.

Каждый поставщик услуг бессерверных вычислений предлагает инструменты, чтобы помочь вам проверить и отладить ваши бессерверные функции. В AWS это CloudWatch.

CloudWatch – это служба AWS, предназначенная для трассировки, регистрации и мониторинга ресурсов в AWS. Ее можно считать бессерверной версией старого доброго журнала на сервере, хотя она способна на большее. Подобно другим службам AWS, CloudWatch доступна из командной строки AWS CLI, и мы будем использовать ее из своего терминала.

Так как мы пользуемся услугами AWS, служба CloudWatch – наш выбор по умолчанию.

> **ПРИМЕЧАНИЕ.** Бессерверную функцию можно запустить локально, но это не значит, что она будет выполняться так же, как в бессерверном окружении. В Azure есть возможность запустить функцию в Visual Studio, а в Google Cloud Platform есть локальный эмулятор для локальной отладки, но ни один поставщик не рекомендует использовать свой эмулятор для промышленного использования, поскольку оба находятся на альфа-стадии разработки.

AWS CloudWatch можно использовать из:

- веб-консоли AWS в окне браузера;
- AWS CLI в терминале;
- AWS API;
- AWS SDK (в зависимости от выбранного языка программирования).

Вы можете использовать любой из этих вариантов, который вам нравится, но в этой книге мы будем работать с интерфейсом командной строки AWS CLI, потому что он удобен для разработчиков и его можно использовать в окне локального терминала.

CloudWatch – это простая служба, которая регистрирует вывод и сообщения об ошибках из ваших бессерверных функций. Всякий раз, когда вы что-то выводите в своей функции, например с помощью `console.log`, этот вывод автоматически отправляется в AWS CloudWatch. AWS CloudWatch отвечает за их

хранение и группировку. Вы можете получить доступ к этим журналам через интерфейс командной строки AWS CLI или через пользовательский интерфейс веб-консоли AWS. На рис. 5.1 показано, как это работает.

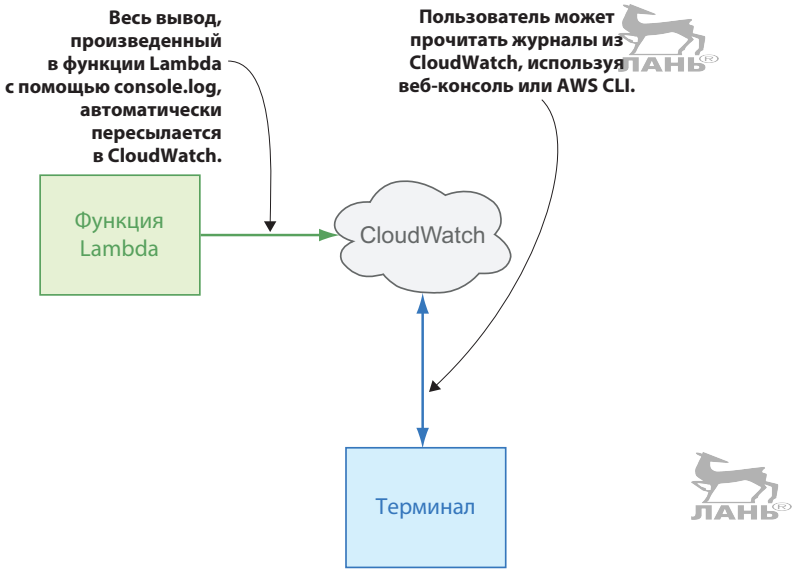


Рис. 5.1. AWS Lambda посылает вывод console.log непосредственно в CloudWatch

> **ПРИМЕЧАНИЕ.** Захват вывода в CloudWatch не влияет на время отклика функции Lambda. Но журналы доступны не сразу; между вызовом функции и появлением журналов в CloudWatch имеется задержка не менее нескольких секунд.
>
> По умолчанию журналы CloudWatch хранятся неопределенно долго, но вы можете уточнить, сколько должны храниться разные группы журналов.
>
> CloudWatch имеет бесплатный тариф, но количество журналов и срок хранения могут повлиять на вашу ежемесячную плату. За дополнительной информацией обращайтесь по адресу: https://aws.amazon.com/cloudwatch/pricing/.

5.2. Отладка функции Lambda

Теперь, узнав, что такое CloudWatch, используем эту службу, чтобы найти источник проблем Пьера. Пьер сообщает, что ошибка возникает, когда он пытается создать заказ на пиццу с типом пиццы и адресом доставки. Мы должны попытаться воспроизвести проблему, включив мониторинг журналов в CloudWatch. Добавим оператор `log` в начало обработчика `create-order.js`, повторно развернем API и попросим Пьера повторить попытку.

Мы должны зарегистрировать запрос с каким-либо сопроводительным текстом – например, «Сохранить заказ» – в первой строке функции `createOrder`, как показано в листинге 5.1. Такой текст поможет отыскать нужные сообщения.

(В этом листинге показан только начальный фрагмент из файла; остальной код не изменился.)

Листинг 5.1. Измененный обработчик create-order.js

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const rp = require('minimal-request-promise')

module.exports = function createOrder(request) {
  console.log('Save an order', request)

  if (!request || !request.pizza || !request.address)
    throw new Error('To order pizza please provide pizza type and address
      where pizza should be delivered')

  // ...
```

Вывести информацию о запросе и добавить текст «Save an order» в начало.

Остальной код в файле не изменился.

Пьер попытался снова создать заказ и получил ту же ошибку. Теперь мы должны найти в журналах текст «Save an order» («Сохранить заказ»). Просмотр журналов CloudWatch для нашей функции Lambda может оказаться непростым делом из-за большого числа записей с кучей метаданных. К счастью, мы можем ускорить работу с помощью интерфейса командной строки AWS CLI и команды `logs filter-log-events`.

Так как CloudWatch сохраняет журналы в группах, перед запуском команды `logs filter-log-events` нужно найти имя вашей группы журналов. Для этого воспользуемся командой `describe-log-groups`, как показано в листинге 5.2.

Листинг 5.2. Команда describe-logs-groups

```
aws logs describe-log-groups --region eu-central-1
```

Эта команда вернет ответ, включающий имя группы `logGroupName`, например:

```
{
  "logGroups": [
    {
      "arn": "arn:aws:logs:eu-central-1:123456789101:log-group:/aws/lambda/
   pizza-api:*",
      "creationTime": 1524828117184,
      "metricFilterCount": 0,
      "logGroupName": "/aws/lambda/pizza-api",
      "storedBytes": 1024
```

```
    }
  ]
}
```

ПРИМЕЧАНИЕ. Помимо `filter-log-events`, служба `logs` в AWS CLI предлагает несколько других полезных команд. Чтобы получить полный список доступных команд, запустите команду `aws logs help` в терминале.

Запустите в терминале команду `logs filter-log-events` и добавьте текст «Save an order» в качестве фильтра, как показано в листинге 5.3. Вы также можете указать формат вывода (в этом примере используем формат JSON).

Листинг 5.3. Запрос журналов из CloudWatch с выбором группы и текстом «Save an order» для фильтрации

```
aws logs \
  filter-log-events \
  --filter='Save an order' \
  --log-group-name=/aws/lambda/pizza-api \
  --region=eu-central-1 \
  --output=json
```

Использовать службу logs из AWS CLI.

Использовать команду filter-log-events для фильтрации журналов.

Добавить искомый текст.

Показать только искомые сообщения из группы /aws/lambda/pizza-api.

Вывести результат в формате JSON.

Команда в листинге 5.3 поможет нам получить вывод, произведенный инструкцией `console.log` в нашей функции Lambda. Но, как показано в листинге 5.4, результаты возвращаются в формате JSON с большим количеством метаданных, которые нам, по большому счету, не нужны. Единственное, что вас волнует, – это сообщение «message» в каждом из событий. Все остальное в ответе – это метаданные о журналах, в которых произведен поиск, и некоторая дополнительная информация о сообщениях.

Листинг 5.4. Журнал с метаданными, сгенерированный Pizza API в CloudWatch

```
{
  "searchedLogStreams": [
    {
      "searchedCompletely": true,
      "logStreamName": "2017/06/18/[$LATEST]353ce211793946dba5bb276b0bde3e0e"
    }
  ],
  "events": [
    {
      "ingestionTime": 1497802509940,
```

```
      "timestamp": 1497802509920,
      "message": "2017-06-18T16:15:09.860Z\t4cc844ea-5441-11e7-8919-29f1e77e006c\
tSave an order
      { pizza: 1,\n adress: '420 Paper St.' }\n",
      "eventId": "33402112131445556039184566359053029477419337484906135552",
      "logStreamName": "2017/06/18/[$LATEST]e24e0cab3d6f47f2b03005ba4ca16b8b"
    }
  ]
}
```

Кроме того, вывод в формате JSON не очень удобочитаем, если отформатирован как однострочный текст. Вы можете улучшить форматирование, указав тип вывода `text` и повторно выполнив команду, как показано в листинге 5.5.

Листинг 5.5. Измененный запрос журналов из CloudWatch с выбором группы и текстом «Save an order» для фильтрации

```
aws logs \
  filter-log-events \
  --filter='Save an order' \
  --log-group-name=/aws/lambda/pizza-api \
  --query='events[0].message' \
  --region=eu-central-1 \
  --output=text
```

Потребовать вернуть только последнее событие.

Изменить формат вывода.

Эта команда вернет более ясный и понятный текст, как показано в листинге 5.6.

Листинг 5.6. Сообщения, сгенерированные Pizza API, без метаданных

```
2017-06-18T16:15:09.860Z 4cc844ea-5441-11e7-8919-29f1e77e006c
  Save an order { pizza: 1, adress: '420 Paper St.' }
```

Команда вернула только одно сообщение

Этот вывод выглядит намного яснее и полезнее. И да, посмотрите на это, Пьер допустил опечатку! Он отправлял параметр с именем `adress` вместо `address`. Такая суета из-за простой орфографической ошибки! Но мы не будем звонить тетушке, чтобы объяснить проблему.

5.3. Рентген для приложения

Отладка бессерверных приложений иногда бывает трудной, потому что сложно представить поток данных, но в AWS есть инструмент, который вам в этом поможет. AWS X-Ray – это служба, отображающая почти в реальном време-

ни потоки данных внутри приложения и все задействованные службы. Службу X-Ray можно использовать с приложениями, работающими на EC2, ECS, Lambda и Elastic Beanstalk. Кроме того, X-Ray SDK автоматически собирает метаданные для всех вызовов служб AWS, выполняемых с использованием AWS SDK. На рис. 5.2 и 5.3 показано, как выглядит наш Pizza API, с точки зрения AWS X-Ray.

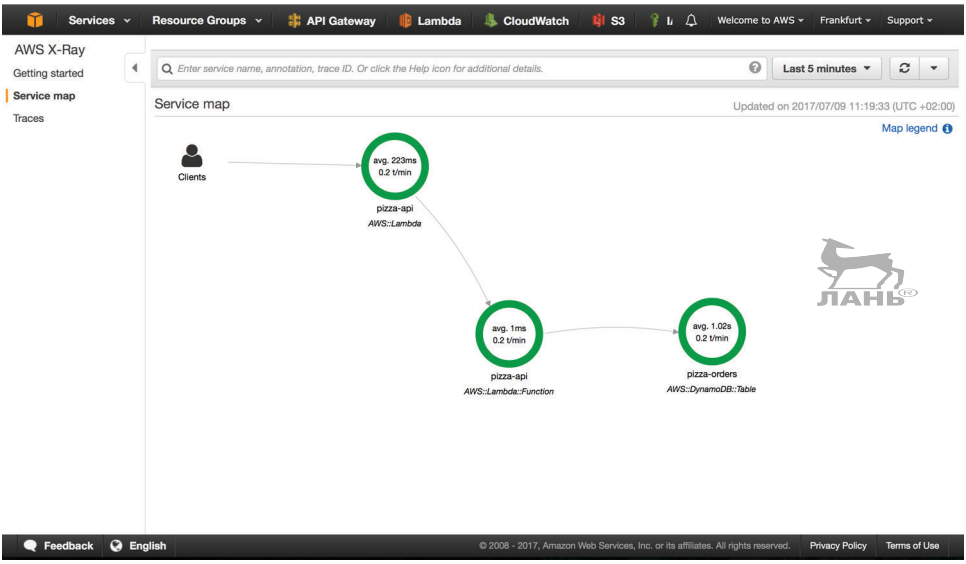


Рис. 5.2. Визуальное представление потока передачи строки «Create an order» в Pizza API

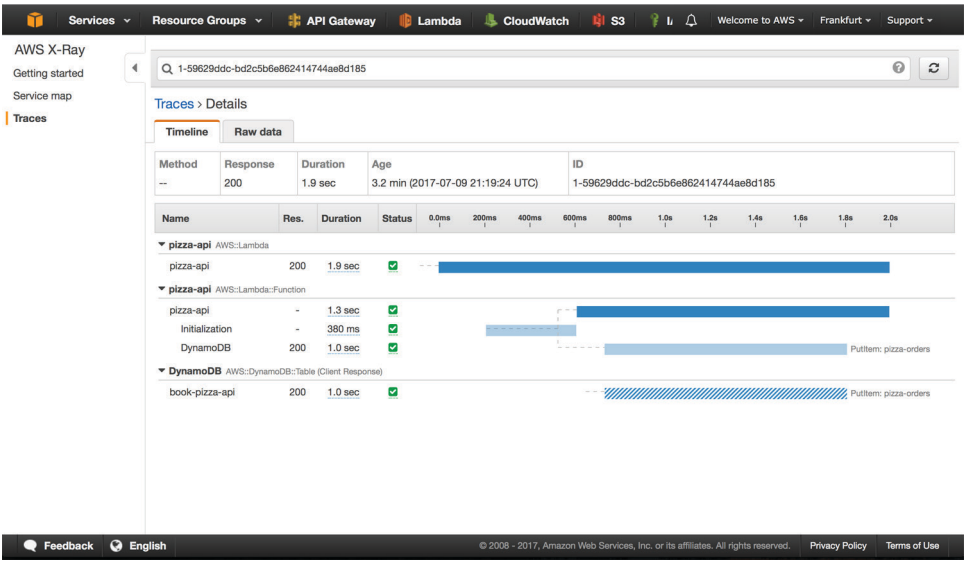


Рис. 5.3. Подробное визуальное представление потока передачи строки «Create an order» в Pizza API

Чтобы включить поддержку AWS X-Ray в функции Lambda, нужно добавить политику, которая разрешит службе X-Ray взаимодействовать с ней, и включить режим активной трассировки в настройках функции.

AWS X-Ray и AWS Lambda

AWS Lambda использует Amazon CloudWatch, чтобы автоматически создать метрики и журналы для всех вызовов вашей функции. Но этот механизм не очень удобен для отслеживания источников событий, вызвавших вашу функцию Lambda, или исходящих вызовов, которые делает ваша функция. И тут на сцену выходит AWS X-Ray. Интеграция с X-Ray для AWS Lambda осуществляется легко и просто, потому что когда выполняется AWS Lambda, демон X-Ray уже запущен.

В этом разделе мы покажем только самый простой случай интеграции нашей функции Lambda с X-Ray; чтобы узнать больше, загляните в официальное руководство по адресу: http://docs.aws.amazon.com/lambda/latest/dg/lambda-x-ray.html.

ПРИМЕЧАНИЕ. Мы будем использовать веб-консоль AWS для работы с AWS X-Ray, потому что в терминале нельзя увидеть визуальное представление приложения.

Итак, посмотрим, как добавить политику и включить режим активной трассировки с помощью интерфейса командной строки AWS CLI. Чтобы добавить политику, снова используем команду `iam attach-role-policy`, но теперь с `arn:aws:iam::aws:policy/AWSXrayWriteOnlyAccess`, как показано в листинге 5.7.

Листинг 5.7. Добавление политики для X-Ray в роль Lambda

```
aws iam \
  attach-role-policy \
  --policy-arn arn:aws:iam::aws:policy/AWSXrayWriteOnlyAccess \
  --role-name pizza-api-executor \
  --region eu-central-1 \
  --output json
```

Использовать службу iam из AWS CLI.

Использовать команду attach-role-policy для добавления политики.

Указать ARN для добавляемой политики.

Выбрать желаемую роль для политики.

Как вы уже знаете, при успешном выполнении эта команда возвращает пустой результат.

Следующий шаг – обновление конфигурации функции. Сделать это можно с помощью команды `lambda update-function-configuration`. Она ожидает имя функции и параметры; в данном случае нам нужно обновить параметр `tracing-config`, установив в нем режим `Active`. В листинге 5.8 приводится полная команда.

Листинг 5.8. Включение режима активной трассировки в AWS X-Ray

```
aws lambda \                                  ← Использовать службу lambda из AWS CLI.
  update-function-configuration \             ← Обновить конфигурацию функции.
  --function-name pizza-api \                 ← Имя функции.
  --tracing-config Mode=Active \              ← Выбор режима активной трассировки.
  --region eu-central-1
```

Эта команда вернет конфигурацию функции Lambda в формате JSON, как показано в листинге 5.9. Теперь служба X-Ray готова визуализировать поток нашей функции Lambda, но по умолчанию мы не сможем увидеть другие службы AWS, которые использует наша функция, такие как DynamoDB.

Листинг 5.9. Ответ после включения трассировки в X-Ray

```
{
  "TracingConfig": {                          ← Режим активной трассировки.
    "Mode": "Active"
  },
  "CodeSha256": "HwV+/VdUztZ782NBEqY9Dvzj3nxF6tigLOZPt8yyCoU=",
  "FunctionName": "pizza-api",                ← Информация о функции, включая имя,
  // ...                                        ARN, версию и другие метаданные.
}
```

Чтобы увидеть другие службы AWS, поддерживаемые X-Ray, нужно завернуть AWS SDK для Node.js в модуль `aws-xray-sdk-core`. После установки этого модуля из NPM измените обработчик `create-order.js`, как показано в листинге 5.10.

Листинг 5.10. Измененный обработчик create-order.js для обертывания AWS SDK функцией X-Ray

```
'use strict'                                                  ← Импортировать модуль
                                                                aws-xray-sdk-core.
const AWSXRay = require('aws-xray-sdk-core')
const AWS = AWSXRay.captureAWS(require('aws-sdk'))            ← Завернуть модуль aws-sdk
const docClient = new AWS.DynamoDB.DocumentClient()             в команду AWSXRay.captureAWS.

module.exports = function updateDeliveryStatus(request) {
  console.log('Save an order', request)

  if (!request.deliveryId || !request.status)
    throw new Error('Status and delivery ID are required')

// ...                                                        ← Остальной код в файле остался без изменений.
```

После запуска команды `claudia update` для повторного развертывания API служба X-Ray будет полностью настроена.

Чтобы увидеть визуальное представление функции, перейдите в раздел X-Ray в веб-консоли AWS. В этом случае URL имеет вид: https://eu-central-1.console.aws.amazon.com/xray/home?region=eu-central-1#/service-map. Ваш URL может отличаться, если вы использовали другой регион для развертывания функции.

5.4. Опробование!

Упражнение для этой главы довольно простое, но в следующей главе мы рассмотрим более сложные темы.

5.4.1. Упражнение

Теперь, когда вы узнали, как отлаживать бессерверные приложения, вернитесь к листингам из глав 3 и 4 и попробуйте прочитать их журналы.

Ваша задача – попытаться прочитать журналы CloudWatch и отыскать все сообщения об успешных операциях и ошибках в обработчике `create-order.js`.

Поскольку это всего лишь отладочное упражнение, мы не будем давать никаких советов. Если вам нужна помощь, можете подсмотреть решение в следующем разделе.

5.4.2. Решение

В главе 3 вы добавили в обработчик `create-order.js` вывод сообщений об успехах и ошибках. Чтобы прочитать эти журналы с помощью CloudWatch, используйте команду `aws logs filter-log-events`. Как вы уже знаете, она требует указать фильтр. Напоминаем, что успешные сообщения регистрировались с префиксом «Order is saved!». Для ошибок мы использовали префикс «Oops, order is not saved :(». Используйте оба эти префикса для фильтрации журналов.

Команда для извлечения записей с текстом «Order is saved!» приводится в листинге 5.11.

Листинг 5.11. Команда для извлечения из CloudWatch записей с текстом «Order is saved!»

```
aws logs \
  filter-log-events \
  --filter='Order is saved!' \
  --log-group-name=/aws/lambda/pizza-api \
  --query='events[0].message' \
  --output=text
```

Выбрать из CloudWatch записи с текстом «Order is saved!»

> **ПРИМЕЧАНИЕ.** Эту же команду можно использовать для извлечения записей с текстом «Oops, order is not saved :(», чтобы прочитать ошибки. Но поскольку наше сообщение содержит запятую и двоеточие, которые считаются специальными символами, безопаснее использовать только часть текста в фильтре – например, «order is not saved».

Ответы, возвращаемые командами, будут различаться в зависимости от количества успешных и неудачных попыток оформить заказ. Если ошибок нет, на выходе появится ответ None.

В заключение

- Для чтения журналов функций Lambda используйте CloudWatch.
- Вместо фильтрации журналов вручную можно использовать разные команды из AWS CLI.
- Для визуализации потока выполнения функции можно использовать службу AWS X-Ray.

Глава 6

Совершенствование API

Эта глава охватывает следующие темы:

- как осуществляются аутентификация и авторизация в бессерверных приложениях;
- реализация аутентификации и авторизации в нашем бессерверном приложении;
- идентификация пользователей с помощью социальных сетей.

Аутентификация и авторизация – одна из множества сложностей, с которыми приходится сталкиваться при разработке распределенных приложений. Проблема состоит в том, чтобы передать информацию об авторизованном пользователе вместе с его привилегиями всем распределенным службам, составляющим приложение, и правильно интегрировать сторонние механизмы аутентификации.

Эта глава покажет вам, как реализовать аутентификацию и авторизацию в нашем бессерверном приложении для удобства клиентов тетушки Марии. Здесь вы узнаете разницу между аутентификацией и авторизацией в бессерверной среде и как реализовать механизм веб-авторизации с помощью AWS Cognito. Затем вы научитесь идентифицировать пользователей с помощью социальных сетей – в нашем случае с помощью Facebook.

6.1. Бессерверная аутентификация и авторизация

Тетушка Мария и Пьер, нанятый ею разработчик мобильного приложения, с которым мы столкнулись в предыдущей главе, сообщили нам, что наш API в ответ на запрос списка заказов возвращает все заказы, независимо от того, кто послал запрос. Все заказы должны видеть только сотрудники пиццерии. Клиентам должны быть доступны лишь их заказы. Не клиенты и не сотрудники вообще ничего не должны видеть.

Вот как мы исправим эту проблему:

1) добавим в приложение поддержку аутентификации пользователей двумя способами:
   - по адресу электронной почты;
   - по учетной записи в Facebook;
2) создадим список пользователей нашего API и позволим каждому пользователю видеть только его заказы.

Аутентификация и авторизация

Вы, наверное, заметили, что два разных существительных – аутентификация и авторизация – выглядят похожими, но в действительности они соответствуют двум разным понятиям. В сочетании с другими понятиями, такими как идентификация и привилегии, они могут вызывать немалые сложности.

Попробуем разобрать их на примере.

Представьте, что наше приложение – это крупная компания, владеющая или арендующая офисное здание. Часто такие офисные здания охраняются, чтобы в здание никто не мог войти, кроме сотрудников компании. Поскольку служба безопасности здания должна знать, кому разрешено входить, сотрудникам этой службы обычно передается список работников компании с информацией о них, включая фотографии.

Если человек попытается войти в здание, охранник остановит его и потребует предъявить информацию, **идентифицирующую** личность человека. Если человек не предъявит никакой идентифицирующей информации, охранник запретит вход и выведет его из здания. Если человек предъявит идентифицирующую информацию, охранник проверит ее, чтобы убедиться в достоверности. Этот процесс называется **аутентификацией**.

Если представленная информация достоверна, человек пройдет проверку *подлинности* (*аутентифицируется*). Но затем охранник проверит, присутствует ли человек, пытающийся войти, в списке сотрудников компании. Если этого человека нет в списке, охранник запретит вход. Если человек присутствует в списке, ему будет разрешено войти. Этот процесс называется авторизацией.

А теперь интересный вопрос: имеет ли право любой сотрудник тратить деньги с банковского счета компании? Если сотрудник не является генеральным директором, скорее всего, он не имеет такого права (иногда таким правом не обладает даже генеральный директор). Право тратить деньги компании или делать что-то ограничительное называется **привилегией**.

Проще говоря:

- *аутентификация* – это проверка личности пользователя; действительно ли он является тем, кем себя называет;
- *авторизация* – это проверка права войти;
- *идентификационная информация* – информация, описывающая личность пользователя;
- *привилегии* – набор прав на выполнение некоторых действий.

Основываясь на опыте работы с Express.js или другими более традиционными приложениями, вы, вероятно, захотите реализовать аутентификацию как часть API и хранить список пользователей в таблице базы данных. В принципе, это возможно, но для бессерверных приложений мы рекомендуем использовать другой способ.

Авторизация необходима в большинстве приложений и обычно осуществляется вводом комбинации из адреса электронной почты и пароля. Практически всегда авторизация реализуется аналогичным, если не идентичным образом. Поэтому практически все провайдеры бессерверных вычислений предлагают встроенные службы аутентификации и авторизации для работы с бессерверными ресурсами. В частности, Amazon предлагает AWS Cognito – службу управления пользователями, которая решает задачи аутентификации и авторизации пользователей, управляет доступом и обеспечивает передачу информации о пользователях между службами.

Amazon Cognito поддерживает два основных механизма, каждый со своей областью ответственности:

- *пулы пользователей* – служба, отвечающая за управление идентификацией, предлагающая возможность авторизации «из коробки». Проще говоря, это набор каталогов (*пулов пользователей*), позволяющий определить свой механизм авторизации. Для нашего мобильного и веб-приложения мы можем реализовать авторизацию пользователей с помощью AWS Cognito SDK.
  Пул пользователей представлен единственной коллекцией, или каталогом пользователей;
- *федеративная идентификация* (также этот механизм называют пулами идентификации) – служба, отвечающая за взаимодействие с провайдерами аутентификации и временную авторизацию для доступа к ресурсам AWS. Служба федеративной идентификации предлагает:
  - интеграцию с механизмами идентификации социальных сетей (таких как Facebook, Google и OpenId) и провайдером идентификации вашего пула пользователей Cognito;
  - временный доступ к ресурсам приложения AWS для аутентифицированных пользователей.

  Служба федеративной идентификации хранит каталоги с информацией об отдельных пользователях. Она определяет момент входа каждого пользователя с использованием разных механизмов идентификации. Для хранения актуальных данных о пользователях требует подключения пулов пользователей Cognito.

Одним из ключевых преимуществ AWS Cognito является авторизация запросов до того, как они попадут в ваше бессерверное приложение. Это делается путем настройки авторизации на уровне шлюза API Gateway. Если пользователь не авторизован, его запросы будут остановлены до попадания в вашу функцию Lambda, что может сэкономить немало времени и денег. Несмотря

на то что услуги AWS Lambda стоят недорого, дополнительное сокращение расходов никогда не будет лишним.

В примере с пиццерией тетушки Марии нам нужно настроить оба механизма – пулы идентификации и пулы пользователей Cognito. Пул идентификации позволит нам интегрировать службу идентификации в Facebook, а также даст временный доступ к нашему пулу пользователей Cognito без жесткой привязки к нашим ключам доступа к AWS в мобильном и веб-приложении. Пул пользователей будет управлять базой данных пользователей, которые могут заказать пиццу.

Мы должны разрешить клиентам пиццерии тетушки Марии производить аутентификацию через Facebook. Как показано на рис. 6.1, процесс аутентификации через Facebook включает следующие шаги:

1) аутентифицировать пользователя в Facebook и получить ключ доступа;
2) передать ключ доступа в пул идентификации Cognito, который предоставит временный доступ к пулу пользователей Cognito;
3) использовать пул пользователей Cognito для входа или регистрации пользователя. После успешного входа или регистрации пул пользователей вернет ключ JWT;
4) использовать полученный ключ JWT для соединения с Pizza API и создания нового заказа либо получения списка прошлых заказов.

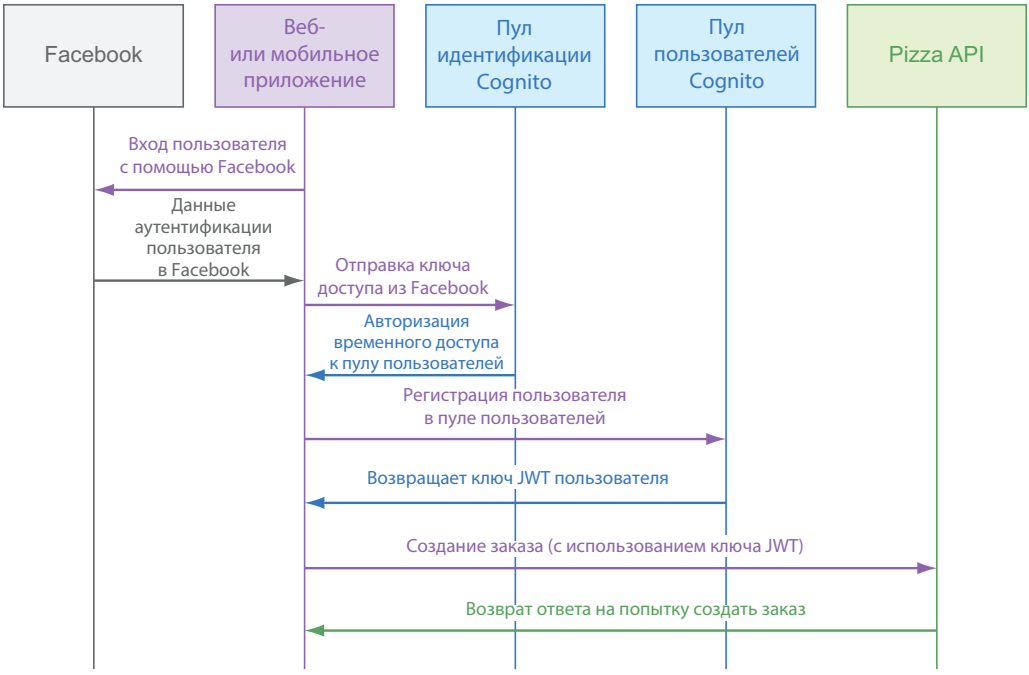


Рис. 6.1. Процесс авторизации пользователя в бессерверном Pizza API с использованием Facebook, пула пользователей и пула идентификации

Как показано на рис. 6.2, аутентификация с использованием адреса электронной почты и пароля осуществляется аналогично:

1) запросить у службы пула идентификации Cognito временный доступ к пулу пользователей Cognito;
2) выполнить вход или регистрацию в пуле пользователей Cognito с использованием адреса электронной почты и пароля. После успешного входа или регистрации пул пользователей вернет ключ JWT;
3) использовать ключ JWT для соединения с Pizza API и создания нового заказа или получения списка прошлых заказов.

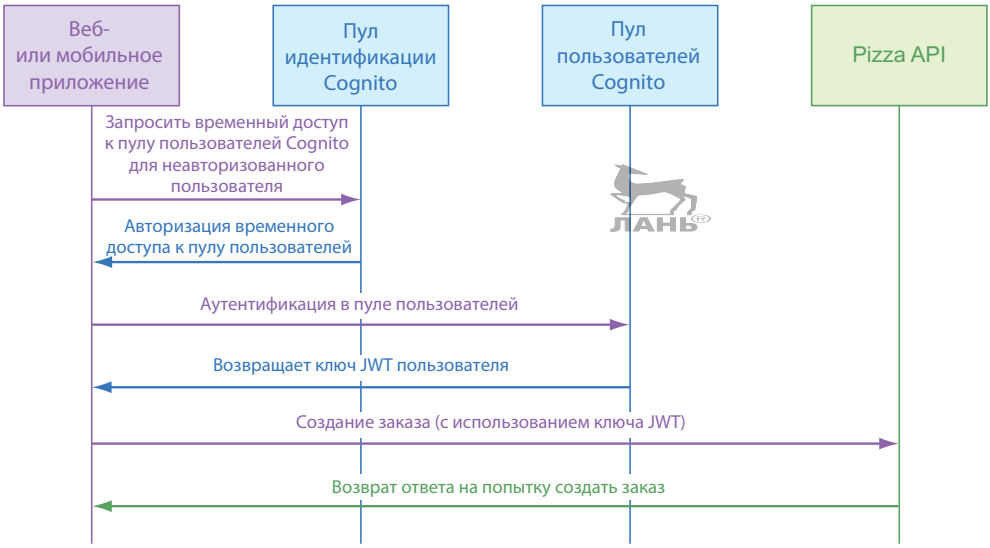


Рис. 6.2. Процесс авторизации пользователя в бессерверном Pizza API с использованием адреса электронной почты и пароля

6.2. Создание пулов пользователей и идентификации

Для реализации процедуры аутентификации, описанной в предыдущем разделе, нужно создать пулы пользователей и идентификации.

Начнем с пула пользователей. Чтобы создать его, выполните в терминале команду `aws cognito-idp create-user-pool`. Эта команда имеет единственный обязательный параметр – имя нового пула. Кроме имени пула, добавьте также в команду ключ `--username-attributes`, указывающий, что для идентификации пользователей будут применяться адреса электронной почты. Дополнительно можно настроить политику паролей, передав ключ `--policies`. Политика по умолчанию требует присутствия в пароле букв верхнего и нижнего регистров, цифр и специальных символов. Полная команда создания пула пользователей приводится в листинге 6.1.

Другие способы авторизации

Кроме Amazon Cognito, в AWS поддерживаются другие способы защиты вашего API:

- *с использованием ролей и политик IAM* – самый простой механизм авторизации. Чтобы позволить одному API вызывать другой API, необходимо определить политики IAM, разрешающие указанному API вызывать некоторый метод, для которого включена IAM-аутентификация пользователя. Для нас это не самое оптимальное решение, потому что у нас имеется единственный API в Gateway API и несколько маршрутов, которые необходимо защитить;
- *с использованием нестандартного механизма авторизации* – нестандартные механизмы авторизации реализуются в Amazon API Gateway как функции Lambda, которые управляют доступом к методам вашего API с применением ключей аутентификации, таких как OAuth или SAML. При этом при каждом обращении к вашему API в API Gateway будет вызываться ваша функция авторизации. Если функция авторизации подтвердит право доступа, запрос будет передан функции-обработчику Lambda.

Листинг 6.1. Создание пула пользователей

```
aws cognito-idp create-user-pool \
    --pool-name Pizzeria \
    --policies "PasswordPolicy={MinimumLength=8,RequireUppercase=false,
        RequireLowercase=false,
        RequireNumbers=false,RequireSymbols=false}" \
    --username-attributes email \
    --query UserPool.Id \
    --output text
```

Создать пул пользователей.

Имя пула.

Настройка политики паролей.

Использовать адреса электронной почты для идентификации пользователей.

Вывести идентификатор пула в виде текста.

Команда вернет идентификатор пула, потому что в нее был добавлен ключ `--query`. Сохраните этот идентификатор – он еще понадобится нам.

> **ПРИМЕЧАНИЕ.** Для простоты в этом примере используется лишь часть возможностей Cognito. Пулы пользователей поддерживают множество других интересных функций, таких как автоматическая проверка электронной почты или телефона, список обязательных атрибутов. За более полной информацией обращайтесь к официальной документации, доступной по адресу: https://console.aws.amazon.com/cognito/home.

В пуле пользователей должен иметься хотя бы один клиент, чтобы можно было подключить его. Создать клиента можно с помощью команды `aws cognito-idp create-user-pool-client`, как показано в листинге 6.2. Команде нуж-

но передать идентификатор пула, который вернула предыдущая команда, и имя клиента. Мы будем тестировать эту конфигурацию с помощью простого веб-приложения, поэтому создадим клиента без секретного ключа (это означает, что в будущем вам придется создать еще одного клиента для мобильного приложения Пьера).

Листинг 6.2. Создание клиента для пула пользователей

```
aws cognito-idp create-user-pool-client \
  --user-pool-id eu-central-1_userPoolId \
  --client-name PizzeriaClient \
  --no-generate-secret \
  --query UserPoolClient.ClientId \
  --output text
```

← Создать клиента пула пользователей.
← Идентификатор пула, который вернула предыдущая команда.
← Имя клиента.
← Не генерировать секретный ключ.
← Вывести только идентификатор клиента в виде текста.

Эта команда вернет идентификатор клиента; сохраните его – он понадобится вам на следующем шаге.

Перед реализацией аутентификации через Facebook и привилегий внутри приложения посетите портал разработчиков Facebook, чтобы создать приложение и получить его идентификатор.

> **ПРИМЕЧАНИЕ**. Если вы не знаете, как создать приложение в Facebook, загляните в документацию для разработчиков Facebook, где этот процесс описан во всех подробностях: https://developers.facebook.com/docs/apps/register.
>
> Если вы не пользуетесь Facebook и не хотите создавать учетную запись для своего приложения, ваше приложение сможет работать, используя для авторизации электронную почту и пароль. Там, где потребуется внести соответствующие изменения, мы сообщим.

Следующий шаг – создание пула идентификации с помощью команды `aws cognito-identity create-identity-pool` из AWS CLI, как показано в листинге 6.3. В команде нужно указать имя пула идентификации, перечислить всех поддерживаемых провайдеров аутентификации (в вашем случае Facebook) и провайдера идентификации Cognito. В параметре `--cognito-identity-provider` нужно указать имя провайдера и идентификатор клиента, а также необходимость проверки ключей на стороне сервера. Имя провайдера имеет следующий формат: `cognito-idp.<REGION>.amazonaws.com/<USER_POOL_ID>`. Идентификатор клиента – это то, что вернула предыдущая команда. Нам не требуется проверка ключей на стороне сервера, поэтому для этого параметра установлено значение `false`.

Листинг 6.3. Создание пула идентификации

```
aws cognito-identity create-identity-pool \
  --identity-pool-name Pizzeria \
```

← Создать пул идентификации.
← Имя пула.

```
--allow-unauthenticated-identities \
--supported-login-providers graph.facebook.com=266094173886660 \
--cognito-identity-providers ProviderName=cognito-idp.eu-central-1.
  amazonaws.com/
eu-central-1_qpPMn1Tip,ClientId=4q14u0qalmkangdkhieekqbjma,
  ServerSideTokenCheck=false \
--query IdentityPoolId \
--output text
```

Добавить поддерживаемых провайдеров аутентификации – в нашем случае Facebook.

Разрешить неаутентифицированным пользователям доступ к пулу идентификации.

Добавить провайдера идентификации Cognito, указав идентификатор пула пользователей и идентификатор клиента, полученные на предыдущих шагах.

Вывести идентификатор пула в виде текста.

После успешного создания пула идентификации нужно добавить две роли для аутентифицированных и неаутентифицированных пользователей. Если вам понадобится помощь в создании ролей, обращайтесь по адресу: https://aws.amazon.com/blogs/mobile/understanding-amazon-cognito-authentication-part-3-roles-and-policies/.

> **СОВЕТ.** Если у вас возникнут сложности с созданием ролей из интерфейса командной строки AWS CLI, воспользуйтесь веб-консолью, где все то же самое можно сделать одним щелчком мыши. Перейдите в свой пул идентификации, щелкните на кнопке **Edit identity pool** (Изменить пул идентификации), а затем на ссылке **Create New Role** (Создать новую роль) для аутентифицированных и неаутентифицированных ролей.

Для настройки ролей используйте команду `aws cognito-identity set-identity-pool-role`, передав ей идентификатор пула и роли для аутентифицированных и неаутентифицированных пользователей, как показано в листинге 6.4. Не забудьте заменить `<ROLE1_ARN>` и `<ROLE2_ARN>` именами ресурсов Amazon (Amazon Resource Name, ARN) двух ролей, которые вы только что создали.

Листинг 6.4. Добавление ролей в пул идентификации

```
aws cognito-identity set-identity-pool-roles \
  --identity-pool-id eu-central-1:2a3b45c6-1234-123d-1234-1e23fg45hij6 \
  --roles authenticated=<ROLE1_ARN>,unauthenticated=<ROLE2_ARN>
```

Добавить роли для аутентифицированных и неаутентифицированных пользователей.

Идентификатор пула идентификации.

Настройка ролей в пуле идентификации.

В случае успеха эта команда вернет пустой ответ.

6.2.1. Управление доступом к API с помощью Cognito

Теперь, создав пулы пользователей и идентификации, можно добавить поддержку аутентификации в наш код.

Claudia в сочетании с Claudia API Builder поддерживает все три метода авторизации, упомянутых выше: с использованием ролей IAM, нестандартных механизмов авторизации и пулов пользователей Cognito. В этой книге мы рассмотрим только последний вариант, но два других работают аналогично. Более подробную информацию о них ищите в официальной документации для Claudia API Builder: https://github.com/claudiajs/claudia-api-builder/blob/master/docs/api.md#require-authorization.

> **ПРИМЕЧАНИЕ.** Пул идентификации Cognito не используется ни библиотекой Claudia, ни нашей функцией Lambda. Он используется интерфейсными приложениями для получения временного доступа к пулам пользователей Cognito, что избавляет нас от необходимости включать в код ключи доступа к профилю AWS.

Для реализации авторизации с использованием пула пользователей Cognito необходимо зарегистрировать свою функцию авторизации вызовом метода `registerAuthorizer` экземпляра Claudia API Builder. Этот метод принимает два атрибута: имя функции авторизации и объект с массивом ARN пулов пользователей Cognito, например:

```
api.registerAuthorizer('MyCognitoAuth', {
  providerARNs: ['<COGNITO_USER_POOL_ARN>']
});
```

После регистрации функции авторизации добавьте в определение маршрута (как третий аргумент) объект с атрибутом `cognitoAuthorizer`, содержащим имя зарегистрированной функции авторизации. Определение маршрута должно выглядеть так:

```
api.post('/protectedRoute', request => {
  return doSomething(request)
}, { cognitoAuthorizer: 'MyCognitoAuth' })
```

Проделайте то же самое для всех маршрутов в файле `api.js`. После этого маршруты должны выглядеть, как показано в листинге 6.5. Все маршруты, имеющие отношение к заказам, должны быть защищены функцией авторизации Cognito, а маршруты, возвращающие прейскурант, должны оставаться общедоступными.

Листинг 6.5. API с нестандартной функцией авторизации

```
'use strict'

const Api = require('claudia-api-builder')
```

```
const api = new Api()

const getPizzas = require('./handlers/get-pizzas')
const createOrder = require('./handlers/create-order')
const updateOrder = require('./handlers/update-order')
const deleteOrder = require('./handlers/delete-order')

api.registerAuthorizer('userAuthentication', {
  providerARNs: [process.env.userPoolArn]
})

// Определение маршрутов
api.get('/', () => 'Welcome to Pizza API')

api.get('/pizzas', () => {
  return getPizzas()
})
api.get('/pizzas/{id}', (request) => {
  return getPizzas(request.pathParams.id)
}, {
  error: 404
})

api.post('/orders', (request) => {
  return createOrder(request)
}, {
  success: 201,
  error: 400,
  cognitoAuthorizer: 'userAuthentication'
})

api.put('/orders/{id}', (request) => {
  return updateOrder(request.pathParams.id, request.body)
}, {
  error: 400,
  cognitoAuthorizer: 'userAuthentication'
})

api.delete('/orders/{id}', (request) => {
  return deleteOrder(request.pathParams.id)
}, {
  error: 400,
  cognitoAuthorizer: 'userAuthentication'
})
```

Регистрация своей функции авторизации.

Получить ARN пула пользователей из переменной окружения и установить его как ARN провайдера.

Передать объект запроса целиком, включая его тело и данные авторизации.

Добавить авторизацию только в выбранные маршруты.

```
})

api.post('delivery', (request) => {
  return updateDeliveryStatus(request.body)
}, {
  success: 200,
  error: 400,
  cognitoAuthorizer: 'userAuthentication'
})

module.exports = api
```

Добавить авторизацию только в выбранные маршруты.

Последнее, что нужно сделать, – изменить обработчики маршрутов, задействовав в них функцию авторизации.

Например, в обработчике `create-order.js` нужно:

- принять объект запроса целиком, а не только его тело. Нам потребуется прочитать данные о пользователе из пула пользователей Cognito; эта информация хранится в объекте запроса, но за пределами его тела;
- получить данные о пользователе, вызвав функцию авторизации. Они доступны в объекте контекста авторизации, в атрибуте с именем `claims`;
- извлечь адрес пользователя из тела запроса, если указан, иначе извлечь адрес по умолчанию из профиля авторизованного пользователя;
- сохранить имя пользователя из Cognito в таблице заказов.

На рис. 6.3 показано, как осуществляется управление доступом к API с использованием API Gateway и пула пользователей Amazon Cognito.

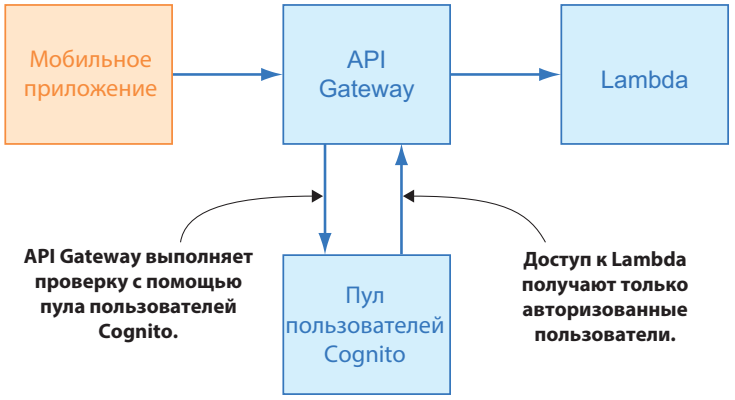


Рис. 6.3. Схема управления доступом к нашему API с использованием API Gateway и пула пользователей Amazon Cognito

Измененный обработчик `create-order.js` представлен в листинге 6.6.

Листинг 6.6. Обработчик create-order.js с поддержкой авторизации

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const rp = require('minimal-request-promise')

function createOrder(request) {
  console.log('Save an order', request.body)
  const userData = request.context.authorizer.claims
  console.log('User data', userData)

  let userAddress = request.body && request.body.address
  if (!userAddress) {
    userAddress = JSON.parse(userData.address).formatted
  }

  if (!request.body || !request.body.pizza || userAddress)
    throw new Error('To order pizza please provide pizza type and address where
pizza should be delivered')

  return rp.post('https://some-like-it-hot.effortless-serverless.com/delivery', {
    headers: {
      Authorization: 'aunt-marias-pizzeria-1234567890',
      'Content-type': 'application/json'
    },
    body: JSON.stringify({
      pickupTime: '15.34pm',
      pickupAddress: 'Aunt Maria Pizzeria',
      deliveryAddress: userAddress,
      webhookUrl: 'https://g8fhlgccof.execute-api.eu-central-1.amazonaws.com/
latest/delivery',
    })
  })
  .then(rawResponse => JSON.parse(rawResponse.body))
  .then(response => {
    return docClient.put({
      TableName: 'pizza-orders',
      Item: {
        cognitoUsername: userAddress['cognito:username'],
        orderId: response.deliveryId,
        pizza: request.body.pizza,
```

Функция createOrder принимает объект запроса целиком.

Получить данные о пользователе из объекта контекста авторизации и вывести их в журнал.

По умолчанию использовать адрес из тела запроса.

Если адрес не указан, использовать адрес из профиля.

Передать адрес службе доставки Some Like It Hot.

Сохранить имя пользователя из Cognito в базе данных.

```
        address: userAddress,
        orderStatus: 'pending'
      }
    }).promise()
  })
  .then(res => {
    console.log('Order is saved!', res)

    return res
  })
  .catch(saveError => {
    console.log(`Oops, order is not saved :(`, saveError)

    throw saveError
  })
}

module.exports = createOrder
```

Сохранить адрес в базе данных.

После изменения кода запустите команду `claudia update`, чтобы развернуть API. Для проверки работы авторизации вам потребуется реализовать процесс входа/регистрации в систему. Реализация авторизации в серверной части не вызвала у нас особых сложностей. Но основная работа, включая интеграцию пулов пользователей и идентификации, должна выполняться на стороне клиента. Обсуждение этой части приложения выходит за рамки книги, но вы можете увидеть рабочий пример с практическим руководством в репозитории GitHub по адресу: https://github.com/effortless-serverless/pizzeria-web-app.

Однако, прежде чем запустить этот код из репозитория, можно выполнить простую проверку невозможности доступа к API для неавторизованного пользователя, выполнив такую команду `curl`:

```
curl -o - -s -w ", status: %{http_code}\n" \
  -H "Content-Type: application/json" \
  -X POST \
  -d '{"pizzaId":1,"address":"221B Baker Street"}' \
  https://21cioselv9.execute-api.us-east-1.amazonaws.com/latest/orders
```

Передать userData как дополнительный аргумент.

Эта команда должна вернуть HTTP-код ошибки 401.

6.3. Опробование!

Теперь, когда вы знаете, как работает авторизация, настал момент опробовать ее.

6.3.1. Упражнение

Ваша задача – изменить обработчик `delete-order.js`, чтобы позволить пользователям удалять только их собственные заказы.

Вот несколько подсказок для тех, кому они понадобятся:

- авторизация уже была добавлена в маршрут в листинге 6.5;
- в настоящее время функция `deleteOrder` принимает только идентификатор заказа `orderId`, поэтому вам нужно добавить прием информации об авторизованном пользователе;
- для проверки принадлежности заказа текущему пользователю используйте в методе `deleteOrder` атрибут `cognito:username` из объекта `request.context.authorizer.claims`;
- если заказ не принадлежит пользователю, вы должны вернуть ошибку.

Возврат нестандартных ошибок из Claudia

Когда обработчик генерирует ошибку, Claudia посылает клиенту код «400 Bad Request», как мы определили раньше. Но в случае, когда пользователь пытается удалить заказ, который ему не принадлежит, можно вернуть HTTP-ошибку «403 Forbidden» или «401 Unauthorized».

Для этого нужно установить код ошибки и ответ динамически. Claudia API Builder позволяет сделать это, предлагая метод ApiResponse в экземпляре API Builder. Например, вот как можно вернуть код 403:

```
return new api.ApiResponse({ message: 'Action is forbidden' },
{ 'Content-Type': 'application/json' }, 403)
```

Дополнительные подробности о динамических ответах можно найти в официальной документации для Claudia API Builder: https://github.com/claudiajs/claudia-api-builder/blob/master/docs/api.md#dynamic-responses.

Если упражнение покажется вам слишком простым, вот вам еще пара упражнений посложнее:

- измените первичный ключ заказа, превратив его в комбинацию из идентификатора заказа и имени пользователя Cognito, создавшего его. При таком подходе появляется возможность находить и удалять заказы, принадлежащие только авторизованному пользователю;
- измените обработчик `update-order.js` так, чтобы пользователи могли изменять только свои заказы.

6.3.2. Решение

Прежде всего нужно изменить обработчик `delete-order.js`, чтобы он принимал идентификатор заказа `orderId` и данные об авторизованном пользователе. Также нужно извлечь заказ из базы данных и проверить его принадлежность авторизованному пользователю. Измененный обработчик показан в листинге 6.7.

Листинг 6.7. Обработчик `delete-order.js` с поддержкой авторизации

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const rp = require('minimal-request-promise')

function deleteOrder(orderId, userData) {
  return docClient.get({
    TableName: 'pizza-orders',
    Key: {
      orderId: orderId
    }
  }).promise()
    .then(result => result.Item)
    .then(item => {
      if (item.cognitoUsername !== userData['cognito:username'])
        throw new Error('Order is not owned by your user')

      if (item.orderStatus !== 'pending')
        throw new Error('Order status is not pending')

      return
        rp.delete(`https://some-like-it-hot.effortless-serverless.com/
delivery/${orderId}`, {
          headers: {
            Authorization: 'aunt-marias-pizzeria-1234567890',
            'Content-type': 'application/json'
          }
        })
    })
    .then(() => {
      return docClient.delete({
        TableName: 'pizza-orders',
        Key: {
```

Принять дополнительный аргумент userData.

Проверить принадлежность заказа авторизованному пользователю.

Сгенерировать ошибку, если заказ не принадлежит авторизованному пользователю.

```
          orderId: orderId
        }
      }).promise()
    })
}

module.exports = deleteOrder
```

После изменения обработчика нужно обновить маршрут и передать правильные данные обработчику. В листинге 6.8 показан фрагмент из файла `api.js`, в котором идентификатор заказа и данные о пользователе передаются в обработчик `delete-order.js`. Как было показано выше, данные доступны в виде атрибута `claims` в объекте `request.context.authorizer`.

Листинг 6.8. Изменение маршрута удаления заказа для передачи в обработчик данных о пользователе

```
api.delete('/orders/{id}', (request) => {
  return deleteOrder(request.pathParams.id, request.context.authorizer.claims)
}, {
  error: 400,
  cognitoAuthorizer: 'userAuthentication'
})
```

Передать в обработчик orderId и claims из объекта authorizer.

После внесения изменений в код просто запустите команду `claudia update`, чтобы развернуть его. По завершении вы можете использовать веб-приложение из репозитория https://github.com/effortless-serverless/pizzeria-web-app для получения ключа авторизации и для тестирования. Попытка удалить старый заказ с помощью этого ключа не увенчается успехом, потому что он был создан без авторизации – имя пользователя Cognito не будет совпадать.

В заключение

- Аутентификацию пользователей в бессерверных приложениях можно организовать с помощью Amazon Cognito.
- Для больших групп пользователей с разными привилегиями используйте пулы идентификации Amazon Cognito.
- В одном приложении с легкостью можно организовать аутентификацию разными способами; просто помните, что для каждого способа нужно создать свой пул пользователей.
- Библиотека Claudia ускоряет настройку аутентификации с AWS Cognito.

Глава 7

Работа с файлами

Эта глава охватывает следующие темы:

- хранение медиафайлов и другого статического содержимого в бессерверных приложениях;
- управление файлами и доступ к ним в бессерверных API;
- обработка статических файлов с использованием бессерверных функций.

Приложениям часто требуется хранить информацию не только в базах данных, но и в статических файлах. Обычно в статических файлах хранятся фотографии, аудио- или видеозаписи, а также простой текст (например, файлы HTML, CSS и JavaScript).

Бессерверным приложениям тоже приходится хранить статические файлы. Однако бессерверный характер таких приложений предполагает необходимость использования решения для хранения данных, основанного на тех же принципах бессерверных вычислений. В этой главе рассматриваются возможности хранения файлов в бессерверном окружении и описываются приемы создания отдельной функции для обработки файлов, которая использует хранилище и возвращает требуемые файлы другой функции Lambda – в данном случае нашему бессерверному API.

7.1. Хранение статических файлов в бессерверных приложениях

Наше приложение для пиццерии тетушки Марии не будет полным без изображений вкусной пиццы. Ваш двоюродный брат Микеланджело (также известный как Майк) уже сделал потрясающие фотографии всех пицц, поэтому нам остается только организовать хранение и обслуживание этих статических файлов. Для этого в AWS есть служба Simple Storage Service (S3), позволяющая хранить файлы до 5 Тбайт в бессерверном окружении.

Amazon S3 хранит файлы в так называемых *корзинах* (buckets) – структурах, напоминающих папки, которые принадлежат учетной записи AWS. Каждый файл или объект в корзине имеет уникальный идентификационный ключ. Корзины S3 поддерживают триггеры для функций Lambda, которые позволяют вызвать определенную функцию, когда что-то происходит в корзине.

> **ПРИМЕЧАНИЕ.** Мы советуем ознакомиться с основами Amazon S3, перед тем как продолжить чтение этой главы. Хорошей отправной точкой вам послужит официальная документация, доступная по адресу http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html.

Основой всего в S3 является корзина, которую можно создать с помощью веб-консоли AWS или интерфейса командной строки AWS CLI – нашего любимого инструмента. Команде `mb` требуется передать аргумент с URI службы S3. S3 URI – это имя вашей корзины в S3 с префиксом `s3://`. Если понадобится, с помощью флага `--region` можно указать регион. В нашем примере мы создадим корзину с именем `aunt-marias-pizzeria` и укажем регион.

Выполните следующую команду в AWS CLI:

```
aws s3 mb s3://aunt-marias-pizzeria --region eu-central-1
```

> **ПРИМЕЧАНИЕ.** Обратите внимание, что имя корзины должно быть уникальным среди всех корзин, существующих в Amazon S3. Ваша команда не будет выполнена, если вы используете имя, которое указано в предыдущем листинге. Для успешного выполнения команды используйте уникальное имя. За дополнительной информацией о соглашениях и правилах именования корзин в S3 обращайтесь по адресу: https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html.

Команда должна вернуть ответ: `make_bucket: aunt-marias-pizzeria`. Если указать неуникальное имя для корзины, она вернет следующую ошибку, и вам придется повторно запустить команду с другим именем:

```
make_bucket failed: s3://bucket-name An error occurred
(BucketAlreadyExists) when calling the CreateBucket operation: The
requested bucket name is not available. The bucket namespace
is shared by all users of the system.

Please select a different name and try again.
```

(Перевод:

```
make_bucket, ошибка: s3://bucket-name При выполнении операции
CreateBucket возникла ошибка (BucketAlreadyExists): указанное
имя корзины уже занято. Все пользователи системы используют
одно и то же пространство имен.

Пожалуйста, выберите другое имя и повторите попытку.)
```

После создания корзины следует определить пользователей, имеющих право выгружать в нее файлы. Но перед этим нужно подумать о структуре папок в корзине.

> **ПРИМЕЧАНИЕ.** Корзины в Amazon S3 на самом деле не поддерживают папок – только объекты. Но чтобы упростить взаимодействие с S3, Amazon отображает в веб-консоли имена объектов, представляющих папки, в виде реальных папок. Например, объект с именем `/images/large/pizza.jpg` будет показан как изображение `pizza.jpg` в папке с именем `large`, которая вложена в папку `images`.

Как показано на рис. 7.1, вы должны выгрузить изображения в папку `images`. Иногда исходные изображения могут оказаться слишком большими для мобильного приложения, поэтому также нужно создать папку `thumbnails`, где будут храниться уменьшенные версии изображений. Кроме того, поскольку в каждый момент времени будет иметься только один файл `menu.pdf`, его необязательно хранить в папке.

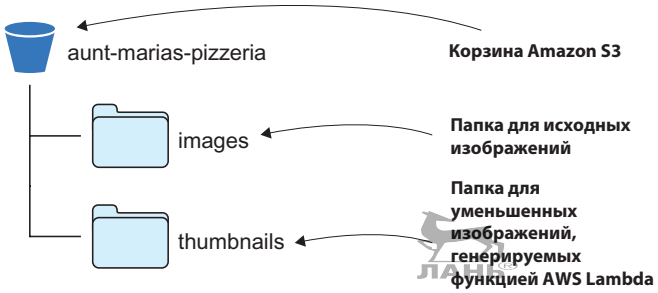


Рис. 7.1. Рекомендованная структура папок в корзине Amazon S3

Далее, определив структуру папок, можно настроить разрешения для определенных пользователей выгружать изображения в корзину. Самый простой способ сделать это – сгенерировать предварительно подписанный URL, который будет использоваться для выгрузки изображений.

По умолчанию все объекты и сегменты являются приватными – доступ к ним имеет только пользователь, создавший их. Предварительно подписанный URL позволяет пользователям, не имеющим права доступа, выгружать файлы в корзину. Такой URL создается пользователем, который имеет доступ к корзине и предоставит временные разрешения всем, кого знает.

Поскольку этот URL должен быть защищен от несанкционированного доступа, мы создадим новый маршрут в Pizza API, который будет генерировать и возвращать нужный URL. Этот маршрут также должен быть защищен; в нашем примере мы разрешим всем авторизованным пользователям использовать эту конечную точку API, но в реальном приложении у вас должна быть определена специальная группа пользователей, которая сможет получить доступ к конкретным конечным точкам API, например группа администраторов.

ПРИМЕЧАНИЕ. Упоминаемые здесь группы пользователей являются группами в пулах пользователей Cognito. Узнать больше о группах в пулах пользователей Cognito можно по адресу: http://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-pools-user-groups.html.

Чтобы сгенерировать URL, нужно создать новый обработчик и использовать в нем метод `getSignedUrl` класса `S3`. Этот метод принимает два аргумента: имя метода, который будет использоваться через подписанный URL (`putObject`), и объект с параметрами. В этом объекте должны быть определены следующие параметры:

- имя корзины, доступ к которой будет осуществляться посредством подписанного URL;
- уникальный ключ для подписи URL. Поскольку сгенерировать уникальный ключ вручную непросто, для этой цели лучше использовать модуль `uuid`. Мы уже использовали этот модуль в главе 3; только не забудьте переустановить его, если вы удалили его из файла `package.json` (Устанавливайте UUID версии 4, прямо указав `uuid/v4`.);
- список управления доступом (Access Control List, ACL), определяющий порядок взаимодействий с объектами в корзине. В нашем случае объекты должны быть доступны всем пользователям, поэтому мы передадим значение `public-read`;
- время действия сгенерированного URL в секундах. Двух минут будет вполне достаточно, поэтому мы передадим в этом параметре 120 секунд.

После создания объекта параметров вызовем метод `getSignedUrl`, чтобы подписать URL, а затем вернем его как объект JSON. Создайте файл с именем `generate-presigned-url.js` в папке `handlers` и скопируйте в него код из листинга 7.1.

Листинг 7.1. Обработчик в Pizza API для создания предварительно подписанного URL

```
'use strict'

const uuidv4 = require('uuid/v4')
const AWS = require('aws-sdk')
const s3 = new AWS.S3()

function generatePresignedUrl() {
  const params = {
    Bucket: process.env.bucketName,
    Key: uuidv4(),
    ACL: 'public-read',
```

Импортировать модуль uuid.

Импортировать AWS SDK и инициализировать класс S3.

Определить функцию-обработчик.

Получить имя корзины из переменной окружения bucketName.

Создать уникальный идентификатор.

Объявить объект доступным для чтения всем пользователям.

```
    Expires: 120
  }

  s3.getSignedUrl('putObject', params).promise()
    .then(url => {
      return {
        url: url
      }
    })
}

module.exports = generatePresignedUrl
```

Определить время действия URL в секундах.

Получить подписанный URL для метода putObject.

Вернуть объект JSON и подписанный URL.

Теперь добавим в файл `api.js` новый маршрут к этому обработчику. Назовем его `/upload-url`. Как отмечалось выше, мы должны защитить этот маршрут, как уже защитили маршруты `/orders`, чтобы только пользователи, авторизованные с помощью функции `userAuthentication`, могли получить этот URL. В листинге 7.2 показан конец файла `api.js`. Остальной код в нем остался без изменений. Кроме того, не забудьте импортировать обработчик `getSignedUrl` в начале файла `api.js`, добавив строку `const getSignedUrl = require('./handlers/generate-presigned-url.js')`.

Листинг 7.2. Новые маршруты `/delivery` и `/upload-url` в файле `api.js`

```
api.post('delivery', (request) => {
  return updateDeliveryStatus(request.body)
}, {
  success: 200,
  error: 400
}, {
  cognitoAuthorizer: 'userAuthentication'
})

api.get('upload-url', (request) => {
  return getSignedUrl()
},
{ error: 400 },
{ cognitoAuthorizer: 'userAuthentication' })

module.exports = api
```

Добавить новый маршрут GET.

Вызвать обработчик getSignedURL.

В случае ошибки вернуть HTTP-код 400.

Потребовать авторизацию для этого нового маршрута.

Если теперь обновить API командой `claudia update` и затем послать запрос по новому маршруту с ключом авторизации (полученным из веб-приложения, как описано в предыдущей главе), в ответ будет возвращен подписанный URL, который можно использовать для выгрузки файлов в корзину.

7.2. Создание миниатюр

Поскольку выгруженные изображения могут оказаться слишком большими для мобильного приложения, также имеющегося у тетушки Марии, мы должны предусмотреть создание уменьшенных копий всех фотографий-миниатюр. Создание миниатюр не должно каким-либо образом блокировать наш API, поэтому для обработки изображений лучше создать независимый микросервис.

Независимая служба в данном случае представляет собой отдельную функцию Lambda, которая автоматически запускается после выгрузки новой фотографии в Amazon S3. При этом имеет место следующая последовательность событий (рис. 7.2):

- обращением к маршруту `/upload-url` пользователь запрашивает новый подписанный URL;
- выгружает новую фотографию в этот URL;
- Amazon S3 запускает нашу новую функцию Lambda;
- функция Lambda изменяет размер изображения и сохраняет миниатюру в папке `thumbnails`.

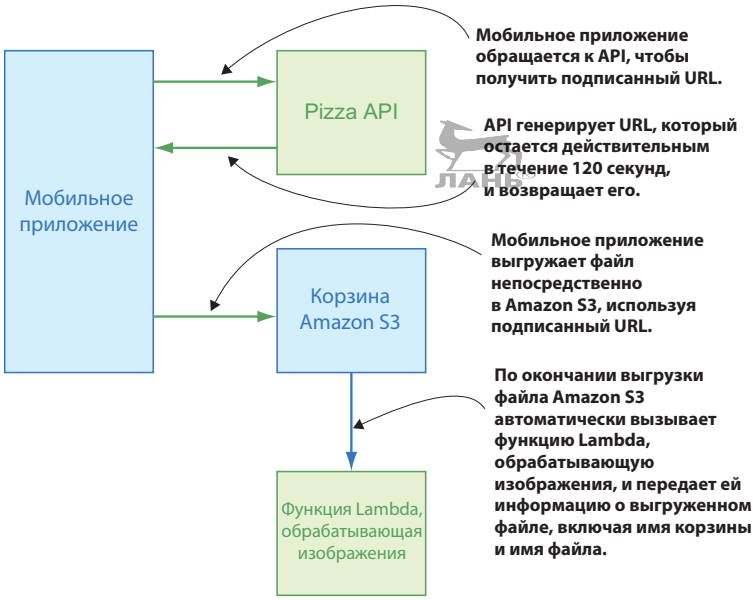


Рис. 7.2. Процесс выгрузки и обработки изображений

Новая функция Lambda не будет запускаться HTTP-запросами, поэтому нам не потребуется использовать Claudia API Builder. Вместо этого она будет получать новый объект из S3 и изменять его размер с помощью ImageMagick. Пакет ImageMagick доступен в AWS Lambda по умолчанию; вам не нужно устанавливать его перед использованием.

ImageMagick

ImageMagick – это бесплатный программный пакет с открытым исходным кодом для отображения, преобразования и редактирования файлов растровых и векторных изображений. Он включает несколько интерфейсов командной строки и может читать и записывать изображения в более чем 200 различных форматах.

ImageMagick может преобразовывать файлы из одного формата в другой, масштабировать и трансформировать изображения, манипулировать цветами, композицией и многими другими аспектами.

Узнать больше об ImageMagick можно на сайте http://imagemagick.org.

Первый шаг на пути к созданию отдельной службы – создание нового проекта. Вы должны:

1) создать новую папку за пределами папки `pizza-api` (мы выбрали имя `pizza-image-processor`);
2) внутри папки инициализировать новый пакет NPM (командой `npm init`).

Следующий шаг – создание файла, экспортирующего функцию-обработчик. Поскольку это всего лишь обработчик изображений, а не API, доступный извне, в нем не нужно использовать Claudia API Builder.

ПРИМЕЧАНИЕ. Когда Claudia API Builder не используется, вы не сможете экспортировать функцию-обработчик с помощью `module.exports`. Вместо этого экспортирование следует выполнять с помощью `export.handler`.

Эта служба будет небольшой и вполне уместится в одном файле, но для простоты обслуживания и удобства тестирования разделим ее на два файла: первый – файл с исходным кодом, который просто извлекает данные из события Lambda, а второй – фактическая реализация преобразования.

В первом файле определим функцию-обработчик, принимающую три аргумента:

- событие, вызвавшее запуск функции Lambda;
- контекст функции Lambda;
- функцию обратного вызова для отправки ответа.

В первом файле сначала проверим существование действительной записи о событии и поступило ли оно из Amazon S3. Поскольку одну и ту же функцию Lambda могут запускать разные службы, необходимо также проверить, исходит ли событие от хранилища S3. Затем нужно извлечь имя корзины S3 и имя файла с путем или ключом объекта, выполнив соответствующий запрос к S3.

Ответом на запрос будет изображение, которое нужно передать в функцию `convert`.

> **ПРИМЕЧАНИЕ.** Реализация функции преобразования `convert` основана на использовании `Promise`, как того требует Claudia API Builder. Мы должны придерживаться общего стиля программирования во всех своих службах, но если вы предпочитаете обратные вызовы, можете использовать их.

Код в первом файле показан в листинге 7.3.

Листинг 7.3. Код функции Lambda, реализующий службу обработки изображений

```
'use strict'

const convert = require('./convert')

function handlerFunction(event, context, callback) {
  const eventRecord = event.Records && event.Records[0]

  if (eventRecord) {
    if (eventRecord.eventSource === 'aws:s3' && eventRecord.s3) {
      return convert(eventRecord.s3.bucket.name, eventRecord.s3.object.key)
        .then(response => {
          callback(null, response)
        })
        .catch(callback)
    }

    return callback('unsupported event source')
  }
  callback('no records in the event')
}

exports.handler = handlerFunction
```

Импортировать функцию convert из второго файла.

Определить функцию-обработчик, принимающую событие, контекст Lambda и функцию обратного вызова.

Получить запись о событии в отдельную переменную.

Также проверить, что событие исходит из S3, и преобразовать файл.

Проверить существование записи.

В случае успешного преобразования вернуть признак успеха с помощью функции обратного вызова.

Иначе вернуть признак ошибки.

Вернуть ошибку, если событие исходит не из S3.

Также вернуть ошибку, если запись о событии отсутствует.

Экспортировать функцию-обработчик.

Теперь займемся созданием функции `convert`. Так как служба невелика, нет необходимости усложнять структуру папок: поместим файл `convert.js` в основную папку проекта. На рис. 7.3 изображена следующая последовательность операций, выполняемых функцией `convert`:

- S3 вызывает функцию AWS Lambda в первом файле, которая, в свою очередь, вызывает функцию `convert`;
- функция `convert` загружает изображение из S3 и сохраняет его локально, в папке `/tmp`;

- изображение преобразуется с помощью команды `convert` из пакета ImageMagick, и полученная в результате миниатюра сохраняется в папке `/tmp`;
- затем функция `convert` выгружает новую миниатюру в корзину S3
- и объявляет объект `Promise` вычисленным, сообщая тем самым первому файлу, что операция успешно выполнена.

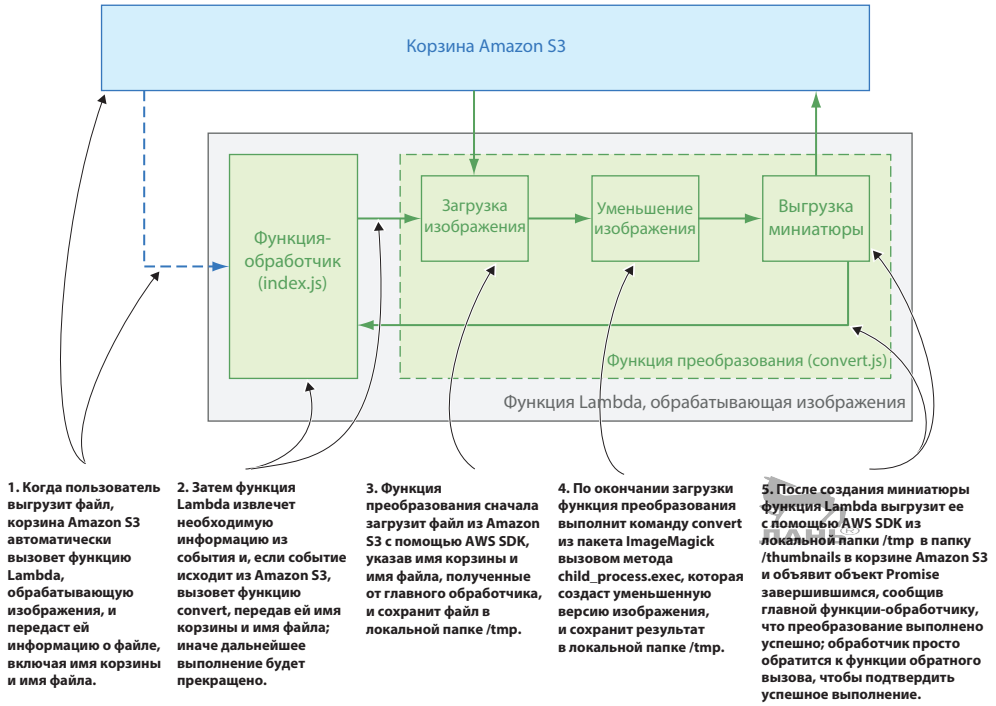


Рис. 7.3. Алгоритм работы функции convert

Как показано на рис. 7.3, функция `convert` должна сначала загрузить файл из S3, вызвав метод `getObject` класса `S3`. Этот метод принимает имя корзины и путь к файлу в S3 и возвращает объект `Promise`, который после завершения вычислений вернет ответ с телом файла в буфере.

Файл `convert.js` должен экспортировать функцию `convert`, которая является обычной функцией Node.js, принимающей имя корзины и путь к файлу в S3 и возвращающей объект `Promise`. Для решения задач, стоящих перед функцией `convert`, необходимо импортировать три модуля Node.js:

- `fs` – для работы с файловой системой;
- `path` – для выполнения операций с путями к файлам;
- `child_process` – для вызова команд ImageMagick.

В дополнение к этим трем модулям также необходимо установить два дополнительных пакета из NPM: `mime` (который определяет тип MIME

загруженного файла) и `aws-sdk` (AWS SDK требуется для программного доступа к службе S3).

Следующий шаг – сохранение загруженного файла. Внутри функции Lambda только папка `/tmp` доступна для записи. Поэтому мы должны создать две папки внутри `/tmp`: одну с именем `images` для загруженного изображения и другую с именем `thumbnails` для миниатюры.

> **ПРИМЕЧАНИЕ.** Перед созданием этих папок проверьте – возможно, они уже существуют. AWS может вызвать функцию Lambda не в первый раз, и папки могут быть уже созданы.

Убедившись в наличии папки `images` внутри `/tmp`, используем команду `fs.writeFile` с именем загруженного файла, чтобы сохранить его в этой папке. Этот метод действует асинхронно, но он не возвращает объект `Promise`, поэтому мы должны заключить его вызов в `Promise`.

Затем, когда файл будет сохранен в локальной папке, можно создать из него миниатюру с помощью ImageMagick. Для этого нужно выполнить команду `convert`, которая позволяет изменять размеры и преобразовывать файлы изображений. Мы не будем менять формат файла, а просто изменим размер изображения. Для этого вызовем команду `convert` со следующими аргументами:

- путь к исходному файлу;
- флаг `-resize`, определяющий операцию изменения размера;
- значение `120x120\>`, определяющее размеры, до которых должно быть уменьшено изображение. Обратите внимание на символы `\>`, следующие за значением: они сообщают, что размер следует изменить, только если исходное изображение имеет размеры больше указанных;
- путь для сохранения миниатюры.

Ниже приводится полная команда для создания миниатюры изображения с именем `image.png`, имеющей размеры 120×120 пикселей:

```
convert /tmp/images/image.png -resize 120x120\> /tmp/thumbnails/image.png
```

Чтобы выполнить команду в функции Lambda, нужно использовать метод `exec` из модуля `child_process`, который мы импортировали в начале файла. Метод `exec` действует асинхронно, но не возвращает объект `Promise`, поэтому мы должны заключить этот вызов в `Promise`.

В заключение функция `convert` должна выгрузить файл в корзину Amazon S3. Сделать это можно с помощью метода `putObject` класса `S3`. Этот метод возвращает `Promise` и принимает следующие аргументы:

- объект с параметрами, содержащий имя корзины;
- путь к файлу в S3;
- тело файла в буфере;

- ACL;
- тип содержимого файла.

Так как служба обработки изображений может работать с разными типами файлов, нам потребуется пакет `mime`, чтобы получить тип MIME исходного изображения и передать его как тип содержимого миниатюры. Если этого не сделать, S3 будет считать, что содержимое файла имеет тип `binary/octet-stream`.

exec и spawn

Модуль `child_process` в Node.js предлагает два метода для выполнения внешних команд: `exec` и `spawn`. Оба могут выполнять одну и ту же работу, но делают это немного по-разному.

Метод `spawn` возвращает объект, который содержит потоки `stdout` и `stderr`. Этот метод больше подходит для команд, которые принимают или возвращают больший объем информации.

Метод `exec` принимает функцию обратного вызова, которая вызывается после завершения команды. Этот обратный вызов возвращает ошибку, если она возникла, а также вывод `stdout` и `stderr`. По умолчанию `exec` ограничивает размер вывода 200 Кбайт, поэтому он больше подходит для команд, которые не возвращают много данных и для которых важнее окончательные, а не промежуточные результаты.

За дополнительной информацией по обеим командам обращайтесь по адресу: https://nodejs.org/api/child_process.html#child_process_asynchronous_process_creation.

В листинге 7.4 приводится полный код для файла `convert.js`.

Листинг 7.4. Преобразование изображений в миниатюры

```
'use strict'

const fs = require('fs')
const path = require('path')
const exec = require('child_process').exec
const mime = require('mime')
const aws = require('aws-sdk')
const s3 = new aws.S3()

function convert(bucket, filePath) {
  const fileName = path.basename(filePath)

  return s3.getObject({
```

Определить функцию-обработчик, принимающую имя корзины и путь к файлу в S3.

```
  Bucket: bucket,
  Key: filePath
}).promise()
  .then(response => {
    return new Promise((resolve, reject) => {
      if (!fs.existsSync('/tmp/images/'))
        fs.mkdirSync('/tmp/images/')

      if (!fs.existsSync('/tmp/thumbnails/'))
        fs.mkdirSync('/tmp/thumbnails/')

      const localFilePath = path.join('/tmp/images/', fileName)

      fs.writeFile(localFilePath, response.Body, (err, fileName) => {
        if (err)
          return reject(err)

        resolve(filePath)
      })
    })
  })
  .then(filePath => {
    return new Promise((resolve, reject) => {
      const localFilePath = path.join('/tmp/images/', fileName)
      const localThumbnailPath = path.join('/tmp/thumbnails/', fileName)

      exec(`convert ${localFilePath} -resize 120x120\\>
 ${localThumbnailPath}`, (err, stdout, stderr) => {
        if (err)
          return reject(err)

        resolve(fileName)
      })
    })
  })
  .then(fileName => {
    const localThumbnailPath = path.join('/tmp/thumbnails/', fileName)

    return s3.putObject({
      Bucket: bucket,
      Key: `thumbnails/${fileName}`,
      Body: fs.readFileSync(localThumbnailPath),
      ContentType: mime.getType(localThumbnailPath),
```

Заключить вызовы асинхронных функций в JavaScript-объект Promise.

Создать папки images и thumbnails внутри папки /tmp, если они отсутствуют.

Сохранить файл, полученный из S3, в локальной папке.

Заключить вызовы асинхронных функций в JavaScript-объект Promise.

Уменьшить изображение с помощью ImageMagick.

Отправить объект обратно в S3.

Прочитать содержимое файла из папки /tmp.

Получить тип MIME-файла.

```
        ACL: 'public-read'                    ← Назначить разрешения
      }).promise()                              для миниатюры.
    })
}

module.exports = convert
```

7.2.1. Развертывание функции обработки файлов в S3

Теперь, после реализации службы, ее нужно развернуть с помощью Claudia. Интересно отметить, что в этом случае у нас нет API. Точно так же, как в главе 2, мы используем команду `claudia create` с флагом `--region`, но вместо флага `--api-module`, определяющего модуль для обработки запросов, используем флаг `--handler`. Полная команда приводится в листинге 7.5. С помощью флага `--handler` передается путь к обработчику с суффиксом `.handler`. Например, если обработчик экспортировался из файла `index.js`, путь будет иметь вид: `index.handler`; если обработчик экспортировался из файла `lambda.js`, следует указать `lambda.handler`.

> **ПРИМЕЧАНИЕ.** Формат `<имя_файла>.handler` параметра для флага `--handler` должен строго соблюдаться, потому что, к сожалению, флаг `--handler` не поддерживает никаких других форматов.
>
> Если вы инициализируете свойство `exports.somethingElse` или `module.exports` в главном файле и затем выполните команду с флагом `--handler index` или `--handler index.default`, команда потерпит неудачу, потому что главный файл должен экспортировать свойство `handler`. Поэтому флаг `--handler` работает только со свойством `exports.handler`.

Листинг 7.5. Развертывание службы обработки изображений с помощью Claudia

```
claudia create \              ← Создать новую функцию.
  --region eu-central-1 \     ← Выбор региона.
  --handler index.handler     ← Путь к функции-обработчику.
```

Эта команда вернет информацию о функции Lambda, как показано в листинге 7.6, и создаст файл `claudia.json` в корневом каталоге проекта.

Листинг 7.6. Ответ команды claudia create, завершившейся успехом

```
execution
{
  "lambda": {
```

```
    "role": "pizza-image-processor-executor",
    "name": "pizza-image-processor",
    "region": "eu-central-1"
  }
}
```

Имя роли для функции, созданной библиотекой Claudia.

Имя функции Lambda.

Регион, где развернута функция Lambda.

Прежде чем опробовать нашу новую службу, мы должны сделать еще один шаг – настроить триггер для вызова функции из корзины S3. Для этого в библиотеке Claudia есть команда `claudia add-s3-event-source`. Она принимает несколько параметров, но мы будем использовать только два из них:

- `--bucket` – обязательный флаг, определяющий имя корзины;
- `--prefix` – дополнительный флаг, позволяющий указать папку.

> **ПРИМЕЧАНИЕ.** Полный список параметров команды можно найти по адресу: https://github.com/claudiajs/claudia/blob/master/docs/add-s3-event-source.md.

Как показано в листинге 7.7, мы должны указать также префикс `images/`, потому что тогда команда настроит триггер, реагирующий только на события в папке `images`.

Листинг 7.7. Добавление триггера S3 для вызова функции Lambda

```
claudia add-s3-event-source \
  --bucket aunt-marias-pizzeria \
  --prefix images/
```

Имя корзины.

Добавить триггер S3 для передачи событий в функцию.

Префикс – имя папки в корзине S3, для которого генерируются события.

В случае успеха команда вернет пустой объект, иначе – сообщение об ошибке.

Самый простой способ проверить работоспособность нашей новой службы – вручную выгрузить файл в папку `images` в корзине S3. Попробуйте сделать это, затем подождите несколько секунд и проверьте папку `thumbnails` в корзине S3.

Чтобы опробовать всю цепочку, включающую API пиццерии и обработчик изображений, можно использовать веб-приложение по адресу: https://github.com/effortless-serverless/pizzeria-web-app.

7.3. Опробование!

Это была довольно простая глава, но, так как это конец первой части этой книги, мы решили усложнить упражнение.

7.3.1. Упражнение

Некоторые изображения, подготовленные Микеланджело, имеют очень большие размеры – до 10 мегапикселей и даже больше. Чтобы файлы больших размеров не замедляли их загрузку в мобильном и веб-приложении, мы должны уменьшить изображения, которые имеют высоту или ширину больше 1024 пикселей.

Вот несколько советов:

- повторно используйте функцию `convert` для изменения размеров файла;
- будьте осторожны, потому что вы изменяете файлы, которые используются для создания миниатюр, – возможно, не стоит выполнять обе операции параллельно;
- выгрузите оба файла в S3.

7.3.2. Решение

Как показано в листинге 7.8, большая часть кода осталась прежней – вам точно так же нужно загрузить изображение из S3 и сохранить его в папке `/tmp`, затем сгенерировать миниатюру и, наконец, выгрузить ее в S3.

Но есть некоторые отличительные особенности. После загрузки изображения из S3 и сохранения его в локальной файловой системе вам нужно изменить его размер перед созданием миниатюры. Технически можно изменить размер после создания миниатюры, но лучше сделать наоборот – создать миниатюру из уменьшенного изображения.

После уменьшения размеров изображения и создания миниатюры выгрузите оба файла в Amazon S3. Выгружать файлы можно одновременно, поэтому используйте `Promise.all` для распараллеливания процесса выгрузки.

В листинге 7.8 приводится полный код примера. Выполните команду `claudia update` и попробуйте вручную выгрузить большое изображение в корзину, чтобы протестировать решение.

> **СОВЕТ.** На этот раз команда `claudia update` может потребовать больше времени на выполнение, что обусловлено большим размером пакета `aws-sdk`. Он доступен в AWS Lambda по умолчанию, поэтому для ускорения развертывания его можно объявить необязательной зависимостью и выполнить команду `claudia update` с флагом `--no-optional-dependencies`. В этом случае все необязательные зависимости будут удалены из zip-файла, предназначенного для развертывания вашей функции Lambda.

Листинг 7.8. Преобразование изображений в миниатюры с предварительным уменьшением слишком больших изображений

```
'use strict'

const fs = require('fs')
```

```
const path = require('path')
const exec = require('child_process').exec
const mime = require('mime')

const aws = require('aws-sdk')
const s3 = new aws.S3()

function convert(bucket, filePath) {
  const fileName = path.basename(filePath)

  return s3.getObject({
    Bucket: bucket,
    Key: filePath
  }).promise()
    .then(response => {
      return new Promise((resolve, reject) => {
        if (!fs.existsSync('/tmp/images/'))
          fs.mkdirSync('/tmp/images/')

        if (!fs.existsSync('/tmp/thumbnails/'))
          fs.mkdirSync('/tmp/thumbnails/')

        const localFilePath = path.join('/tmp/images/', fileName)

        fs.writeFile(localFilePath, response.Body, (err, fileName) => {
          if (err)
            return reject(err)

          resolve(filePath)
        })
      })
    })
    .then(filePath => {
      return new Promise((resolve, reject) => {
        const localFilePath = path.join('/tmp/images/', fileName)

        exec(`convert ${localFilePath} -resize 1024x1024\\>
  ${localFilePath}`, (err, stdout, stderr) => {
          if (err)
            return reject(err)

          resolve(fileName)
```

← Загрузить файл из S3.

Сохранить файл в локальном каталоге /tmp.

Уменьшить исходное изображение.

Заключить команду convert в JavaScript-объект Promise.

Выполнить команду convert.

```
      })
    })
  })
  .then(filePath => {
    return new Promise((resolve, reject) => {
      const localFilePath = path.join('/tmp/images/', fileName)
      const localThumbnailPath = path.join('/tmp/thumbnails/', fileName)

      exec(`convert ${localFilePath} -resize 120x120\\>
${localThumbnailPath}`, (err, stdout, stderr) => {
        if (err)
          return reject(err)

        resolve(fileName)
      })
    })
  })
  .then(fileName => {
    const localThumbnailPath = path.join('/tmp/thumbnails/', fileName)
    const localImagePath = path.join('/tmp/images/', fileName)

    return Promise.all([
      s3.putObject({
        Bucket: bucket,
        Key: `thumbnails/${fileName}`,
        Body: fs.readFileSync(localThumbnailPath),
        ContentType: mime.getType(localThumbnailPath),
        ACL: 'public-read'
      }).promise(),
      s3.putObject({
        Bucket: bucket,
        Key: `images/${fileName}`,
        Body: fs.readFileSync(localImagePath),
        ContentType: mime.getType(localImagePath),
        ACL: 'public-read'
      }).promise()
    ])
  })
}

module.exports = convert
```

Сгенерировать миниатюру и сохранить в локальной файловой системе.

Выгрузить файл в S3.

Вернуть Promise.all, который выгрузит в S3 оба файла.

Выгрузить миниатюру.

Выгрузить изображение.

7.4. Конец первой части: специальное упражнение

Вы подошли к концу первой части книги. Вы познакомились с основами создания бессерверных API, и теперь пришло время проверить ваши знания. Каждая часть книги заканчивается специальным упражнением, в котором вы будете проверять знания и навыки, полученные в этой части. Каждое специальное упражнение требует вспомнить все, о чем рассказывалось, и содержит усложненное задание для тех, кому будет интересно заняться их решением.

Чтобы выполнить специальное упражнение, предлагаемое далее, вам потребуется применить все знания, полученные в этой части. Итак, ваша цель: создать в DynamoDB новую таблицу `pizzas`, которая будет хранить прейскурант, добавить в нее статический список пицц, а затем реализовать новый API-вызов, использующий более удачный алгоритм обработки изображений. В отличие от реализации, представленной выше в этой главе, новый API-вызов должен сохранить выгруженное изображение пиццы в S3, а затем записать сгенерированный URL-адрес в базу данных DynamoDB, в дополнительный столбец в таблице `pizzas`. Это означает, что каждой пицце будет соответствовать свой URL-адрес изображения.

> **ПРИМЕЧАНИЕ.** Для специальных упражнений мы не даем подсказок и оставляем вам возможность самим проверить ваше решение.

7.4.1. Усложненное задание

Если предыдущее задание показалось слишком простым, попробуйте решить следующую, более сложную задачу, которая часто встречается во многих приложениях: расширьте объект, представляющий пиццу, чтобы можно было связать с ним несколько изображений, а также выбрать одно из них как изображение по умолчанию.

> **ПРИМЕЧАНИЕ.** Усложненные задания всегда описываются довольно кратко.

В заключение

- Бессерверные приложения необязательно должны использовать бессерверное хранилище, но всегда должны быть полностью бессерверными.
- При использовании AWS вам пригодится служба бессерверного хранилища S3.
- Всегда старайтесь разделить свое бессерверное приложение на небольшие микросервисы. Например, для обработки изображений всегда используйте отдельную бессерверную функцию.

- Claudia.js поможет вам связать вашу функцию Lambda с событиями, происходящими в S3.
- Внутри бессерверной функции можно использовать ImageMagick для обработки изображений и сохранять результаты в S3.

Часть II

Поболтаем

Теперь, когда тетушка Мария получила действующее приложение, пришло время приблизить пиццерию к молодому поколению, добавив поддержку чат-ботов и голосовых помощников. Зачем запускать приложение, когда можно просто попросить Алексу[1] заказать пиццу для вас?!

Для начала мы создадим простой чат-бот для Facebook (глава 8), свяжем его с текущей базой данных и службой доставки (глава 9). Затем мы создадим SMS-чат-бот для клиентов, плохо разбирающихся в современных технологиях, таких как дядюшка Фрэнк, чтобы они могли заказать пиццу, отправив простое SMS-сообщение (глава 10). Наконец, учитывая, что племянница Джулия подарила тетушке Марии на Рождество устройство Amazon Echo Dot, мы попробуем подключиться к голосовому помощнику Алекса, чтобы дать клиентам возможность заказывать пиццу с помощью голосовых команд.

[1] Алекса (Alexa) – голосовой помощник, реализованный и поддерживаемый компанией Amazon. – *Прим. перев.*

Глава 8

Заказ пиццы одним сообщением: чат-боты

Эта глава охватывает следующие темы:

- создание бессерверного чат-бота;
- как работают бессерверные чат-боты и как Claudia Bot Builder помогает их создавать;
- использование сторонней платформы для чат-ботов (Facebook Messenger).

Бессерверные приложения не всегда реализуют какой-то API или простые микросервисы обработки данных. Программное обеспечение развивается, и люди находят разные, иногда необычные способы его использования. Мы прошли долгий путь от изолированных настольных приложений до веб-сайтов и мобильных приложений и в последнее время наблюдаем повсеместное распространение чат-ботов и голосовых помощников.

В этой главе мы покажем, как еще больше сократить дистанцию между вами и вашими пользователями, создав бессерверный чат-бот в Facebook Messenger и интегрировав его с Pizza API. Вы также узнаете, как работают чат-боты и как легко они реализуются в бессерверном окружении с помощью Claudia.

8.1. Заказ пиццы без браузера

Пока мы работали над Pizza API для тетушки Марии, ее племянница Джулия несколько раз заходила в пиццерию, чтобы поздороваться. Джулия учится в средней школе и, конечно же, много времени проводит в своем телефоне. Она рада, что вы помогли пиццерии создать службу онлайн-заказов, но отметила, что она недостаточно крутая. Мы отстаем от основного конкурента тетушки Марии – пиццерии Chess, которая имеет чат-бота в Facebook Messenger, помогающего клиентам заказывать пиццу, не покидая Facebook. Так как од-

ноклассники Джулии много общаются в Facebook Messenger, они постоянно пользуются этой возможностью. Наличие чат-бота определенно поможет тетушке Марии привлечь больше молодых клиентов, поэтому она спрашивает, сможем ли мы помочь ей с этим.

ЧТО ТАКОЕ ЧАТ-БОТ? Чат-бот – это компьютерная программа, имитирующая осмысленный диалог с одним или несколькими людьми с помощью текстовых или аудиометодов.

Краткая история чат-ботов

Для многих термин «чат-бот» звучит как что-то новое, хотя сама идея вовсе не нова. Чат-боты появились в середине XX века в результате попыток организовать более удобный способ взаимодействия человека с компьютером.

Интерфейс чат-бота упоминается в известном тесте Тьюринга 1950 года. Затем, в 1966 году, появилась программа ELIZA, имитирующая роджерианского психотерапевта, которая стала первым примером обработки естественного языка. Потом, в 1972 году, появилась программа PARRY, имитирующая человека с параноидальной шизофренией (и да, была устроена встреча PARRY и ELIZA – см. https://tools.ietf.org/html/rfc439).

В 1983 году программой Racter была сгенерирована книга под названием «The Policeman's Beard Is Half Constructed» («Борода полицейского наполовину сконструирована»), которая воспроизводила случайную англоязычную прозу. Позже Racter была выпущена как чат-бот.

Одним из самых известных чат-ботов была Алиса (также известная как A.L.I.C.E., сокращенно от Artificial Linguistic Internet Computer Entity – искусственное лингвистическое интернет-компьютерное существо), выпущенная в 1995 году. Она не смогла пройти тест Тьюринга, но трижды выиграла конкурс Лебнера. Конкурс Лебнера – это ежегодный конкурс по искусственному интеллекту, по итогам которого призами награждаются компьютерные программы, наиболее похожие на человека. В 2005 и 2006 годах тот же конкурс выиграли два персонажа-бота Jabberwocky.

Выход платформы Slackbot в 2014 году снова сделал чат-боты популярными. В 2015 году поддержку чат-ботов реализовали Telegram, а затем Facebook Messenger. В 2016 году то же самое сделала Skype. Apple и некоторые другие компании тоже объявили о создании своих платформ для чат-ботов.

Создание своего чат-бота в наши дни часто является маркетинговой инициативой с целью привлечь больше потенциальных клиентов, пользующихся основными социальными платформами, не требуя от них посещения другого веб-сайта, установки мобильного приложения или настольной программы.

8.2. Привет из Facebook Messenger

Поскольку мы решили создать чат-бота для Pizza API в Facebook Messenger, нам важно понять, как пользователи будут отправлять сообщения этому чат-боту и как работает чат-бот в Facebook.

Чат-боты Facebook являются приложениями поддержки страниц Facebook, то есть они не являются отдельными и независимыми приложениями, подобно играм. Чат-боты для Facebook создаются в четыре этапа:

1) подготовка страницы Facebook для будущего чат-бота;
2) создание приложения Facebook, которое будет обслуживать ваш чат-бот и подключать к вашей странице;
3) реализация и развертывание чат-бота.
4) подключение чат-бота к приложению Facebook.

Чтобы начать взаимодействовать с чат-ботом, пользователь должен открыть вашу страницу в Facebook и отправить сообщение. Приложение, связанное со страницей в Facebook, получит сообщение и отправит запрос чат-боту с сообщением пользователя. Далее чат-бот получит и обработает сообщение и вернет ответ приложению Facebook, которое, в свою очередь, выведет его на вашей странице в Facebook.

Этот процесс изображен на рис. 8.1.

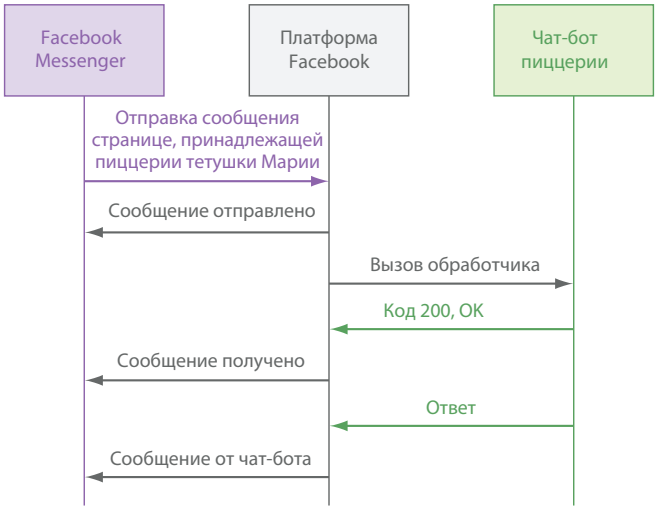


Рис. 8.1. Процесс отправки сообщения чат-боту и получения ответа

Пока вы еще не начали глубоко задумываться о том, как реализовать свой чат-бот, мы хотим сообщить вам приятную новость. Помимо API Builder, в библиотеке Claudia имеется также модуль Bot Builder.

Библиотека Claudia Bot Builder – это тонкая обертка вокруг Claudia API Builder. Она абстрагирует различные API платформы обмена сообщениями и предоставляет простой и универсальный интерфейс для создания чат-ботов.

Цель Claudia Bot Builder – помочь в создании чат-ботов для различных платформ обмена сообщениями, таких как Facebook, Slack, Viber и многих других.

Для начала создадим новую папку на том же уровне, где находятся папки `pizza-api` и `pizza-image-processing`. Мы выбрали имя `pizza-fb-chatbot`. Создав папку, откройте ее и создайте новый проект NPM. Затем установите Claudia Bot Builder как зависимость проекта (см. приложение А).

После установки Bot Builder необходимо создать страницу Facebook с приложением и связать их. О том, как это сделать, рассказывается в приложении В.

Теперь, когда у нас есть готовый проект, можно приступать к фактической реализации. Создайте файл `bot.js` для исходного кода чат-бота в корневой папке `pizza-fb-bot` и откройте его в текстовом редакторе.

В начале файла `bot.js` импортируйте библиотеку Claudia Bot Builder.

> **СОВЕТ.** В отличие от Claudia API Builder, Bot Builder – это модуль, а не класс, поэтому вам не нужно создавать его экземпляр.

Модуль Bot Builder – это функция, которая в первом аргументе принимает функцию-обработчик сообщений и возвращает экземпляр Claudia API Builder. Функция-обработчик сообщений – это функция, которая должна вызываться при получении сообщения чат-ботом. Самый простой способ вернуть ответ из чат-бота – вернуть текстовое сообщение. Claudia Bot Builder отформатирует это текстовое сообщение в соответствии с шаблоном для платформы – в данном случае для Facebook Messenger.

Так как Claudia Bot Builder возвращает экземпляр Claudia API Builder, мы также должны экспортировать экземпляр API Builder, возвращаемый функцией Bot Builder.

В листинге 8.1 приводится содержимое файла `bot.js`.

Листинг 8.1. Простой чат-бот, возвращающий текст приветствия

```
'use strict'

const botBuilder = require('claudia-bot-builder')

const api = botBuilder(() => {
  return `Hello from Aunt Maria's pizzeria!`
})

module.exports = api
```

Импортировать модуль Claudia Bot Builder.

Определение функции-обработчика сообщений для Claudia Bot Builder и сохранение экземпляра Claudia API Builder.

Вернуть простой текстовый ответ.

Развертывание чат-бота выполняется подобно развертыванию Pizza API. Как показано в листинге 8.2, нужно запустить команду `claudia create`, указав регион и флаг `--api-module`, определяющий путь к файлу с исходным кодом (без расширения). Наш файл с исходным кодом называется `bot.js`, поэтому укажем путь `bot`. Кроме региона и модуля API, также нужно указать флаг `--configure-fb-`

bot. Он обеспечит автоматическую настройку чат-бота для Facebook Messenger. Для начала посмотрим, как действует наш чат-бот, а потом исследуем его работу более подробно.

Листинг 8.2. Развертывание чат-бота и его настройка для Facebook Messenger

```
claudia create \
  --region eu-central-1 \
  --api-module bot \
  --configure-fb-bot
```

Настройка API Gateway.

Настройка чат-бота для Facebook Messenger.

Команда `claudia create` с флагом `--configure-fb-bot` будет выполнена в интерактивном режиме. Если развертывание API завершится успехом, команда предложит ввести ключ доступа к странице Facebook, а затем выведет URL-адрес точки входа с вашим ключом верификации. (Подробное описание процесса настройки Facebook приводится в приложении B.)

> **ПРИМЕЧАНИЕ.** После развертывания чат-бота общаться с ним смогут только пользователи, добавленные в страницу Facebook и в приложение Facebook. Чтобы сделать чат-бот общедоступным, необходимо передать его для проверки. За дополнительной информацией о процессе проверки обращайтесь по адресу: https://developers.facebook.com/docs/messenger-platform/app-review/.

После ввода ключа доступа к странице Facebook бот будет готов и доступен для тестирования. Чтобы проверить бота, перейдите на свою страницу в Facebook и отправьте ему сообщение. В настоящее время бот всегда возвращает один и тот же текст «Hello from Aunt Maria's pizzeria!» («Привет из пиццерии тетушки Марии!»), как показано на рис. 8.2.

8.3. Какие виды пиццы у нас имеются?

Создание чат-бота за несколько минут действует воодушевляюще, но на данный момент он не имеет никакой практической ценности. Чтобы сделать его полезным, нужно дать клиентам возможность посмотреть список доступных видов пицц и оформить заказ. Начнем с отображения списка.

Как вы наверняка помните, в настоящее время список доступных видов пицц хранится в статическом файле JSON. Как временное решение скопируйте этот файл в новый проект. Сначала создайте в проекте `pizza-fb-chatbot` папку `data`, а затем скопируйте туда файл `pizzas.json` из папки `pizza-api`.

Следующее, что мы должны сделать, – вернуть список пицц в ответе чат-бота. Выбор формы вежливого обращения чат-бота к клиентам выходит за рамки этой книги, поэтому просто придумайте достаточно дружеское сообщение, которое должно появиться перед списком, например: «Привет, вот меню нашей пиццерии», – а затем спросите пользователя, какую пиццу тот желает заказать. Для этого мы должны внести изменения в файл `bot.js`.

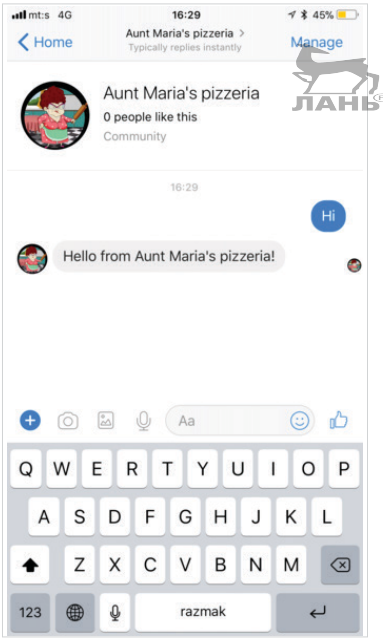


Рис. 8.2. Первое сообщение, полученное от чат-бота

Сначала импортируйте список пицц из файла `pizzas.json`, который находится в папке `data`. Затем измените функцию-обработчик сообщений `botBuilder`, вернув из нее список пицц и предложив пользователю выбрать понравившуюся ему для заказа.

Claudia Bot Builder позволяет вернуть пользователю несколько ответов. Для этого достаточно вернуть массив сообщений вместо одной статической строки. Каждое сообщение в массиве будет отправлено отдельно, в порядке их следования в массиве.

Теперь нужно прочитать список пицц из файла JSON и преобразовать описание каждой пиццы в строку. Для этого содержимое файла можно отобразить в массив, получить название каждой пиццы и затем преобразовать массив в строку с помощью функции `Array.join`.

Измененный код в файле `bot.js` должен выглядеть, как показано в листинге 8.3.

Листинг 8.3. Чат-бот возвращает список пицц

```
'use strict'

const pizzas = require('./data/pizzas.json')

const botBuilder = require('claudia-bot-builder')

const api = botBuilder(() => {
```

Импортировать файл JSON со списком пицц.

```
  return [
    `Hello, here's our pizza menu: ` + pizzas.map(pizza => pizza.name).
    join(', '),
    'Which one do you want?'
  ]
})

module.exports = api
```

Первое сообщение – список пицц.

Вернуть несколько сообщений в виде массива.

Второе сообщение – предложение выбрать понравившуюся пиццу.

Разверните обновленного бота командой `claudia update` без аргументов. Примерно через минуту команда завершится и вернет результат, как показано в листинге 8.4.

Листинг 8.4. Результат выполнения команды `claudia update`

```
{
  "FunctionName": "pizza-fb-bot",
  "FunctionArn": "arn:aws:lambda:eu-central-1:721177882564:function:pizza-fb-
bot:2",
  "Runtime": "nodejs6.10",
  "Role": "arn:aws:iam::721177882564:role/pizza-fb-bot-executor",
  "url": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/latest",
  "deploy": {
    "facebook": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/
      latest/facebook",
    "slackSlashCommand": "https://wvztkdiz8c.execute-api.eu-central-1.
      amazonaws.com/latest/slack/slash-command",
    "telegram": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/
      latest/telegram",
    ...
  }
}
```

Имя функции Lambda.

Версия окружения Node.js, в котором выполняется функция.

ARN функции Lambda.

Роль для функции.

Адреса URL точек входа для всех поддерживаемых платформ.

Адрес URL для API Gateway.

Попробуйте послать сообщение чат-боту, и вы увидите новый ответ, как показано на рис. 8.3.

8.4. Ускорение развертывания

Как вы уже, наверное, заметили, обновление чат-бота занимает чуть больше времени, чем обновление API. Это объясняется тем, что Claudia Bot Builder не знает, изменилась ли конфигурация API, и поэтому перестраивает маршруты к точкам входа для всех поддерживаемых платформ. Подробнее о том, как работает Claudia Bot Builder, рассказывается в разделе 8.6.

К счастью, есть возможность пропустить этап перестройки маршрутов и ускорить процесс развертывания. Для этого нужно передать команде `claudia`

update параметр --cache-api-config с именем переменной, где будет храниться конфигурация API. При вызове команды с этим параметром библиотека Claudia создаст хеш конфигурации API Gateway и сохранит его в переменной с указанным именем. При каждом последующем развертывании claudia update будет проверять наличие этой переменной и сравнивать хеш, чтобы определить, необходимо ли обновить конфигурацию API Gateway. Это ускоряет развертывание, если маршруты API не изменились.

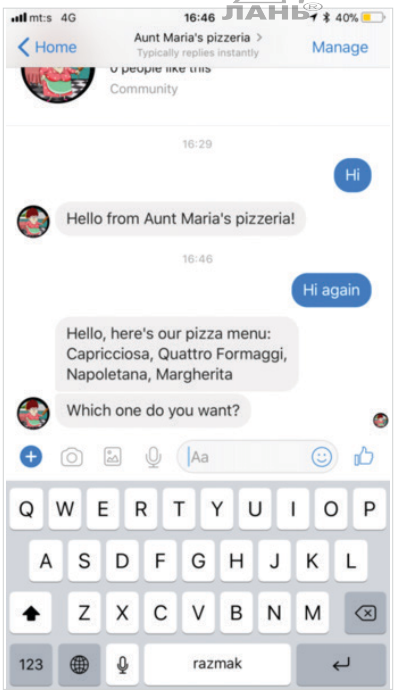


Рис. 8.3. Новый ответ чат-бота

Мы рекомендуем добавить команду claudia update с параметром --cache-api-config как сценарий NPM в файл package.json. После этого вам также следует определить библиотеку Claudia как зависимость dev (см. приложение А). Ваш файл package.json должен выглядеть примерно так, как показано в листинге 8.5.

Листинг 8.5. Файл package.json со сценарием обновления

```
{
  "name": "pizza-fb-chatbot",
  "version": "1.0.0",
  "description": "A pizzeria chatbot",
  "main": "bot.js",
  "scripts": {
    "update": "claudia update --cache-api-config apiConfig"
```

Добавить команду claudia update как сценарий NPM.

```
  },
  "license": "MIT",
  "dependencies": {
    "claudia-bot-builder": "^2.15.0"
  },
  "devDependencies": {
    "claudia": "^2.13.0"
  }
}
```

Определить Claudia как зависимость dev.

Параметр `--cache-api-config` удобно использовать, когда API изменяется нечасто, потому что он значительно ускоряет развертывание. Но Claudia Bot Builder создает точки входа для всех платформ, и если вы создаете чат-бота только для одной платформы, эти точки входа не нужны. Начиная с версии 2.7.0 Claudia Bot Builder позволяет включать поддержку только определенного круга платформ. Для этого нужно передать функции `botBuilder` второй аргумент – объект с параметрами, перечисляющий поддерживаемые платформы. Например, чтобы включить только платформу Facebook Messenger, добавьте в объект с параметрами ключ `platforms` с массивом, содержащим строку `"facebook"`, как показано в листинге 8.6.

Узнать больше о выборе платформ можно по адресу: https://github.com/claudiajs/claudia-bot-builder/blob/master/docs/API.md.

Листинг 8.6. Чат-бот, поддерживающий только платформу Facebook Messenger

```
'use strict'

const pizzas = require('./data/pizzas.json')

const botBuilder = require('claudia-bot-builder')

const api = botBuilder(() => {
  return [
    'Hello, here's our pizza menu: ' + pizzas.map(pizza => pizza.name).
    join(', '),
    'Which one do you want?'
  ]
}, {
  platforms: ['facebook']
})

module.exports = api
```

Передать функции botBuilder объект с параметрами во втором аргументе.

Перечислить в массиве поддерживаемые платформы. В данном случае поддерживается только Facebook.

Если теперь выполнить команду `npm run update`, вы увидите, что развертывание произойдет намного быстрее.

8.5. Шаблоны для взаимодействий

Теперь чат-бот для пиццерии тетушки Марии посылает клиентам список пицц, которые они могут заказать. Однако клиенты могут не понимать, что делать дальше.

Создать хороший чат-бот сложно. Текстовый интерфейс непривычен для большинства пользователей, а кроме того, в чат-боте необходимо предусмотреть некоторую обработку естественного языка и реализацию искусственного интеллекта – и то, и другое сложно настроить правильно. Для некоторых разговорных языков в настоящее время это практически невозможно. Эта проблема давно известна, поэтому многие платформы для чат-ботов предлагают поддержку элементов интерфейса, напоминающих графический интерфейс приложений, таких как кнопки и списки.

Facebook Messenger – одна из таких платформ, и ее элементы пользовательского интерфейса называются *шаблонами*. Она предлагает несколько разных шаблонов, в том числе:

- *универсальный* – посылает сообщение в форме списка с горизонтальной прокруткой, включающего карточки, каждая из которых имеет заголовок, подзаголовок/описание, изображение и до трех кнопок;
- *кнопка* – посылает сообщение с простыми кнопками (до трех) под текстом;
- *список* – посылает сообщение в форме вертикального списка с названиями, описаниями, изображениями и кнопкой;
- *квитанция* – посылает подтверждение заказа (квитанцию) после оформления заказа.

Полный список поддерживаемых шаблонов можно найти на странице https://developers.facebook.com/docs/messenger-platform/send-messages/templates/.

В нашем случае можно использовать шаблон списка или универсальный шаблон. Но шаблон списка имеет ограничение по размеру – чтобы использовать его, мы должны представить список не менее чем с двумя и не более чем с четырьмя элементами. Универсальный шаблон более гибкий; он может отображать от 1 до 10 элементов. Поскольку нам нужно отображать более четырех пицц, используем универсальный шаблон.

Чтобы вместо текста со списком пицц вернуть шаблон, ответ следует оформить в виде объекта JSON с определенной структурой. На первый взгляд кажется, что все просто, но эти объекты JSON могут быть довольно большими, и, поскольку мы собираемся отображать до 10 пицц, это ухудшит удобочитаемость кода.

Чтобы улучшить удобочитаемость и упростить работу с шаблонами, Claudia Bot Builder предлагает классы-обертки шаблонов для некоторых из поддерживаемых платформ (включая Facebook, Telegram и Viber). Построитель сообщений для Facebook доступен как `botBuilder.fbTemplate` и предлагает набор классов для каждого из поддерживаемых шаблонов.

> **ПРИМЕЧАНИЕ.** Полный список классов в Claudia Bot Builder в построителе сообщений можно найти на странице https://github.com/claudiajs/claudia-bot-builder/blob/master/docs/FB_TEMPLATE_MESSAGE_BUILDER.md.

Как упоминалось выше, универсальный шаблон отображает список элементов с горизонтальной прокруткой. Каждый элемент включает изображение, заголовок, необязательное описание и кнопки для ввода ответа пользователя. Кнопки в универсальном шаблоне могут выполнять различные действия, такие как открытие URL-адреса или отправка ответа клиента в точку входа. Полный список действий, поддерживаемых кнопками, и подробную информацию о шаблоне можно найти на странице https://developers.facebook.com/docs/messenger-platform/send-messages/template/generic.

В Claudia Bot Builder универсальный шаблон доступен в виде класса `botBuilder.fbTemplate.Generic`. Нам нужно инициализировать класс, вызвав конструктор без аргументов, и сохранить экземпляр в константе `message`.

Затем для каждой пиццы нужно добавить свой элемент в список, также известный как *пузырь*. Для этого выполним обход массива пицц и для каждой добавим элемент вызовом метода `message.addBubble` класса `fbTemplate.Generic`. Этот метод принимает текст заголовка для элемента.

Далее для каждой пиццы добавим изображение и кнопку с помощью методов `addImage` и `addButton` соответственно. Методу `addImage` необходимо передать действительный URL-адрес изображения, а методу `addButton` – имя кнопки и значение, которое будет передано обработчику в случае ее нажатия. Для начала добавим только кнопку **Details** (Подробности), которая отправит идентификатор пиццы в качестве значения. Логику кнопок мы реализуем в следующей главе.

Вызовы всех методов класса можно объединить в цепочку, как показано ниже:

```
message.addBubble(pizza.name).addImage(pizza.image).addButton('Details', pizza.id)
```

В конце цепочки следует вызвать метод `message.get`, чтобы преобразовать элемент списка (пузырь) в объект JSON, который ожидает получить Facebook. Поскольку для оформления заказа пользователи будут применять шаблон кнопки (эта возможность будет реализована в следующей главе), мы можем заменить надпись «Which one do you want?» («Какую желаете?») вызовом `message.get`.

Измененный код в файле `bot.js` показан в листинге 8.7.

Листинг 8.7. Функция Bot Builder, возвращающая ответ в виде универсального шаблона

```
'use strict'

const pizzas = require('./data/pizzas.json')

const botBuilder = require('claudia-bot-builder')
```

```
const fbTemplate = botBuilder.fbTemplate

const api = botBuilder(() => {
  const message = new fbTemplate.Generic()

  pizzas.forEach(pizza => {
    message.addBubble(pizza.name)
      .addImage(pizza.image)
      .addButton('Details', pizza.id)
  })

  return [
    'Hello, here's our pizza menu:',
    message.get()
  ]
}, {
  platforms: ['facebook']
})

module.exports = api
```

Создать константу fbTemplate для хранения экземпляра построителя сообщений для Facebook.

Создать новый экземпляр класса шаблона Generic.

Обход списка пицц.

Добавить элемент списка для каждой пиццы.

Добавить изображение пиццы.

Добавить кнопку для каждой пиццы и указать идентификатор пиццы, который нужно передать после нажатия кнопки пользователем.

После изменения файла `bot.js` выполните команду `npm run update`. Как она завершится, вы сможете отправить новое сообщение своему чат-боту. Ответ должен выглядеть примерно так, как показано на рис. 8.4.

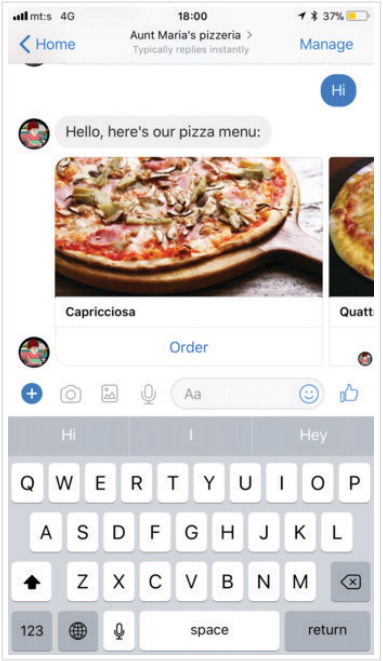


Рис. 8.4. Ответ чат-бота, оформленный с универсальным шаблоном

8.6. Как работает Claudia Bot Builder?

Теперь, после создания чат-бота, возвращающего список пицц, можно посмотреть, как работает Claudia Bot Builder.

Большинство популярных платформ для чат-ботов использует точки входа для уведомления сервера о получении нового сообщения. Но каждая платформа отправляет данные с разной структурой, а также ожидает, что ответ будет соответствовать конкретной платформе.

Главная цель Claudia Bot Builder – абстрагировать структуру, характерную для той или иной платформы, и дать возможность приема и отправки сообщений с использованием простого API. Он использует Claudia API Builder для создания точек входа для каждой поддерживаемой платформы. На момент написания этих строк Claudia Bot Builder поддерживал 10 платформ (включая Facebook Messenger, Slack, Amazon Alexa и Telegram).

Как показано на рис. 8.5, жизненный цикл сообщение–ответ в Claudia Bot Builder выглядит следующим образом:

1) пользователь посылает сообщение, используя платформу обмена сообщениями;
2) платформа передает сообщение в точку входа через API Gateway, которую вы указали в настройках;
3) API Gateway запускает вашу функцию Lambda, передавая запрос конечной точке API для конкретной платформы;
4) запрос преобразуется в универсальный формат с использованием парсера для конкретной платформы;
5) преобразованное сообщение передается логике вашего чат-бота;
6) ответ чат-бота преобразуется в формат для данной платформы;
7) Claudia Bot Builder вызывает API платформы и передает ответ;
8) API платформы возвращает ответ приложению обмена сообщениями пользователя.

Вы уже знаете, что функция `botBuilder` принимает функцию-обработчик сообщений и объект с дополнительными параметрами. Функция-обработчик – это логика чат-бота, а объект с параметрами используется только для определения поддерживаемых платформ, чтобы ускорить развертывание.

Функция-обработчик вызывается с двумя аргументами: объектом сообщения и объектом запроса Claudia API Builder.

Объект преобразованного сообщения имеет следующие свойства:

- `text` – текст сообщения. Чтобы реализовать ответы на текстовые сообщения, достаточно использовать это единственное свойство;
- `type` – название платформы, через которую получено сообщение; список платформ можно найти на странице https://github.com/claudiajs/claudia-bot-builder/blob/master/docs/API.md;

- `sender` – идентификатор отправителя. Зависит от платформы, но в большинстве случаев это идентификатор пользователя;
- `postback` – булево свойство. Имеет значение `true`, если сообщение является результатом отправки ответа клиента в Facebook (например, если пользователь нажал кнопку в универсальном шаблоне). Для новых сообщений или если платформа не поддерживает ответов, содержит значение `false`;
- `originalRequest` – оригинальный объект сообщения, полученный точкой входа. Может пригодиться для выполнения некоторых операций, в зависимости от платформы, которые не поддерживаются модулем Claudia Bot Builder.

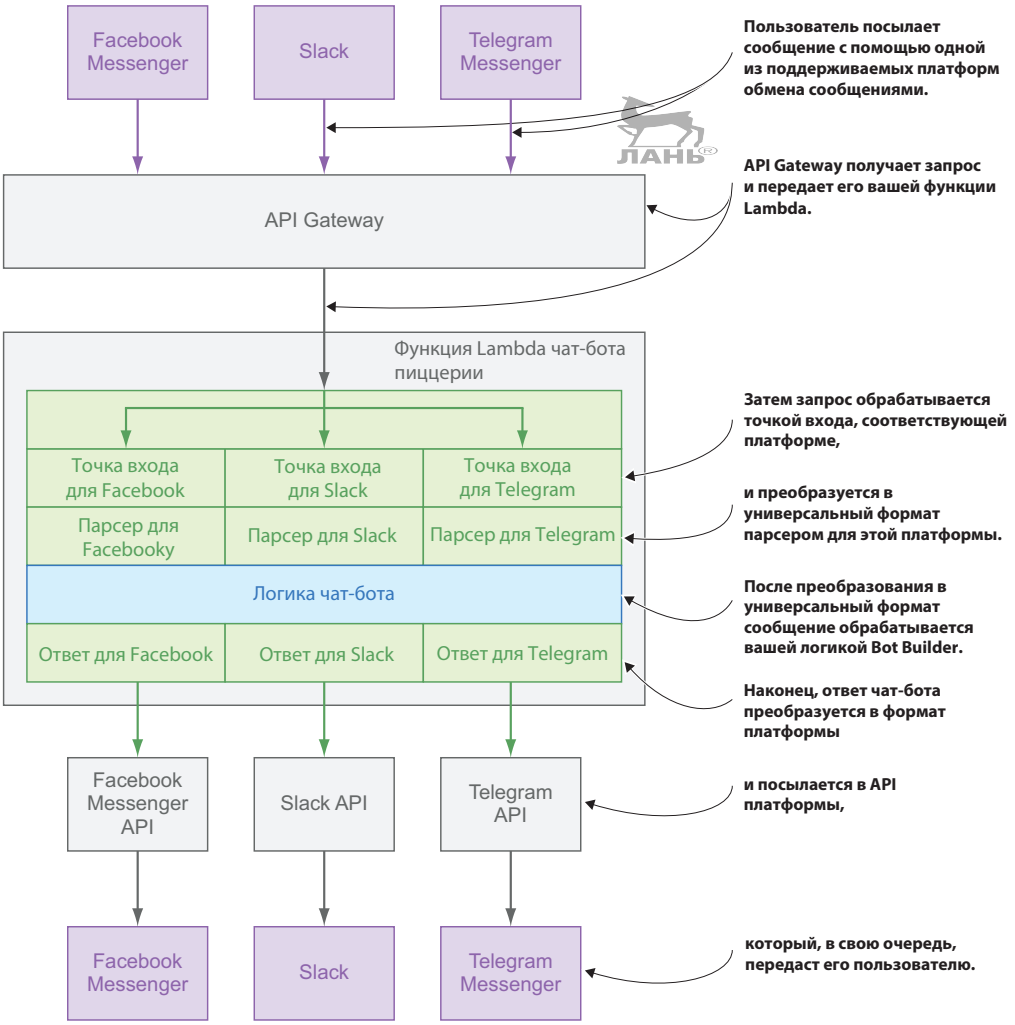


Рис. 8.5. Процесс преобразования сообщения в Claudia Bot Builder

Наконец, вернуть ответ на сообщение из Claudia Bot Builder так же просто, как из Claudia API Builder: чтобы ответить текстовым сообщением, нужно вернуть строку. Также можно ответить шаблоном для выбранной платформы, воспользоваться построителем сообщений из шаблонов или вернуть объект JSON. Чтобы отправить ответ асинхронно, верните текст или объект в конце цепочки `Promise`.

8.7. Опробование!

В этой главе вы познакомились с основами реализации чат-ботов для Facebook Messenger с использованием AWS Lambda. Наш чат-бот не способен выиграть приз Лебнера, но может доставить немало приятных минут. Как и во всех предыдущих главах, мы подготовили для вас упражнение.

8.7.1. Упражнение

Цель первого упражнения – показать вам, насколько просто создаются чат-боты. Используя ту же страницу и приложение Facebook, создайте чат-бота, который будет отображать текст полученного сообщения в обратном порядке.

Вот несколько советов:

- исследуйте параметр `message`, познакомьтесь со всеми его атрибутами и реализуйте возврат перевернутого текста сообщения, отправленного пользователем;
- для переворачивания текста используйте встроенные методы.

8.7.2. Решение

Это задание имеет простое и очевидное решение, представленное в листинге 8.8.

Листинг 8.8. Простой чат-бот, возвращающий перевернутый текст полученного сообщения

```
'use strict'

const botBuilder = require('claudia-bot-builder')

const api = botBuilder((message) => {
  return message.text.split('').reverse().join('')
}, {
  platforms: ['facebook']
})

module.exports = api
```

Импортировать Claudia Bot Builder.

Вызвать функцию botBuilder с функцией-обработчиком, принимающей атрибут message.

Вернуть текст полученного сообщения.

Передать объект с параметрами, определяющий поддержку только платформы Facebook Messenger.

Экспортировать экземпляр Claudia API Builder, созданный функцией botBuilder.

Этот чат-бот прост, но не надейтесь, что остальные будут такими же простыми!

В заключение

- Библиотека Claudia позволяет развернуть чат-бот для нескольких платформ одной командой.
- Claudia Bot Builder – это обертка вокруг Claudia API Builder, которая возвращает экземпляр API.
- Claudia Bot Builder упаковывает текстовый ответ в формат, поддерживаемый платформой, которой посылается сообщение.
- Используя Bot Builder, можно создавать шаблоны сообщений для конкретной платформы.

Глава 9

Ввод... асинхронные и отложенные ответы

Эта глава охватывает следующие темы:

- подключение бессерверного чат-бота к AWS DynamoDB;
- отправка отложенных сообщений пользователю, когда пицца будет готова;
- интегрирование простой обработки естественного языка.

Возможность быстро создавать и развертывать различные приложения трудно переоценить. Как вы уже видели, Claudia Bot Builder существенно упрощает создание чат-ботов, и вы можете создать простой чат-бот типа запрос/ответ всего за несколько минут.

Но в реальном мире чат-боты должны выполнять более сложные операции, чем просто возвращать статические сообщения. Вам, вероятно, понадобится хранить информацию о клиенте и запрашивать дополнительные данные, а также делать некоторые расчеты или даже отвечать на некоторые несвязанные вопросы. Обо всем этом мы расскажем в данной главе: здесь вы узнаете, как создавать заказы на пиццу по запросам пользователей, как отправить сообщение в службу доставки, когда заказ будет готов, и как интегрировать простые алгоритмы обработки естественного языка (Natural Language Processing, NLP) для анализа ввода пользователя.

9.1. Добавление интерактивности в чат-бот

Ваша двоюродная сестра Джулия следила за вашими успехами и была обрадована, увидев, что вы создали прокручиваемый список доступных пицц. Она уже начала распространять в школе информацию об удивительном чат-боте для заказа пиццы, который гораздо лучше, чем в пиццерии Chess. Это ставит

вас в тупик, но тетя Мария счастлива, потому что она уже заметила увеличение трафика на ее веб-сайте в результате этого слуха.

Но не будем разочаровывать их и закончим чат-бот для заказа пиццы, добавив несколько улучшений, чтобы наш бот превзошел бота пиццерии Chess.

9.1.1. Выбор заказа: получение ответа от пользователя

Отображение списка с фотографиями пиццы в ответе чат-бота – отличная идея, потому что клиенты предпочитают визуальный интерфейс простому текстовому ответу. В предыдущей главе каждая пицца отображалась в своем визуальном блоке, в котором также присутствует кнопка **Details** (Подробности). Однако если нажать ее, это ничего не изменит.

Так как наша главная цель – дать возможность заказать пиццу, мы сделаем следующее:

1) добавим кнопку **Order** (Заказать) под кнопкой **Details** (Подробности), как показано на рис. 9.1;

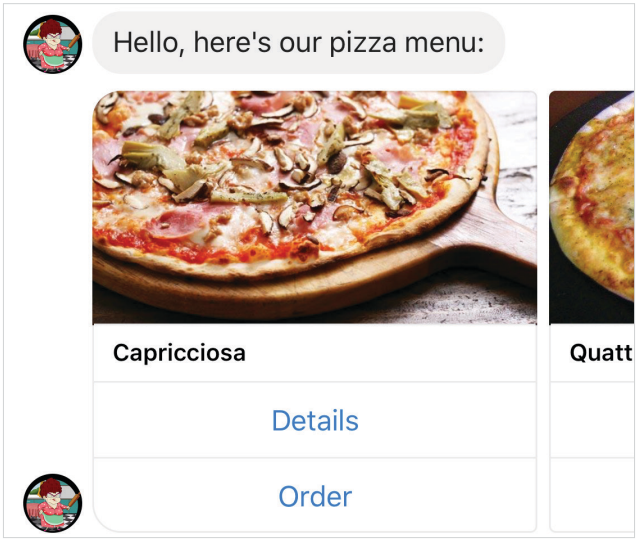


Рис. 9.1. Чат-бот откликается на нажатие кнопок **Details** (Подробности) и **Order** (Заказать)

2) реализуем оформление заказа пиццы, сохранив информацию для заказа в базе данных после нажатия кнопки **Order** (Заказать);
3) запланируем выполнение заказа;
4) добавим в чат-бот обработку текста на естественном языке, чтобы наделить наш чат-бот некоторым интеллектом и сделать для клиентов общение с ним более привлекательным. К концу он сможет отвечать на любые вопросы одноклассников Джулии.

Обработка естественного языка

Обработка естественного языка (Natural Language Processing, NLP) – это раздел искусственного интеллекта, который занимается анализом и созданием текстов и речи на естественных языках, которые люди могут использовать для взаимодействия с компьютерами.

Желающим заняться изучением NLP мы рекомендуем книгу Хобсона Лейна (Hobson Lane) «Natural Language Processing in Action», изданную издательством Manning: https://www.manning.com/books/natural-language-processing-in-action.

Далее мы продолжим разработку чат-бота с того места (листинг 9.1), на котором остановились в главе 8.

Листинг 9.1. Текущая функция `botBuilder`, возвращающая универсальный шаблон

```
'use strict'

const pizzas = require('./data/pizzas.json')

const botBuilder = require('claudia-bot-builder')
const fbTemplate = botBuilder.fbTemplate

const api = botBuilder(message => {
  const messageTemplate = new fbTemplate.Generic()

  pizzas.forEach(pizza => {
    messageTemplate.addBubble(pizza.name)
      .addImage(pizza.image)
      .addButton('Details', pizza.id)
  })

  return [
    'Hello, here\'s our pizmenu:',
    messageTemplate.get()
  ]
}, {
  platforms: ['facebook']
})

module.exports = api
```

Создать новый экземпляр класса Generic универсального шаблона для отправки сообщения в Facebook.

Подготовить шаблон меню Facebook для выбора пиццы.

Послать приветственное сообщение с меню для выбора пиццы.

Как можно заметить в листинге 9.1, аргумент `message` функции `botBuilder` содержит полезную информацию о запросе, полученном чат-ботом, например: является ли сообщение результатом нажатия кнопки или это текстовое сообщение, написанное пользователем.

Чтобы сделать чат-бот более полезным, мы должны вернуть подробное описание пиццы, когда клиент нажмет кнопку **Details** (Подробности). Нажатие кнопки **Order** (Заказать) клиентом должно запустить процедуру оформления заказа выбранной пиццы.

Теперь процесс обработки запросов в чат-боте разделится на три ветви (рис. 9.2):

- пользователь может посмотреть подробности о выбранной пицце;
- пользователь может заказать выбранную пиццу;
- в любых других случаях чат-бот должен вернуть начальное сообщение с меню.

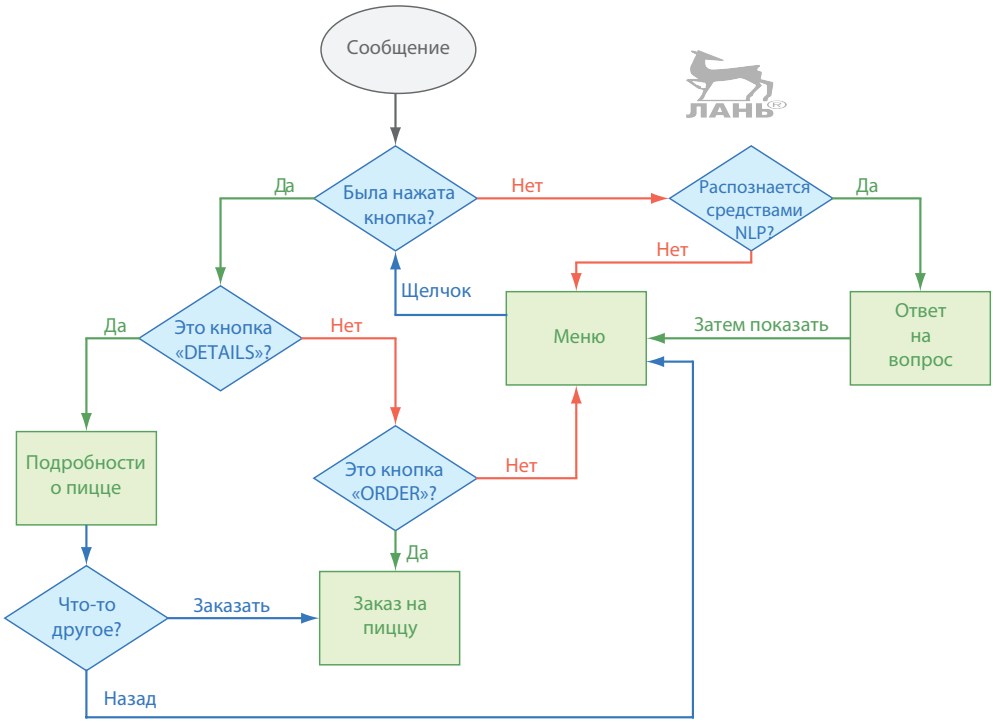


Рис. 9.2. Алгоритм работы чат-бота

Запросы, сгенерированные нажатием кнопок, отличаются значением в свойстве `postback` сообщений.

Для реализации алгоритма, представленного на рис. 9.2, прежде всего нужно проверить, является ли сообщение ответом пользователя, результатом нажатия кнопки, сравнив `message.postback` со значением `true`.

Если сообщение является ответом пользователя (`postback` имеет значение `true`), далее нужно проверить, пожелал ли пользователь посмотреть подробное описание пиццы или заказать ее (назовем это *действием*), и извлечь идентификатор выбранной пиццы. Чтобы сохранить оба значения, можно обновить значение нажатой кнопки, сериализовав значение JSON, или сформировать строку, перечисляющую действие и идентификатор пиццы через символ вертикальной черты (`|`). В случае с чат-ботом последний вариант выглядит проще. Мы можем сохранить строку в формате `ДЕЙСТВИЕ | ИДЕНТИФИКАТОР`, где `ДЕЙСТВИЕ` – название выполненного действия (например, `ORDER` или `DETAILS`), а `ИДЕНТИФИКАТОР` – идентификатор пиццы.

Если сообщение является ответом пользователя, его значение будет храниться в `message.text`. Извлечь имя действия и идентификатор пиццы в этом случае можно, разбив строку по символу вертикальной черты (`|`) с помощью встроенного метода `String.split`:

```
const values = message.text.split('|')
```

Новый массив `values` будет хранить действие в первом элементе, а идентификатор пиццы – во втором. Но поддержка деструктуризации в ES6 может сделать этот код проще. Если заменить `const values` на `const [action, pizzaId]`, действие сохранится непосредственно в константе `action`, а идентификатор пиццы – в `pizzaId`.

После извлечения действия и идентификатора пиццы нужно проверить, какое действие выбрано – `ORDER` или `DETAILS`.

Идентификатор пиццы нам потребуется в любом случае, чтобы найти пиццу в массиве `pizzas`. Для этого можно использовать встроенный метод `Array.find`:

```
const pizza = pizzas.find(pizza => pizza.id == pizzaId)
```

Обратите внимание, что в этом примере используется оператор `==` вместо `===`. Это объясняется тем, что `pizzaId` имеет тип `String`, так как мы получили идентификатор с помощью функции `String.split`, а идентификаторы в массиве `pizzas` – целые числа.

Поиск пиццы по идентификатору должен выполняться в обеих инструкциях `if`. Это может показаться избыточным, но дело в том, что позднее может быть добавлено действие, не требующее идентификатора пиццы.

> **ПРИМЕЧАНИЕ.** Если вы переместили список пицц в таблицу DynamoDB, то можете остановиться на этом и попробовать подключить своего чат-бота к таблице в DynamoDB – подобно тому, как мы подключили свой API в главе 3. Если у вас это не получится сделать самостоятельно, не волнуйтесь, мы займемся подключением к DynamoDB далее в этой главе.

Когда ваш чат-бот получит действие `DETAILS`, мы должны составить список ингредиентов, перечислив их через запятую, и вернуть этот список клиенту. Но что далее должен сделать клиент, получив список ингредиентов?

В отличие от веб-приложений, где пользователь может видеть на экране следующие доступные действия, в потоке чата следующий шаг не всегда очевиден для пользователя. Если вернуть только список ингредиентов, пользователь, скорее всего, не поймет, что делать дальше, и в результате вы можете получить массу неожиданных пользовательских сообщений, таких как «Фу-у! Козий сыр!», или «Какую пиццу вы предпочитаете?», или даже «Я люблю вас», потому что человеческая фантазия бесконечна.

Даже обладая поддержкой анализа естественного языка, чат-боты все еще очень далеки от возможности общаться на человеческом уровне, поэтому лучшее, что можно сделать, – это добавить обработку ошибок и попытаться направить пользователя в русло, с которым наш чат-бот сможет справиться. Проектирование чат-ботов – интересная тема, но ее обсуждение выходит за рамки этой книги.

Самый простой способ направить пользователя к следующему действию – показать меню с доступными вариантами. Это не гарантирует, что пользователь нажмет одну из кнопок, но меню поможет получить более однозначные результаты, чем простой вопрос.

Мы должны показать два варианта: возможность заказать пиццу, которую только что просмотрел пользователь, или вернуться к списку пицц. Для этого используем класс `Button` из `fbTemplate`. Класс `Button` позволяет отобразить шаблон с тремя кнопками, которые выглядят как кнопки из универсального шаблона и отображают текстовый ответ. Этот класс используется подобно классу `Generic`, поэтому наш ответ должен выглядеть примерно так:

```
return [
  `${pizza.name} has following ingredients: ` + pizza.ingredients.join(', '),
  new fbTemplate.Button('What else can I do for you?')
    .addButton('Order', `ORDER|${pizzaId}`)
    .addButton('Show all pizzas', 'ALL_PIZZAS')
    .get()
]
```

Как видите, вторая кнопка имеет значение `ALL_PIZZAS`, поэтому этот вариант отвергнут обе инструкции `if` и клиенту вернется меню со списком пицц. Позже вы сможете изменить этот порядок действий, чтобы показать какое-то другое сообщение, в зависимости от предыдущего диалога, например: «Не любите грибы? Тогда вот еще несколько пицц, которые могут вам понравиться».

> **ПРИМЕЧАНИЕ.** За дополнительной информацией о классе `Button` обращайтесь к документации по адресу: https://github.com/claudiajs/claudia-bot-builder/blob/master/docs/FB_TEMPLATE_MESSAGE_BUILDER.md.

Если клиент выбрал действие `ORDER`, чат-бот должен отыскать пиццу по идентификатору и сообщить пользователю, что заказ принят. Мы реализуем этот вариант чуть позже, в этой же главе.

Наконец, если сообщение не является ответом пользователя (то есть не является результатом нажатия кнопки) или если указано действие, отличное от `DETAILS` и `ORDER`, можно вернуть универсальный ответ, подобный тому, что мы возвращали в главе 8.

Единственное отличие заключается в дополнительной кнопке **Order** (Заказать):

```
pizzas.forEach(pizza => {
  reply.addBubble(pizza.name)
    .addImage(pizza.image)
    .addButton('Details', `DETAILS|${pizza.id}`)
    .addButton('Order', `ORDER|${pizza.id}`)
})
```

Обновленный файл `bot.js` теперь должен выглядеть, как показано в листинге 9.2.

Листинг 9.2. Чат-бот, принимающий заказы и возвращающий подробную информацию о выбранной пицце

```
'use strict'

const pizzas = require('./data/pizzas.json')

const botBuilder = require('claudia-bot-builder')
const fbTemplate = botBuilder.fbTemplate

const api = botBuilder((message) => {
  if (message.postback) {
    const [action, pizzaId] = message.text.split('|')
    if (action === 'DETAILS') {
      const pizza = pizzas.find(pizza => pizza.id == pizzaId)

      return [
        `${pizza.name} has following ingredients: ` + pizza.ingredients.
    join(', '),
        new fbTemplate.Button('What else can I do for you?')
          .addButton('Order', `ORDER|${pizzaId}`)
          .addButton('Show all pizzas', 'ALL_PIZZAS')
          .get()
      ]
    } else if (action === 'ORDER') {
      const pizza = pizzas.find(pizza => pizza.id == pizzaId)

      return `Thanks for ordering ${pizza.name}! I will let you know as soon
```

Добавить параметр message в функцию-обработчик.

Проверить, является ли сообщение ответом пользователя.

Если это ответ, разбить текст сообщения по символу вертикальной черты.

Первая часть сообщения DETAILS?

Если да, получить идентификатор пиццы из массива.

Вернуть список ингредиентов выбранной пиццы.

Также вернуть меню, чтобы пользователю было куда пойти дальше.

Первая часть сообщения ORDER?

И снова получить идентификатор пиццы.

```
as your pizza is ready.`
    }
  }

  const reply = new fbTemplate.Generic()

  pizzas.forEach(pizza => {
    reply.addBubble(pizza.name)
      .addImage(pizza.image)
      .addButton('Details', `DETAILS|${pizza.id}`)
      .addButton('Order', `ORDER|${pizza.id}`)
  })

  return [
    `Hello, here's our pizza menu:`,
    reply.get()
  ]
}, {
  platforms: ['facebook']
})

module.exports = api
```

Затем вернуть идентификатор, чтобы подтвердить заказ.

Если это не ответ пользователя, сгенерированный нажатием кнопки, показать главное меню.

Теперь выполните команду `claudia update` или `npm run update`, чтобы развернуть чат-бот, и попробуйте побеседовать с ним (рис. 9.3).

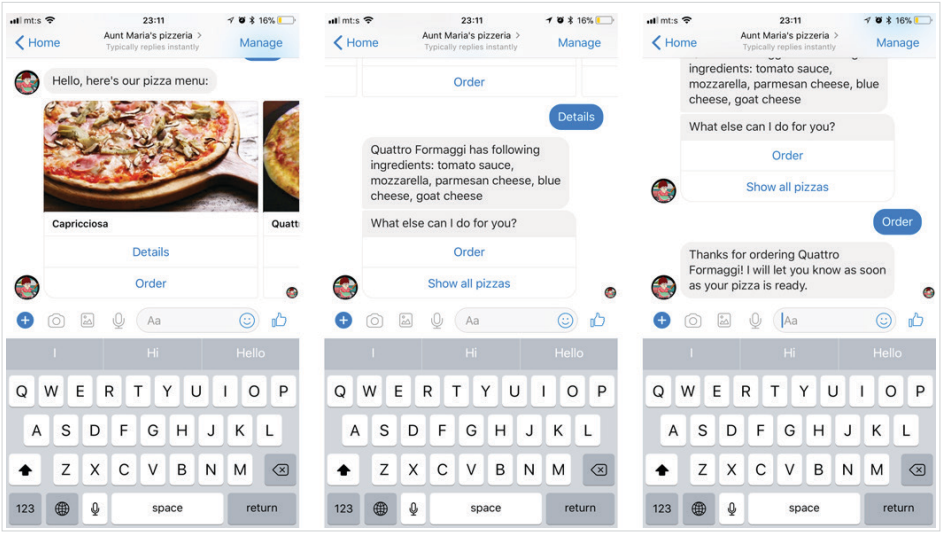


Рис. 9.3. Скриншот, демонстрирующий работу трех ветвей диалога

9.2. Улучшение масштабируемости чат-бота

Как и в случае с API, поток выполнения чат-бота, организованный в одном файле, плохо масштабируется. Можно ли улучшить его организацию?

Чат-бот не имеет маршрутизатора, но у нас есть инструкции `if...else`, которые действуют подобно маршрутизатору, а операции внутри них выглядят как обработчики. Самый простой способ улучшить структуру чат-бота – сохранить маршрутизацию в главном файле и переместить обработчики в отдельные файлы. То есть мы должны оставить основной файл `bot.js` и создать папку `handlers` с тремя файлами обработчиков, по одному для каждой ветви диалога. Соответствующая структура папок проекта показана на рис. 9.4.

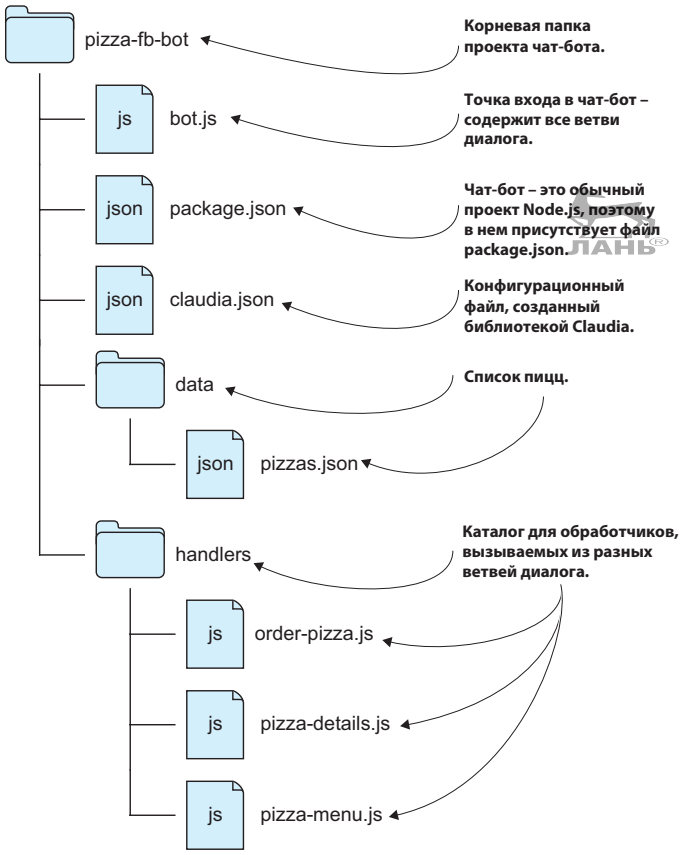


Рис. 9.4. Структура папок проекта чат-бота

Создайте папку `handlers` в папке проекта `pizza-fb-chatbot` и три файла с исходным кодом на JavaScript:

- `order-pizza.js`;
- `pizza-details.js`;
- `pizza-menu.js`.

Затем измените содержимое файла `bot.js`:

- удалите инструкцию импорта объекта `fbTemplate`, потому что он не потребуется в этом файле;
- импортируйте три новые функции из только что созданных файлов;
- замените логику, возвращающую информацию о пицце и оформляющую заказ вызовами функций `pizzaDetails` и `orderPizza` с аргументом `pizzaId`;
- замените логику, которая генерирует меню со списком пицц вызовом обработчика `pizzaMenu`.

После внесения изменений файл `bot.js` должен выглядеть, как показано в листинге 9.3.

Листинг 9.3. Главный файл с исходным кодом чат-бота

```
'use strict'

const botBuilder = require('claudia-bot-builder')

const pizzaDetails = require('./handlers/pizza-details')
const orderPizza = require('./handlers/order-pizza')
const pizzaMenu = require('./handlers/pizza-menu')

const api = botBuilder((message) => {
  if (message.postback) {
    const [action, pizzaId] = message.text.split('|')

    if (action === 'DETAILS') {
      return pizzaDetails(pizzaId)
    } else if (action === 'ORDER') {
      return orderPizza(pizzaId)
    }
  }

  return [
    `Hello, here's our pizza menu:`,
    pizzaMenu()
  ]
}, {
  platforms: ['facebook']
})

module.exports = api
```

Импортировать функции-обработчики.

Если сообщение сгенерировано нажатием кнопки и в нем указано действие DETAILS, вызвать обработчик pizzaDetails.

Если сообщение сгенерировано нажатием кнопки и в нем указано действие ORDER, вызвать обработчик orderPizza.

Если сообщение не определяет действия, вернуть главное меню.

Теперь откройте файл `handlers/pizza-details.js`. Сначала импортируйте список пицц из файла `pizza.json` и `fbTemplate` из Claudia Bot Builder:

```
const pizzas = require('../data/pizzas.json')
const fbTemplate = require('claudia-bot-builder').fbTemplate
```

Затем объявите функцию `pizzaDetails` с одним параметром `pizzaId`. Эта функция должна отыскать пиццу по идентификатору в массиве `pizzas` и вернуть ее ингредиенты, используя шаблон кнопки, который позволит пользователю заказать пиццу или вернуться обратно в главное меню.

В конце экспортируйте функцию-обработчик `pizzaDetails`, добавив строку

```
module.exports = pizzaDetails
```

В конечном итоге файл `handlers/pizza-details.js` должен выглядеть, как показано в листинге 9.4.

Листинг 9.4. Функция-обработчик, возвращающая подробности о выбранной пицце

```
'use strict'

const pizzas = require('../data/pizzas.json')
const fbTemplate = require('claudia-bot-builder').fbTemplate

function pizzaDetails(id) {
  const pizza = pizzas.find(pizza => pizza.id == id)

  return [
    `${pizza.name} has following ingredients: ` + pizza.ingredients.join(',
    '),
    new fbTemplate.Button('What else can I do for you?')
      .addButton('Order', `ORDER|${pizza.id}`)
      .addButton('Show all pizzas', 'ALL_PIZZAS')
      .get()
  ]
}

module.exports = pizzaDetails
```

Импортировать список пицц.

Импортировать fbTemplate из Claudia Bot Builder.

Объявление функции-обработчика.

Получить пиццу по идентификатору.

Вернуть список ингредиентов и меню.

Вернуть сообщение для ответа с двумя кнопками, создав экземпляр fbTemplate.Button с текстом ответа и двумя кнопками под ним.

Экспортировать функцию-обработчик.

Далее откройте файл `handlers/order-pizza.js` и проделайте то же самое:

- импортируйте список пицц из файла `pizzas.json`;
- реализуйте функцию-обработчик `orderPizza`, которая принимает параметр с идентификатором пиццы;
- отыщите пиццу по ее идентификатору и верните текстовое сообщение из функции `orderPizza`;

- экспортируйте функцию `orderPizza`.

Файл `order-pizza.js` должен выглядеть, как показано в листинге 9.5.

Листинг 9.5. Функция-обработчик, оформляющая заказ

```
'use strict'

const pizzas = require('../data/pizzas.json')

function orderPizza(id) {
  const pizza = pizzas.find(pizza => pizza.id == id)

  return `Thanks for ordering ${pizza.name}! I will let you know as soon as
    your pizza is ready.`
}

module.exports = orderPizza
```

Импортировать список пицц.

Объявление функции-обработчика.

Получить пиццу по идентификатору.

Вернуть список ингредиентов и меню.

Экспортировать функцию-обработчик.

После реализации обработчика, оформляющего заказ, откройте файл `handlers/pizza-menu.js` и выполните следующие действия:

- импортируйте список пицц и `fbTemplate`;
- объявите функцию-обработчик `pizzaMenu`;
- внутри функции создайте новый универсальный шаблон;
- выполните цикл по всем пиццам в файле `pizza.json` и для каждой добавьте визуальные блоки (пузыри) в универсальный шаблон;
- верните получившееся сообщение;
- экспортируйте функцию `pizzaMenu`.

Обработчик главного меню должен выглядеть, как показано в листинге 9.6.

Листинг 9.6. Обработчик главного меню

```
'use strict'

const pizzas = require('../data/pizzas.json')
const fbTemplate = require('claudia-bot-builder').fbTemplate

function pizzaMenu() {
  const message = new fbTemplate.Generic()

  pizzas.forEach(pizza => {
    message.addBubble(pizza.name)
      .addImage(pizza.image)
```

Импортировать список пицц.

Импортировать fbTemplate из Claudia Bot Builder.

Объявление функции-обработчика.

Создать сообщение с универсальным шаблоном.

```
      .addButton('Details', `DETAILS|${pizza.id}`)
      .addButton('Order', `ORDER|${pizza.id}`)
  })

  return message.get()          ← Вернуть сообщение.
}

module.exports = pizzaMenu      ← Экспортировать функцию-обработчик.
```

После добавления кода во все файлы выполните команду `npm run update` или `claudia update`, чтобы развернуть чат-бот. После отправки сообщения чат-боту через Facebook Messenger вы должны получить тот же ответ, что и раньше, но теперь чат-бот имеет более масштабируемую организацию.

Другие способы организации работы чат-бота

Организовать процесс работы чат-бота не так-то просто. Код в примерах организован наиболее простым способом, но он не масштабируется из-за множества условий `if ... else` и операторов `switch`.

Есть много альтернативных решений – например, с использованием внешней библиотеки. Некоторые из внешних библиотек позволяют управлять процессом выполнения чат-бота, например библиотека Dialogue Builder, реализованная поверх Claudia Bot Builder. За дополнительной информацией об этой библиотеке обращайтесь по адресу: https://github.com/nbransby/dialogue-builder.

Другой вариант – использовать средства обработки естественного языка (NLP). Создание библиотеки NLP – непростая задача, но, к счастью, существует множество доступных решений NLP, и некоторые из них относительно дешевы. Внедрив NLP, можно организовать свой код вокруг различных сущностей и действий вместо повторяющихся инструкций `if ... else` (которые можно рассматривать как своего рода маршрутизатор для диалогового интерфейса). Некоторые из библиотек NLP также имеют встроенное хранилище сеансов. Подробнее о NLP мы поговорим в разделе 9.6.

9.3. Подключение чат-бота к базе данных DynamoDB

Чтобы сделать наш чат-бот более полезным для клиентов, добавим в него сохранение заказов на пиццу в DynamoDB-таблицу `pizza-orders`.

Как показано на рис. 9.5, когда чат-бот получает сообщение, он должен подключиться к той же таблице в базе данных DynamoDB, которую использует Pizza API, и сохранить в ней принятый заказ. И только после этого вернуть сообщение, подтверждающее получение заказа.

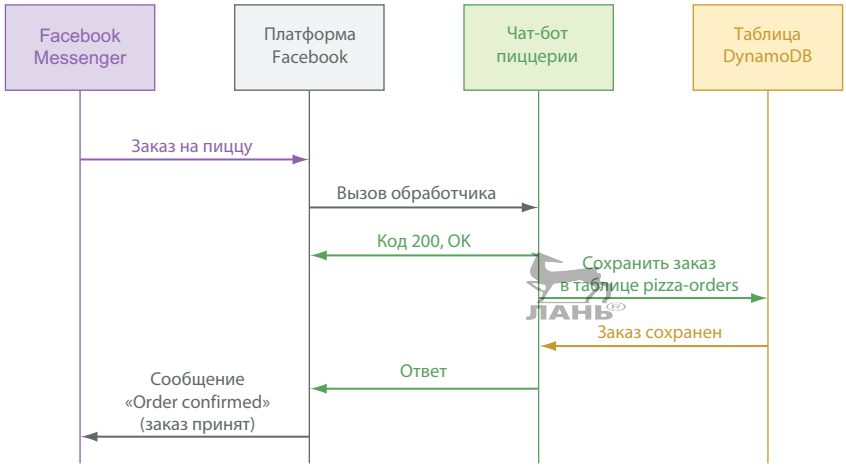


Рис. 9.5. Алгоритм работы чат-бота, подключенного к базе данных DynamoDB

Для простоты в этой главе мы покажем только, как сохранить заказ в таблице. В настоящем чат-боте также желательно дать пользователю возможность просмотреть его текущие заказы и отменить их.

Чтобы сохранить заказ, нужно внести несколько изменений в обработчик `order-pizza.js`.

Во-первых, нужно с помощью `DocumentClient` подключиться к DynamoDB. Для этого мы должны установить модуль `aws-sdk` из NPM как зависимость (или как необязательную зависимость, чтобы оптимизировать скорость развертывания). Затем импортировать `aws-sdk` и создать экземпляр `DocumentClient`, как показано ниже:

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
```

Далее следует вызвать метод `docClient.put`, чтобы сохранить заказ в таблице DynamoDB.

Основное отличие чат-бота от Pizza API – отсутствие адреса доставки на момент отправки запроса с заказом. Это означает, что мы должны записать неполные данные в таблицу DynamoDB, а затем запросить у пользователя адрес. Платформа Facebook Messenger не сохраняет состояния между последовательными сообщениями, поэтому в таблице `pizza-orders` или в какой-то другой (в этом разделе мы выбрали первый вариант) следует сохранить признак неполного заказа с некоторыми дополнительными параметрами.

По той же причине мы не можем использовать идентификатор заказа, возвращаемый службой доставки Some Like It Hot, а значит, нам вновь придется прибегнуть к помощи модуля `uuid`.

В DynamoDB мы сохраним следующие данные:

- `orderId` – идентификатор заказа, сгенерированный с помощью модуля `uuid`;

- `pizza` – идентификатор выбранной пиццы;
- `orderStatus` – в это поле мы запишем состояние заказа `in-progress` (выполняется), потому что заказ еще не выполнен;
- `platform` – укажем `fb-messenger-chatbot` в качестве идентификатора платформы, потому что в будущем, возможно, мы решим добавить поддержку других платформ обмена сообщениями;
- `user` – идентификатор пользователя, отправившего сообщение.

После записи заказа в таблицу мы запросим у пользователя адрес для доставки. Для этого можно послать простой вопрос, например: «Куда доставить вашу пиццу?» Мы реализуем эту операцию ниже.

Также мы должны предусмотреть обработку ошибок, чтобы иметь возможность послать пользователю сообщение, если что-то пойдет не так, и снова показать ему главное меню.

После внесения всех изменений содержимое файла `order-pizza.js` должно выглядеть, как показано в листинге 9.7.

Листинг 9.7. Обработчик заказа на пиццу с подключением к базе данных DynamoDB

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const pizzas = require('../data/pizzas.json')
const pizzaMenu = require('./pizza-menu')
const uuid = require('uuid/v4')

function orderPizza(pizzaId, sender) {
  const pizza = pizzas.find(pizza => pizza.id == pizzaId)

  return docClient.put({
    TableName: 'pizza-orders',
    Item: {
      orderId: uuid(),
      pizza: pizzaId,
      orderStatus: 'in-progress',
      platform: 'fb-messenger-chatbot',
      user: sender
    }
  }).promise()
    .then((res) => {
      return 'Where do you want your pizza to be delivered?'
    })
```

Импортировать AWS SDK.

Создать экземпляр DocumentClient.

Импортировать модуль uuid.

Сохранить заказ в таблице DynamoDB.

Использовать функцию uuid, чтобы сгенерировать уникальный идентификатор lzk заказа.

Установить статус заказа in-progress.

Сохранить идентификатор платформы, с помощью которой сделан заказ.

Сохранить идентификатор пользователя, сделавшего заказ.

Запросить у пользователя адрес доставки.

```
    .catch((err) => {
      console.log(err)
                                  В случае ошибки показать дружественное
                                  сообщение и повторно отправить главное меню.
      return [
        'Oh! Something went wrong. Can you please try again?',
        pizzaMenu()
      ]
    })
}

module.exports = orderPizza
```

Кроме обработчика `order-pizza.js`, также нужно изменить содержимое главного файла `bot.js`, добавив передачу идентификатора отправителя в функцию `orderPizza`. Идентификатор отправителя доступен в объекте `message` как `message.sender`; в случае с Facebook Messenger в этом атрибуте передается уникальный идентификатор пользователя, взаимодействующего со страницей мессенджера.

> **ПРИМЕЧАНИЕ.** Идентификатор пользователя, взаимодействующего со страницей Facebook Messenger (page-scoped user ID), отличается от обычного идентификатора пользователя Facebook, потому что Facebook стремится обеспечить конфиденциальность своих пользователей. Узнать больше об идентификаторах в Facebook Messenger можно по адресу: https://developers.facebook.com/docs/messenger-platform/identity.

В листинге 9.8 приводится только измененная часть файла `bot.js`. Этот код выполняется лишь при анализе значений, возвращаемых в ответах пользователей. Остальной код в файле остался без изменений.

Листинг 9.8. Изменившийся код в файле `bot.js`

```
if (values[0] === 'DETAILS') {
  return pizzaDetails(values[1])
} else if (values[0] === 'ORDER') {
  return orderPizza(values[1], message.sender)
}
```

Передать сообщение отправителя функции orderPizza во втором аргументе.

Теперь нужно создать политику, которая позволит пользователю, вызывающему функцию Lambda, взаимодействовать с базой данных DynamoDB. Создайте папку `roles` в папке проекта `pizza-fb-chatbot` и добавьте в нее файл `dynamodb.json`.

Как показано в листинге 9.9, файл `dynamodb.json` должен содержать разрешения, позволяющие пользователям сканировать, читать, добавлять и изменять элементы в DynamoDB. В настоящий момент чат-бот не поддерживает изме-

нения или отмены заказов, но мы должны добавить действие `dynamodb:UpdateItem`, потому что заказ будет изменяться (дополняться) после того, как пользователь пришлет свой адрес.

Листинг 9.9. Политика DynamoDB

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [                    ← Разрешить выполнять действия Scan, GetItem,
        "dynamodb:Scan",               PutItem и UpdateItem в DynamoDB.
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

Наконец, выполним команду `aws iam put-role-policy` в AWS CLI, чтобы добавить политику из файла `roles/dynamodb.json`. После этого можно заметить появление атрибута `role-name` в файле `claudia.json`. Как вы наверняка помните, при первом развертывании Claudia создает файл `claudia.json` в корневом каталоге проекта, куда сохраняет некоторые данные. Библиотека Claudia использует эти данные в последующих командах `update`, чтобы избавить нас от необходимости указывать дополнительные параметры. Файл `claudia.json` также хранит исполнитель Lambda, который нужен для вновь созданной политики. Загляните в этот файл и отыщите атрибут `role-name`.

Полная команда `aws iam put-role-policy` показана в листинге 9.10.

Листинг 9.10. Добавление политики доступа к DynamoDB для чат-бота пиццерии

```
aws iam put-role-policy \                          ← Имя роли для исполнителя Lambda.
  --role-name pizza-fb-chatbot-executor \
  --policy-name PizzaBotDynamoDB \      ← Имя политики.
  --policy-document file://./roles/dynamodb.json   ← Путь к файлу с определением политики.
```

В случае успешного завершения команда `aws iam put-role-policy` вернет пустой ответ. Теперь чат-бот готов к развертыванию.

> **ПРИМЕЧАНИЕ.** В случае неудачи команда `aws iam put-role-policy` вернет сообщение об ошибке, описывающее суть проблемы. Наиболее распространенные

ошибки: роль не существует или `policy-document` отсутствует в указанном файле. Если роль не существует, попробуйте снова запустить `claudia create` с уже упомянутыми параметрами. Если не найден `policy-document`, измените путь к файлу, чтобы команда `aws iam put-role-policy` смогла найти его.

Выполните команду `npm run update` или `claudia update`, чтобы развернуть чат-бот, и попробуйте послать ему сообщение.

СОВЕТ. С ростом кодовой базы частое развертывание может занимать довольно много времени и превратиться в утомительное занятие. Вы, наверное, заметили, что часто выполняете операцию развертывания в этой книге. Но Claudia предлагает один интересный трюк – она способна ускорить процесс развертывания. Чтобы воспользоваться этой возможностью, добавьте флаг `--no-option-dependencies` в сценарий `update`, который сообщает библиотеке Claudia, что та не должна развертывать любые необязательные зависимости, такие как AWS SDK, которые уже доступны в AWS Lambda:

```
"update": "claudia update --no-optional-dependencies"
```

9.4. Получение адреса доставки заказа в чат-боте

Как уже говорилось, после отправки заказа пользователем в нем отсутствует адрес доставки. В реальном проекте мы должны предусмотреть все необходимое, чтобы обеспечить надежную работу чат-бота, поэтому, возможно, имеет смысл добавить обработку естественного языка для распознавания адресов, присылаемых пользователями. Но, исключительно ради простоты, в этой главе мы используем кнопку, встроенную в Facebook Messenger, позволяющую сообщить свое местонахождение.

Facebook Messenger имеет замечательную функцию, дающую возможность пользователям сообщать о своем текущем местоположении одним нажатием кнопки. Эта кнопка посылает ответ с текущими географическими координатами пользователя. Claudia Bot Builder включает поддержку данной кнопки в `fbTemplate`. Добавить эту кнопку в запрос можно вызовом метода `.addQuickReplyLocation` класса `fbTemplate`.

Давайте изменим код обработчика `order-pizza.js`. Сначала импортируем `fbTemplate` из Claudia Bot Builder, добавив строку

```
const fbTemplate = require('claudia-bot-builder').fbTemplate
```

в начало файла.

Затем заменим ответ, где мы запрашиваем у пользователя его адрес, классом `fbTemplate.Text` и вызовом метода `.addQuickReplyLocation`, как показано в листинге 9.11.

Листинг 9.11. Запрос местоположения после сохранения заказа в таблице DynamoDB

```
.then((res) => {
  return new fbTemplate.Text('Where do you want your pizza to be delivered?')
    .addQuickReplyLocation()
    .get()
})
```

Создать экземпляр класса fbTemplate.Text.

Добавить кнопку, позволяющую сообщить текущее местоположение.

Преобразовать шаблон в формат JSON.

Когда клиент нажмет кнопку, чат-бот получит его текущие координаты: широту и долготу. Кроме обычного адреса, служба доставки Some Like It Hot принимает также географические координаты. (В реальном примере нам пришлось бы добавить кое-какую дополнительную информацию, например этаж, номер квартиры или хотя бы текстовое примечание для курьера.)

Для обработки координат создадим новую функцию-обработчик. С этой целью создайте файл `save-location.js` в папке `handlers`. Этот обработчик должен принимать параметры `userId` и `coordinates` и использовать их для изменения заказа в базе данных.

Чтобы получить возможность изменить заказ, мы должны импортировать AWS SDK, создать экземпляр `DocumentClient` и выполнить следующие действия:

- с помощью метода `DocumentClient.scan` найти в базе данных заказ со статусом `in-progress` (выполняется), принадлежащий данному клиенту;
- использовать метод `DocumentClient.update` и полученное значение `orderId`, чтобы изменить статус заказа.

Для примера изменим статус заказа на `pending` и добавим в него широту и долготу места доставки.

В листинге 9.12 показано, как должен выглядеть обработчик `save-location.js`.

Листинг 9.12. Обработчик сохранения координат клиента

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function saveLocation(userId, coordinates) {
  return docClient.scan({
    TableName: 'pizza-orders',
    Limit: 1,
    FilterExpression: `user = :u, orderStatus: :s`,
    ExpressionAttributeNames: {
      ':u': { S: userId },
      ':s': { S: 'in-progress' }
    }
```

Импортировать AWS SDK и создать экземпляр DocumentClient.

Определить функцию-обработчик, принимающую идентификатор и координаты пользователя.

Сканировать таблицу pizza-orders.

Ограничить количество результатов единственным элементом.

Искать только заказы, отправленные указанным пользователем и имеющие указанный статус.

Определить клиента (это отправитель) и статус (in-progress) для выражения фильтра.

```
  }).promise()
    .then((result) => result.Items[0])
    .then((order) => {
      const orderId = order.orderId
      return docClient.update({
        TableName: 'pizza-orders',
        Key: {
          orderId: orderId
        },
        UpdateExpression: 'set orderStatus = :s, coords=:c',
        ExpressionAttributeValues: {
          ':s': 'pending',
          ':c': coordinates
        },
        ReturnValues: 'ALL_NEW'
      }).promise()
    })
}

module.exports = saveLocation
```

Извлечь только первый элемент из ответа.

Запомнить идентификатор заказа в локальной переменной.

Изменить элемент в таблице pizza-orders.

Указать идентификатор заказа, который требуется изменить.

Определить выражение, осуществляющее изменение.

Определить значения для изменения.

Вернуть все измененные данные.

Экспортировать функцию-обработчик.

Наконец, мы должны изменить содержимое файла `bot.js`:

1) импортировать новый обработчик `save-location.js`;
2) вызвать новую функцию `saveLocation`, когда клиент пришлет свои координаты.

Чтобы импортировать новый обработчик `save-location.js`, добавьте следующий фрагмент в начало файла `bot.js` (например, после функции `pizzaMenu`):

```
const saveLocation = require('./handlers/save-location')
```

Прежде чем обрабатывать координаты клиента, сначала нужно убедиться, что сообщение не является ответом (`postback == false`). Затем извлечь координаты из `message.originalRequest`, если они существуют. Координаты передаются как вложение, поэтому они доступны как объект `message.originalRequest.message.attachments[0].payload.coordinates`.

В листинге 9.13 приводится несколько последних строк из файла `bot.js`.

Листинг 9.13. Обработка координат в главном файле чат-бота

```
if (
    message.originalRequest.message.attachments &&
    message.originalRequest.message.attachments.length &&
    message.originalRequest.message.attachments[0].payload.coordinates &&
    message.originalRequest.message.attachments[0].payload.coordinates.lat &&
```

Проверить, прислал ли клиент свои координаты.

```
      message.originalRequest.message.attachments[0].payload.coordinates.long
  ) {
    return saveLocation(message.sender, message.originalRequest.message.
      attachments[0].payload.coordinates)
  }

  return [
    `Hello, here's our pizza menu:`,
    pizzaMenu()
  ]
}, {
  platforms: ['facebook']
})

module.exports = api
```

Вызвать функцию saveLocation и передать ей идентификатор отправителя и координаты.

Изменив содержимое файла `bot.js`, разверните чат-бот командой `npm run update` и опробуйте его. Результат должен выглядеть, как показано на рис. 9.6.

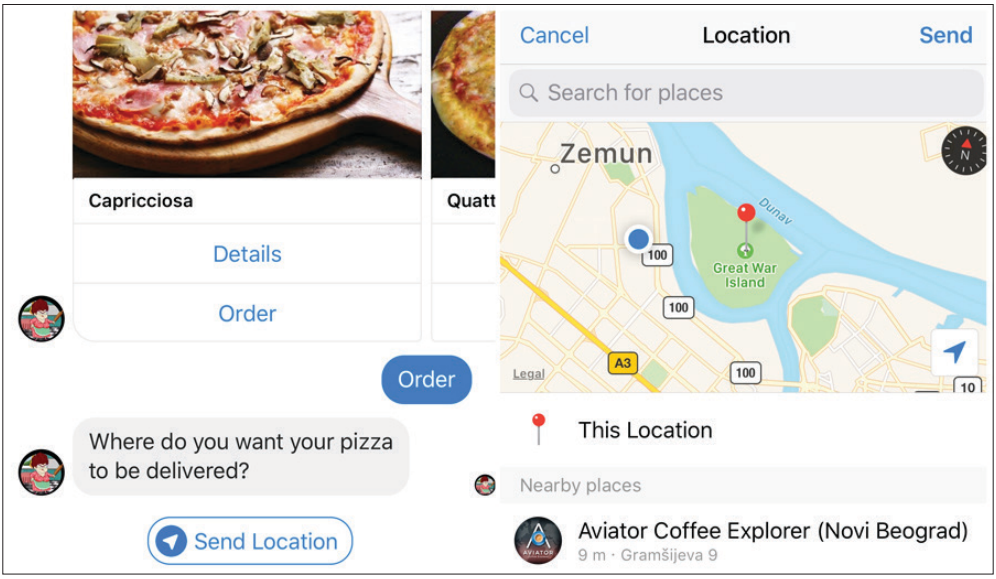


Рис. 9.6. Чат-бот, позволяющий клиенту сообщить свои текущие координаты

9.5. Планирование доставки

Последний фрагмент мозаики, который поможет «сделать чат-бот полезным», – подключение его к API службы доставки Some Like It Hot. Как показано на рис. 9.7, после интеграции с этим API порядок работы чат-бота должен выглядеть следующим образом:

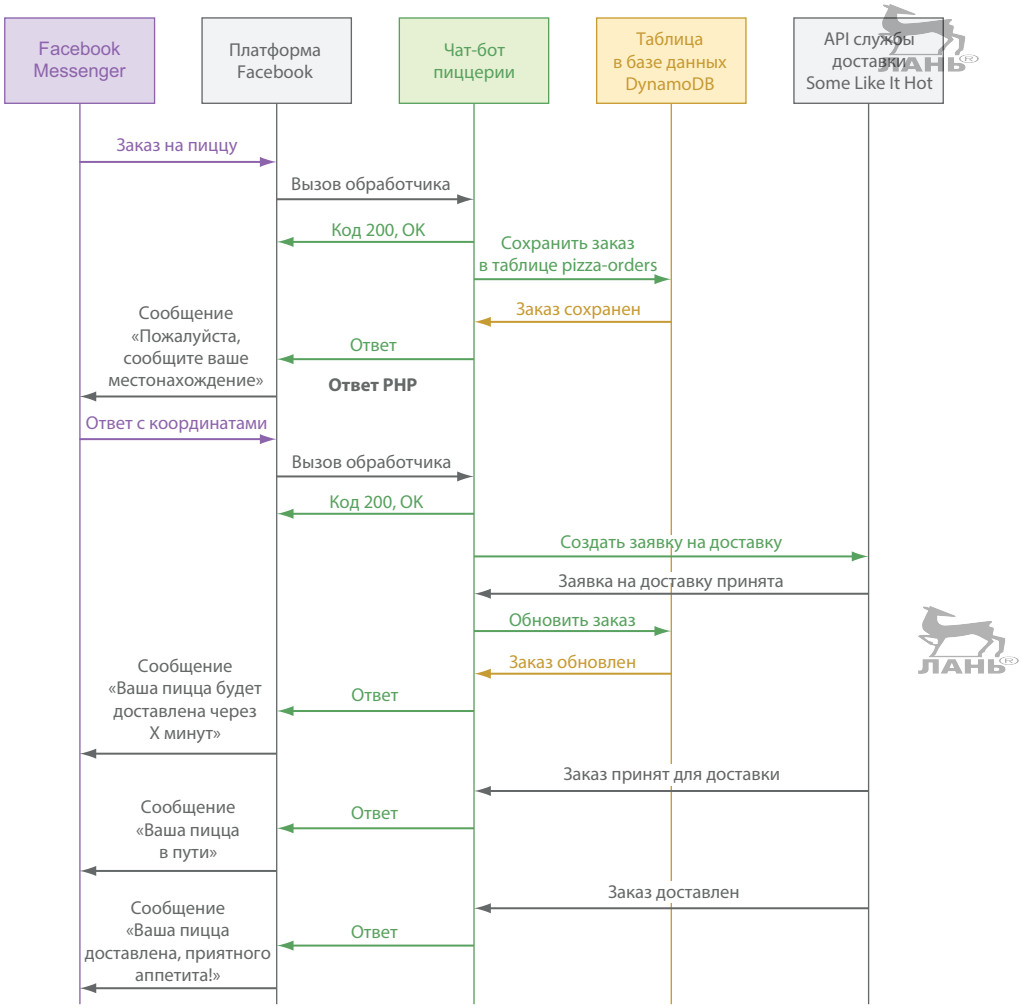


Рис. 9.7. Порядок работы чат-бота, определяющего местонахождение клиента и взаимодействующего со службой доставки Some Like It Hot

1) клиент касается кнопки **Order** (Заказать), чтобы заказать пиццу;
2) заказ сохраняется в базе данных со статусом `in-progress` (выполняется);
3) чат-бот запрашивает у клиента его текущее местонахождение;
4) клиент сообщает свои координаты;
5) чат-бот подключается к API службы доставки;
6) заявка на доставку принимается, и чат-бот обновляет состояние заказа в базе данных и уведомляет клиента;
7) после передачи заказа службе доставки ее API вызывает обработчик в чат-боте;
8) чат-бот посылает уведомление клиенту;

9) после вручения заказа клиенту API службы доставки вновь вызывает обработчик в чат-боте входа;
10) чат-бот посылает заключительное сообщение клиенту.

Как можно заметить на рис. 9.7, мы должны внести в чат-бот два изменения:

- добавить в обработчик `save-location.js` отправку заявки на доставку;
- создать новую точку входа, которой будет пользоваться API службы доставки.

Начнем с самого простого – с обработчика `save-location.js`. Изменения в этом обработчике напоминают код, который мы добавляли в главе 4, когда связывали Pizza API со службой доставки. Мы должны послать запрос `POST` по адресу https://some-like-it-hot-api.effortless-serverless.com/delivery. Единственное отличие – мы должны послать `deliveryCoords` вместо `deliveryAddress`.

Другое важное отличие – мы не можем изменить первичный ключ заказа. Так как мы не можем использовать `deliveryId` в роли `orderId`, то должны сохранить идентификатор заявки на доставку в таблице DynamoDB. Как вы, наверное, помните из главы 3, мы использовали `deliveryId` в роли `orderId`, чтобы увеличить эффективность поиска заказов для изменения состояния доставки.

В листинге 9.14 показана измененная версия обработчика `save-location.js`, в которую добавлено взаимодействие с API службы доставки Some Like It Hot.

Листинг 9.14. Обработчик save-location.js

```
// Первая половина файла осталась прежней
    .then((result) => result.Items[0])
    .then((order) => {
      return rp.post('https://some-like-it-hot-api.effortless-serverless.com/
  delivery', {
        headers: {
          "Authorization": "aunt-marias-pizzeria-1234567890",
          "Content-type": "application/json"
        },
        body: JSON.stringify({
          pickupTime: '15.34pm',
          pickupAddress: 'Aunt Maria's Pizzeria',
          deliveryCoords: coordinates,
          webhookUrl: 'https://g8fhlgccof.execute-api.eu-central-1.amazonaws.
    com/latest/delivery',
        })
      })
```

Послать POST-запрос в API службы доставки.

Добавить заголовки в запрос, включая заголовок Authorization с ключом авторизации.

Преобразовать тело запроса в строку.

Заполнитель для времени.

Добавить pickupTime, pickupAddress и deliveryCoords в тело.

Передать URL обработчика для вызова службой доставки.

```
      .then(rawResponse => JSON.parse(rawResponse.body))
      .then((response) => {
        order.deliveryId = response.deliveryId
        return order
      })
    })
    .then((order) => {
      return docClient.update({
        TableName: 'pizza-orders',
        Key: {
          orderId: order.orderId
        },
        UpdateExpression: 'set orderStatus = :s, coords=:c, deliveryId=:d',
        ExpressionAttributeValues: {
          ':s': 'pending',
          ':c': coordinates,
          ':d': order.deliveryId
        },
        ReturnValues: 'ALL_NEW'
      }).promise()
    })
}

module.exports = saveLocation
```

Выполнить парсинг строкового тела ответа.

Добавить идентификатор заявки на доставку в объект order.

Вернуть объект order для правильной работы цепочки объектов Promise.

Сохранить данные в таблицу DynamoDB.

Сохранить идентификатор заявки на доставку в таблицу DynamoDB.

Теперь, сохранив идентификатор заявки на доставку в DynamoDB, мы должны создать точку входа для Some Like It Hot Delivery API. Но как добавить маршрут в чат-бот?

Как отмечалось выше, Claudia Bot Builder экспортирует экземпляр Claudia API Builder. Это означает, что функция `botBuilder` в нашем файле `bot.js` возвращает полноценный экземпляр Claudia API Builder.

Перед добавлением нового маршрута нужно создать новую функцию-обработчик для него. Для этого создайте в папке `handlers` файл с именем `delivery-webhook.js`. Внутри этого обработчика нужно найти заказ по идентификатору заявки на доставку, который передаст API службы доставки, затем изменить статус в этом заказе и послать сообщение клиенту, чтобы уведомить его об изменении статуса заказа. Порядок работы этого обработчика показан на рис. 9.8.

Поиск и изменение заказа выполняются точно так же, как в обработчике `save-location.js`. Единственная сложность – отправка сообщения клиенту.

Чтобы послать сообщение в Facebook Messenger, нужно отправить запрос платформе Facebook Messenger. В каждом таком запросе требуется указать идентификатор пользователя, текст сообщения и ключ доступа к Facebook Messenger.

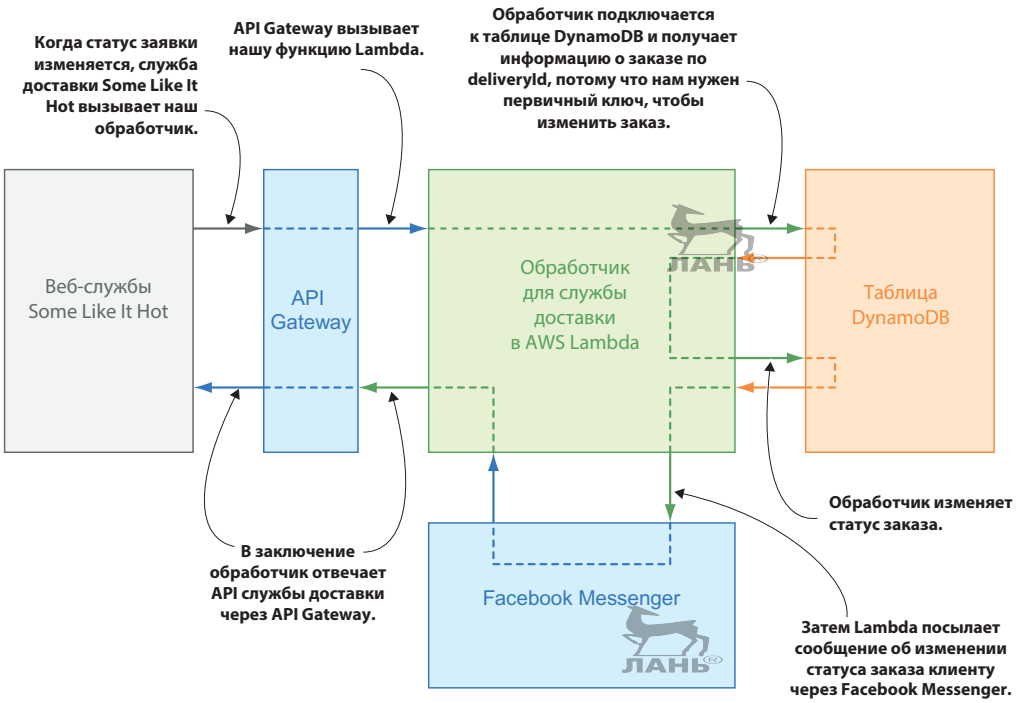


Рис. 9.8. Порядок работы обработчика для службы доставки

Послать подобный запрос можно точно так же, как в API службы доставки, или можно воспользоваться библиотекой `claudia-bot-builder`. Это внутренняя библиотека, но ее можно подключить, импортировав файл `reply.js`:

```
const reply = require('claudia-bot-builder/lib/facebook/reply')
```

В этом случае импортируется только файл `reply.js`, а не весь пакет Claudia Bot Builder, и функция сохраняется в константе `reply`.

Эта функция `reply` принимает три параметра: идентификатор отправителя, текст сообщения и ключ доступа.

Идентификатор отправителя можно получить из базы данных. Текст сообщения – это простой текст в стандартном формате Claudia Bot Builder, который нужно послать клиенту. Можно послать один объект шаблона или массив из нескольких сообщений. Наконец, ключ доступа к Facebook Messenger можно получить из объекта `request.env`. Вы можете передать ключ в функцию-обработчик во втором аргументе.

> **ПРИМЕЧАНИЕ.** Facebook Messenger накладывает определенные ограничения на отправку сообщений: вы не сможете послать сообщение клиенту иначе, как в ответ на его сообщение, а кроме того, ответ можно послать только в течение 24 часов. Узнать больше об ограничениях можно по адресу: https://developers.facebook.com/docs/messenger-platform/send-messages#messaging_types.

В листинге 9.15 показан полный код обработчика сообщений от службы доставки.

Листинг 9.15. Обработчик сообщений от службы доставки

```
'use strict'

const reply = require('claudia-bot-builder/lib/facebook/reply')

function deliveryWebhook(request, facebookAccessToken) {
  if (!request.deliveryId || !request.status)
    throw new Error('Status and delivery ID are required')

  return docClient.scan({
    TableName: 'pizza-orders',
    Limit: 1,
    FilterExpression: `deliveryId = :d`,
    ExpressionAttributeNames: {
      ':d': { S: deliveryId }
    }
  }).promise()
    .then((result) => result.Items[0])
    .then((order) => {
      return docClient.update({
        TableName: 'pizza-orders',
        Key: {
          orderId: order.orderId
        },
        UpdateExpression: 'set orderStatus = :s',
        ExpressionAttributeValues: {
          ':s': request.status
        },
        ReturnValues: 'ALL_NEW'
      }).promise()
    })
    .then((order) => {
      return reply(order.user, `The status of your delivery is updated to:
      ${order.status}.`, facebookAccessToken)
    })
}

module.exports = deliveryWebhook
```

Импортировать функцию reply из Claudia Bot Builder.

Проверить запрос.

Определение функции-обработчика deliveryWebhook, принимающей объект request и ключ доступа.

Найти в таблице DynamoDB запись с данным идентификатором заявки.

Извлечь из массива с результатами только первый элемент.

Изменить статус заказа в таблице DynamoDB.

Ответить клиенту, указав идентификатор пользователя, сообщение и ключ доступа к Facebook Messenger.

Экспортировать функцию-обработчик.

Теперь добавим маршрут к этому обработчику в файл `bot.js`. Для этого импортируем обработчик `delivery-webhook.js` в начале файла:

```
const deliveryWebhook = require('./handlers/delivery-webhook')
```

Затем добавим новый маршрут `POST /delivery` в конец файла, непосредственно перед инструкцией `module.exports = api`. Этот маршрут будет вызывать функцию-обработчик `deliveryWebhook` с телом запроса и ключом доступа к Facebook Messenger и в случае успеха возвращать код 200 или в случае ошибки – код 400.

В листинге 9.16 показано несколько последних строк из файла `bot.js`.

Листинг 9.16. Маршрут к обработчику запросов со стороны службы доставки

```
  return [
    `Hello, here's our pizza menu:`,
    pizzaMenu()
  ]
}, {
  platforms: ['facebook']
})

api.post('/delivery', (request) => deliveryWebhook(request.body, request.env.
      facebookAccessToken), {
  success: 200,
  error: 400
})

module.exports = api
```

Определить маршрут POST /delivery и вызвать обработчик deliveryWebhook.

Вернуть код 200 в случае успешной обработки запроса.

Вернуть код 400 в случае ошибки.

Теперь разверните чат-бот командой `npm run update` или `claudia update` и убедитесь, что он выполняет все функции.

9.6. Добавление простой обработки естественного языка

Для создания более сложного чат-бота не обойтись без реализации обработки сообщений на естественном языке (NLP). Создание библиотеки NLP «с нуля» – сложная задача, которая под силу далеко не всем. Но, к счастью, существует большое количество библиотек, которые вы с успехом сможете использовать для совершенствования своих чат-ботов. Например:

- Wit.ai (https://wit.ai) – разработка Facebook; предлагает средства преобразования естественного языка (речи или текста) в данные, пригодные для обработки;
- DialogFlow (прежде называлась API.ai; https://dialogflow.com) – разработка Google; фактически является диалоговой платформой, обеспечиваю-

щей взаимодействие с устройствами, приложениями и службами на естественном языке;

- IBM Watson (https://www.ibm.com/watson/) – суперкомпьютер IBM, сочетающий в себе искусственный интеллект и сложное аналитическое программное обеспечение, который предлагает инструменты для создания машины типа «вопрос–ответ». Кроме того, Watson предлагает инструменты для продвинутого анализа текста.

Библиотеки Wit.ai и DialogFlow можно использовать бесплатно, хотя и с некоторыми ограничениями; IBM Watson предоставляет бесплатный пробный период.

Интеграция этих библиотек в чат-боты не вызывает сложностей. Все они хороши, и все можно рекомендовать к использованию на практике, но каждая из них имеет свои сильные и слабые стороны, которые, впрочем, не особенно важны для этой книги. Claudia Bot Builder не ограничивает и никак не мешает использованию этих библиотек.

В Facebook Messenger тоже есть встроенная библиотека NLP, но, к сожалению, она предлагает только самые простые инструменты. С ее помощью можно научить чат-бот распознавать приветствия, благодарности и прощания, определять даты, время, координаты, денежные суммы, номера телефонов и адреса электронной почты. Например, фраза «tomorrow at 2pm» (завтра в 2 часа дня) будет преобразована в значение времени.

> **ПРИМЕЧАНИЕ.** За дополнительной информацией о библиотеке NLP, встроенной в Facebook Messenger, обращайтесь по адресу: https://developers.facebook.com/docs/messenger-platform/built-in-nlp.

Несмотря на ограниченные возможности, библиотека NLP, встроенная в Facebook Messenger, включает все необходимое, чтобы позволить клиентам заказать пиццу на определенное время или день. Поскольку это и без того длинная глава, мы используем данную библиотеку, чтобы чат-бот мог ответить на «спасибо».

Для этого нужно следующее:

1) установить и настроить встроенную библиотеку NLP, как описывается в приложении B;
2) добавить в файл `bot.js` проверку, является ли сообщение ответом клиента. Если нет, мы не будем использовать средства NLP в главном меню;
3) если сообщение является ответом клиента, используем библиотеку NLP, чтобы распознать выражение благодарности в тексте. Если благодарность имеет место, ответим сообщением «You're welcome!» («Пожалуйста!»); иначе покажем главное меню.

Встроенная библиотека NLP добавляет опознанные сущности в атрибут `nlp` объекта `message`. Сущности возвращаются в виде массива, и каждая сущность

имеет оценку достоверности в атрибуте `confidence` и значение в атрибуте `value`. Оценка достоверности определяет степень уверенности парсера в опознании (вероятность, что сущность опознана верно) и имеет значение в диапазоне от 0 до 1. Атрибут `value` определяет значение опознанной сущности. В случае с сущностью «thanks» («спасибо»), если она присутствует в сообщении, этот атрибут всегда будет иметь значение `true`. Мы проверим существование сущности «thanks», и если ее оценка уверенности превышает 0,8 (80 %), вернем сообщение «You're welcome!» («Пожалуйста!»).

В листинге 9.17 показаны последние несколько строк в измененном файле `bot.js`.

Листинг 9.17. Ответ на сообщение с благодарностью

```
  if (
    message.originalRequest.message.nlp &&
    message.originalRequest.message.nlp.entities &&
    message.originalRequest.message.nlp.entities['thanks'] &&
    message.originalRequest.message.nlp.entities['thanks'].length &&
    message.originalRequest.message.nlp.entities['thanks'][0].confidence > 0.8
  ) {
    return `You're welcome!`
  }

  return [
    `Hello, here's our pizza menu:`,
    pizzaMenu()
  ]
}, {
  platforms: ['facebook']
})

module.exports = api
```

Если в сообщении присутствует атрибут nlp и сущность «thanks», ответить на выражение благодарности.

Если атрибут nlp или сущность «thanks» отсутствует, вернуть главное меню.

9.7. Опробование!

Сделать чат-бот чуть более интерактивным и чуть более интеллектуальным совсем несложно, но чтобы клиенты испытывали удовольствие от общения с ним, он должен быстро и эффективно удовлетворять их потребности.

9.7.1. Упражнение

Ваша основная задача в этом упражнении – показать каждому клиенту тетушки Марии его последний заказ в приветственном сообщении. Выполняя заказ в сети ресторанов, клиенты часто заказывают одну и ту же еду. В этом упражнении вы должны поприветствовать клиента и напомнить ему о его по-

следнем заказе. Если вы чувствуете, что способны на большее, после решения основного упражнения мы предложим вам решить более сложную задачу.

9.7.2. Решение

Это упражнение решается просто. Нужно просмотреть список заказов, найти последний заказ клиента по идентификатору отправителя в объекте `message` и вернуть название последней заказанной им пиццы, выразив надежду, что она ему понравилась.

Листинг 9.18. Реализация приветствия клиента в главном файле чат-бота

```
'use strict'

const botBuilder = require('claudia-bot-builder')

const pizzaDetails = require('./handlers/pizza-details')
const orderPizza = require('./handlers/order-pizza')
const pizzaMenu = require('./handlers/pizza-menu')
const saveLocation = require('./handlers/save-location')
const getLastPizza = require('./handlers/get-last-pizza')

const api = botBuilder((message) => {
  if (message.postback) {
    const values = message.text.split('|')

    if (values[0] === 'DETAILS') {
      return pizzaDetails(values[1])
    } else if (values[0] === 'ORDER') {
      return orderPizza(values[1], message)
    }
  }

  if (
      message.originalRequest.message.attachments &&
      message.originalRequest.message.attachments.length &&
      message.originalRequest.message.attachments[0].payload.coordinates &&
      message.originalRequest.message.attachments[0].payload.coordinates.lat
      &&
      message.originalRequest.message.attachments[0].payload.coordinates.long
  ) {
    return saveLocation()
  }

  return getLastPizza().then((lastPizza) => {
```

Импортировать модуль get-last-pizza.

Вызвать функцию из модуля и получить информацию о пицце, заказанной клиентом в прошлый раз.

```
    let lastPizzaText = lastPizza ? `Glad to have you back! Hope you liked
      your ${lastPizza} pizza` : ''
    return [
      `Hello, ${lastPizzaText} here's our pizza menu:`,
      pizzaMenu()
    ]
  })
}, {
  platforms: ['facebook']
})

module.exports = api
```

Если раньше клиент уже заказывал пиццу, сконструировать текст сообщения с названием пиццы.

Вернуть текст приветствия.

Изменений в главном файле чат-бота немного, потому что основная логика решения находится в новом файле `get-last-pizza.js`.

Эта логика приводится в листинге 9.19.

Листинг 9.19. Обработчик, возвращающий пиццу, которую заказывал клиент в прошлый раз

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
const pizzaMenu = require('./pizza-menu')
const pizzas = require('../data/pizzas.json')

function getLastPizza(sender) {

  return docClient.scan({
    TableName: 'pizza-orders',
    ScanIndexForward: false,
    Limit: 1,
    FilterExpression: `sender = #{sender}`,
  }).promise()
    .then((lastPizzaOrder) => {
      let lastPizza
      if (lastPizzaOrder){
        lastPizza = pizzas.find(pizza => pizza.id == lastPizzaOrder.pizzaId)
      }
      return lastPizza
    })
    .catch((err) => {
      console.log(err)
```

Импортировать AWS SDK.

Создать экземпляр DocumentClient.

Получить список пицц.

Сканировать таблицу pizza-orders.

Сформировать операцию сканирования, чтобы отыскать последний заказ, то есть последнюю запись в базе данных.

Ограничить операцию сканирования единственным результатом.

Проверять только записи, принадлежащие указанному клиенту.

Вернуть последний найденный заказ.

Сбросить значение переменной lastPizza.

Проверить, имелся ли заказ в прошлом.

Найти последнюю заказанную пиццу по ее идентификатору из заказа.

Вернуть последнюю заказанную пиццу.

```
      return [
        'Oh! Something went wrong. Can you please try again?',
        pizzaMenu()
      ]
    })
}

module.exports = getLastPizza
```

В случае ошибки вернуть дружественное сообщение и меню выбора пиццы.

ПРИМЕЧАНИЕ. Это решение основного упражнения для данной главы. Для усложненного упражнения мы не предлагаем готового решения, чтобы сделать данное упражнение более интересным.

9.7.3. Усложненное задание

Наиболее азартным читателям мы предлагаем решить более сложную задачу. Это упражнение послужит вам хорошим испытанием. Его основная цель – дать клиенту возможность повторить последний заказ. В начальном приветствии, если клиент ранее уже заказывал пиццу, вы должны предложить повторить предыдущий заказ и предоставить две дополнительные кнопки для быстрого ответа. Если клиент нажмет кнопку «Да, повторить заказ», вы должны оформить заказ на ту же пиццу с тем же адресом доставки. Если клиент нажмет кнопку «Нет, покажите мне меню», вы должны вывести список доступных пицц.

В заключение

- Для анализа сообщений, возвращаемых клиентом в ответ на вопрос чат-бота, исследуются поля `message.text` и `message.postback`. Если сообщение действительно является ответом, поле `message.postback` будет содержать значение `true`.
- Логику работы чат-бота желательно разбить на более мелкие задачи, реализованные в разных файлах, вместо использования простой условной инструкции `if...else`.
- Чат-боты, реализованные с применением Claudia Bot Builder, можно подключать к базе данных DynamoDB с помощью `DocumentClient`.
- Чтобы узнать текущее местоположение клиента, можно воспользоваться шаблоном быстрого ответа.

Глава 10

Джарвис, то есть Алекса, закажи мне пиццу

Эта глава охватывает следующие темы:

- создание бессерверного SMS-чат-бота;
- проблемы несовместимости разных бессерверных чат-ботов;
- использование голосового помощника Алекса с помощью Claudia и AWS Lambda.

Чат-боты выгодны для бизнеса, потому что значительно снижают потребность в поддержке клиентов, позволяя им самим взаимодействовать с вашими приложениями удобным и интересным способом. Бессерверные чат-боты дают дополнительные выгоды, помогая справляться с большим наплывом клиентов без дополнительной настройки сервера. Единственное ограничение чат-ботов – они привязаны к соответствующим платформам обмена сообщениями, которые сильно отличаются друг от друга. Например, Facebook существует уже более 10 лет, но значительный процент людей по-прежнему не пользуется им, а они тоже могут быть вашими клиентами. Как решить эту проблему?

С другой стороны, компьютеры все глубже проникают в нашу жизнь, и в последнее время наблюдается бурный рост числа голосовых помощников, таких как Apple Siri, Amazon Alexa, Google Home, Microsoft Cortana и многих других. Теперь не требуется писать текст и посылать его чат-боту – можно просто поговорить с ним. И эта технология широко используется нашими клиентами из другой категории: технически подкованных, легко перенимающих и продвигающих новые технологии. Чтобы охватить эти две категории потребителей, недостаточно написать одного чат-бота. В этой главе мы покажем, как удовлетворить нужды особенно продвинутых клиентов, создав чат-боты для SMS (Short Message Service – служба коротких сообщений) и голосового помощника Amazon Alexa в виде бессерверных служб на основе Claudia.js.

10.1. Не могу сейчас говорить: отправка SMS с помощью службы Twilio

Бизнес тетушки Марии начал набирать обороты, и это хорошая новость! Пьер, ее разработчик мобильного приложения, сообщил об увеличении числа загрузок приложения, а школьные друзья Джулии распространили новость о появлении чат-бота в Facebook, в результате чего число заказов значительно увеличилось. Тетушка Мария счастлива, потому что мы помогли ее бизнесу прочно встать на ноги, и она пригласила нас на бесплатный обед, чтобы встретиться с ней и дядюшкой Фрэнком.

Дядюшка Фрэнк – брат Марии – немолодой, невысокий и грузный мужчина, обычно ходит в темной рубашке с рукавами, закатанными до локтей. Он владеет известным баром, расположенным на той же улице, недалеко от пиццерии. Он любит вкусно поесть и часто звонит Марии, чтобы заказать пиццу для себя или своих клиентов. Но он человек старой закалки и не спешит осваивать новые технологии.

Мы пришли в пиццерию на встречу с тетушкой Марией и дядюшкой Фрэнком. Они довольны, и дядюшка Фрэнк поздравляет нас. Он наслышан о наших успехах, особенно с чат-ботом в Facebook Messenger. Но в ходе праздничного обеда мы начинаем понимать, что бесплатный сыр бывает только в мышеловке. Тетушка Мария и дядюшка Фрэнк, рассыпаясь в благодарностях за нашу большую работу по привлечению молодежи, выражают пожелание привлечь людей старших поколений, таких как клиенты и друзья дядюшки Фрэнка, не имеющих учетной записи в Facebook. Некоторые из них вообще не имеют учетных записей в социальных сетях. Пиццерия тетушки Марии в настоящее время не может нанять новых работников, чтобы отвечать на звонки, поэтому они спрашивают, сможем ли мы создать SMS-чат-бота. Все ее клиенты владеют мобильными телефонами и знают, как отправлять текстовые сообщения, поэтому такой чат-бот мог бы стать хорошим решением. Но с чего начать?

В настоящее время доступно множество облачных платформ для обмена сообщениями, но одной из самых известных и широко используемых является Twilio. Она позволяет клиентам совершать и принимать телефонные звонки и текстовые сообщения с использованием ее API.

> **ПРИМЕЧАНИЕ.** В этой главе рассматривается только возможность отправки текстовых сообщений (SMS) в Twilio. Телефонные звонки выходят далеко за рамки этой книги. Узнать больше о Twilio можно на сайте http://twilio.com.

К счастью, Claudia Bot Builder поддерживает SMS-чат-боты Twilio. Настроить чат-бот для Twilio так же просто, как чат-бот для Facebook. Для начала мы реализуем SMS-чат-бот, приветствующий клиента пиццерии тетушки Марии, чтобы усвоить основные идеи, а затем реализуем отправку полного списка пицц и оформление заказа.

Прежде всего создайте отдельную папку `sms-chatbot` для проекта. Перейдите в нее и создайте файл `sms-bot.js`.

> **ПРИМЕЧАНИЕ.** Может показаться странным, почему мы создаем отдельный чат-бот. Неужели в нем не будет логики, повторяющей логику чат-бота для Facebook? На то есть две причины. Во-первых, SMS-чат-бот существенно отличается от чат-бота для Facebook. В нем не будет интерактивных кнопок, только простые текстовые сообщения, поэтому повторно использовать ту же логику будет проблематично. Другая причина: мы хотели бы, чтобы наши службы были независимыми и более простыми в обслуживании. Наличие двух чат-ботов увеличит сложность всего проекта и сделает его обслуживание менее удобным. Обновление одного может повлиять на работу другого. Разделение служб также означает, что SMS-чат-бот будет находиться в другой функции Lambda. Если бы они оба находились в одной функции Lambda и чат-бот для Facebook потерпел крах, тогда SMS-чат-бот тоже прекратил бы работу.

Итак, выше мы решили написать простой чат-бот, возвращающий простое приветствие, например: «Hello from Aunt Maria's pizzeria!» («Вас приветствует пиццерия тетушки Марии!»). Чтобы реализовать его, сначала импортируем модуль Claudia Bot Builder, который поможет нам создать чат-бота. Затем создадим программный интерфейс чат-бота, который будет использовать функцию обратного вызова для обработки сообщений. Внутри этой функции мы будем получать однострочный текст `"Hello from Aunt Maria's pizzeria!"`. После определения функции мы должны определить объект с атрибутом `platforms`, содержащим массив платформ, поддерживаемых чат-ботом. Поскольку мы должны поддерживать только Twilio, добавим в массив единственную строку `'twilio'`. По окончании наш файл `sms-bot.js` должен выглядеть, как показано в листинге 10.1.

Листинг 10.1. Простой SMS-чат-бот, возвращающий приветствие

```
'use strict'

const botBuilder = require('claudia-bot-builder')

const api = botBuilder(() => {
  return `Hello from Aunt Maria's pizzeria!`
}, { platforms: ['twilio'] })

module.exports = api
```

Импортировать модуль Claudia Bot Builder.

Подготовить функцию-обработчик для Claudia Bot Builder function и сохранить экземпляр Claudia API Builder.

Вернуть простой текст.

Объявить о поддержке платформы Twilio.

Экспортировать экземпляр Claudia API Builder.

Код довольно прост, но, прежде чем увидеть его в действии, необходимо создать учетную запись Twilio и указать номер телефона, с которого можно отправлять и получать SMS-сообщения. После этого нужно настроить услугу

программируемых SMS в панели управления Twilio и назначить ей этот номер телефона. Инструкции по созданию и настройке учетной записи Twilio вы найдете в приложении В.

После этой настройки нужно с помощью Claudia создать новую функцию в AWS Lambda и развернуть в ней SMS-чат-бот. Для этого выполните следующую команду: `claudia create --region <ваш-регион> --api-module sms-bot`.

Как вы наверняка помните, эта команда вернет URL вновь созданного чат-бота. Он должен заканчиваться на `/twilio`. Скопируйте этот URL, откройте страницу службы программируемых SMS-сообщений Twilio и вставьте ее в поле **Inbound URL** (Входящий URL). Не забудьте сохранить новую конфигурацию службы программируемых SMS.

Последний шаг, который нужно выполнить перед опробованием SMS-чат-бота, – выполнить команду `claudia update --configure-twilio-sms-bot`. Эта команда настроит Twilio как платформу для нашего чат-бота. Вот и все!

Теперь попробуйте послать сообщение «Hello» на номер телефона, настроенный в Twilio.

> **ПРИМЕЧАНИЕ.** Если мобильная сеть сильно загружена, до получения ответа от SMS-чат-бота может пройти до 30 секунд.

10.1.1. Список пицц в SMS

В двух предыдущих главах наш чат-бот для Facebook сначала возвращал приветственное сообщение клиентам. А когда клиент запрашивал меню, чат-бот отображал список пицц с горизонтальной прокруткой. Клиент выбирал нужную ему пиццу и запускал процесс оформления заказа.

Эта модель выглядит вполне пригодной и для других чат-платформ, но в SMS используется иной протокол связи, не поддерживающий отправку изображений. Поэтому мы должны отправить список пицц в текстовом виде, явно указав, каким должен быть ответ для заказа той или иной пиццы. Например, чтобы заказать пиццу с грибами, нужно вернуть ответ `FUNGHI`.

> **ПРИМЕЧАНИЕ.** Мы не можем отправлять изображения в SMS-сообщениях; для этого нужно использовать службу мультимедийных сообщений (Multimedia Messaging Service, MMS). Twilio поддерживает MMS-сообщения, но ограничивается телефонными номерами в Соединенных Штатах и Канаде. Кроме того, поддержка MMS не рассматривается в этой книге. Узнать больше о поддержке MMS в Twilio можно на сайте https://www.twilio.com/mms.

Иногда отправка SMS сопряжена с определенными расходами. Причем в некоторых странах это довольно дорогостоящая услуга. Если вы ожидаете, что этим чат-ботом будут пользоваться тысячи клиентов, расходы на сообщения могут быстро возрасти. Поэтому старайтесь минимизировать количество отправляемых SMS, но не в ущерб ясности диалога для клиента.

В нашем случае SMS-чат-бота для пиццерии подразумевается строгое выполнение определенной последовательности действий. Например, в начале диалога следует совместить приветствие со списком предлагаемых пицц в одном сообщении. Это довольно удобно для клиента. Для этого мы должны просто выполнить обход списка пицц, объединить их названия с вариантами ответов в один многострочный текст и отправить его клиенту.

В предыдущих главах вы узнали, что желательно разделить обработчики для лучшей организации приложения, поэтому с самого начала выделим обработчика, конструирующего текст приветствия и список пицц, и поместим его в отдельный файл в папке `handlers`. Итак, создайте папку `handlers` в корневой папке проекта, а внутри нее – файл с именем `pizza-menu.js`. В этом файле сначала импортируйте статический список пицц из файла `pizzas.json` в переменную `pizzas`. Затем объявите функцию `pizzaMenu`, а внутри нее переменную `greeting` с текстом `"Hello from Aunt Maria's pizzeria! Would you like to order a pizza? This is our menu:"` («Вас приветствует пиццерия тетушки Марии! Хотите заказать пиццу? Вот наш прейскурант:»). Затем выполните обход всех элементов в массиве `pizzas` и добавьте его содержимое в переменную `greeting` с коротким кодом для ответа в отдельной строке. В заключение верните переменную `greeting` из функции `pizzaMenu` и экспортируйте эту функцию. Полный код показан в листинге 10.2.

Листинг 10.2. Приветствие с меню для выбора пиццы

```
'use strict'

const pizzas = require('../data/pizzas.json')

function pizzaMenu() {
  let greeting = `Hello from Aunt Maria's pizzeria!
  Would you like to order a pizza?
  This is our menu:`

  pizzas.forEach(pizza => {
    greeting += `\n - ${pizza.name} to order reply with ${pizza.shortCode}`
  })

  return greeting
}

module.exports = pizzaMenu
```

Загрузить список пицц из файла pizzas.json.

Функция pizzaMenu.

Добавить каждый элемент из списка pizzas с коротким кодом shortCode в отдельной строке, который клиент должен использовать для заказа пиццы.

Сконструировать меню для возврата клиенту из SMS-чат-бота.

Экспортировать обработчик pizzaMenu.

Этот обработчик всегда возвращает список пицц при обращении к нему. В списке перечисляются названия пицц и короткие коды. Чтобы заказать выбранную пиццу, вместо нажатия кнопки клиент должен отправить текстовую

команду, потому что взаимодействие с SMS-чат-ботом ограничено текстовыми сообщениями.

Затем нужно изменить файл `sms-bot.js` и добавить вызов обработчика `pizzaMenu` в ответ на получение SMS от клиента. Поскольку на данный момент у нас нет других команд, мы можем просто вернуть импортированный обработчик `pizzaMenu`, как показано в листинге 10.3.

Листинг 10.3. Точка входа в SMS-чат-бот

```
'use strict'

const botBuilder = require('claudia-bot-builder')
const pizzaMenu = require('./handlers/pizza-menu')

const api = botBuilder((message, originalApiRequest) => {
  return [
    pizzaMenu()
  ]
}, { platforms: ['twilio'] })

module.exports = api
```

Импортировать Claudia Bot Builder.

Импортировать обработчик, возвращающий меню.

Просто вернуть pizzaMenu.

Объявить о поддержке платформы Twilio.

Теперь повторно разверните проект командой `claudia update`. Если после этого попробовать послать SMS чат-боту, он должен вернуть приветствие со списком пицц и их короткими кодами.

10.1.2. Оформление заказа

В настоящий момент, если клиент отправит SMS-сообщение нашему чат-боту, тот вернет текст приветствия и список пицц. Но если клиент отправит один из коротких кодов, SMS-чат-бот снова вернет приветствие со списком пицц. В этом разделе мы добавим в чат-бот распознавание коротких кодов и обработку заказа пиццы.

Для начала проверим, содержит ли ответ клиента короткий код. С этой целью загрузим список пицц в файле `sms-bot.js` и проверим содержимое сообщения на наличие короткого кода. Если код присутствует в сообщении, запросим у клиента адрес доставки.

Файл `sms-bot.js` с этими изменениями должен выглядеть, как показано в листинге 10.4.

Листинг 10.4. Распознавание намерения заказать пиццу

```
'use strict'

const botBuilder = require('claudia-bot-builder')
```

```
const pizzas = require('./data/pizzas.json')
const pizzaMenu = require('./handlers/pizza-menu'),
  orderPizza = require('./handlers/order-pizza')

const api = botBuilder((message, originalApiRequest) => {

  let chosenPizza
  pizzas.forEach(pizza => {
    if (message.indexOf(pizza.shortCode) != -1) {
      chosenPizza = pizza
    }
  })

  if (chosenPizza) {
    return orderPizza(chosenPizza, message.sender)
  }

  return [
    pizzaMenu()
  ]
}, { platforms: ['twilio'] })

module.exports = api
```

Импортировать список доступных пицц.

Импортировать обработчик, оформляющий заказ.

Выполнить обход коротких кодов и проверить присутствие каждого в ответе клиента.

Если клиент выбрал пиццу, вызвать обработчик оформления заказа и передать ему пиццу и отправителя сообщения.

Этот код завершает первый этап, проверяя присутствие короткого кода в сообщении клиента и передавая пиццу и отправителя (клиента) обработчику, реализующему оформление заказа. Теперь мы должны написать эту функцию-обработчик. Обработчик `order-pizza.js` должен получить объект выбранной пиццы и отправителя и сохранить новый заказ в таблице `pizza-orders`. Для создания нового заказа мы используем модуль `uuid`, с помощью которого создадим идентификатор заказа `orderID`, и идентификатор пиццы `pizza`. В поле `orderStatus` мы запишем статус `in-progress`, потому что заказ не должен передаваться для доставки до того, как мы узнаем адрес клиента. Кроме того, в атрибуте `platforms` мы укажем `twilio-sms-chatbot`, чтобы как-то различать заказы, хранящиеся в одной таблице. Наконец, сохраним отправителя в атрибуте `user`, чтобы иметь возможность узнать, кто сделал заказ. Код обработчика `order-pizza.js` показан в листинге 10.5.

Листинг 10.5. Обработчик, оформляющий заказ

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()
```

Импортировать AWS SDK.

Создать экземпляр DocumentClient.

```
const uuid = require('uuid/v4')    ← Импортировать модуль uuid.

function orderPizza(pizza, sender) {
  return docClient.put({                  ← Сохранить заказ в таблице DynamoDB.
    TableName: 'pizza-orders',
    Item: {                               ← С помощью uuid сгенерировать уникальный идентификатор заказа.
      orderId: uuid(),
      pizza: pizza.id,                    ← Установить статус заказа равным in-progress.
      orderStatus: 'in-progress',
      platform: 'twilio-sms-chatbot',     ← Указать, что заказ создан с использованием платформы Twilio SMS.
      user: sender                        ← Сохранить идентификатор пользователя, пославшего сообщение.
    }
  }).promise()
    .then((res) => {
      return 'Where do you want your pizza to be delivered? You can write
      your address.'                      ← Запросить адрес доставки.
    })
    .catch((err) => {
      console.log(err)
                                          ← Вернуть дружественное сообщение в случае ошибки.
      return [
        'Oh! Something went wrong. Can you please try again?'
      ]
    })
}

module.exports = orderPizza               ← Экспортировать обработчик orderPizza.
```

Здесь `message.sender` представляет номер телефона, с которого был сделан заказ. На данный момент нам не хватает только адреса клиента.

Обработка SMS-сообщений – не самая простая задача. Эти сообщения содержат простой текст, поэтому из них нельзя просто получить адрес. Учитывая это ограничение, вам действительно придется поломать голову над тем, как получить адрес.

В настоящее время заказ не может получить другого состояния, кроме `in-progress`. Не зная адреса клиента, мы не можем передать заказ компании, осуществляющей доставку. Нам нужно получить адрес и сохранить его, но пока, если сообщение не содержит короткого кода, обозначающего пиццу, наш SMS-чат-бот всегда будет отвечать приветствием и списком пицц. Нам нужно переопределить это поведение и суметь правильно обработать ввод адреса.

К счастью, эта проблема имеет решение. Мы сохранили номер телефона отправителя в заказе со статусом `in-progress`, поэтому, получив ответ с предполагаемым адресом клиента, мы можем сначала проверить наличие в базе дан-

ных заказа `in-progress` с номером отправителя. Если такой заказ есть и в нем отсутствует адрес, мы можем сохранить отправленное сообщение как адрес. На рис. 10.1 показан процесс парсинга сообщения.

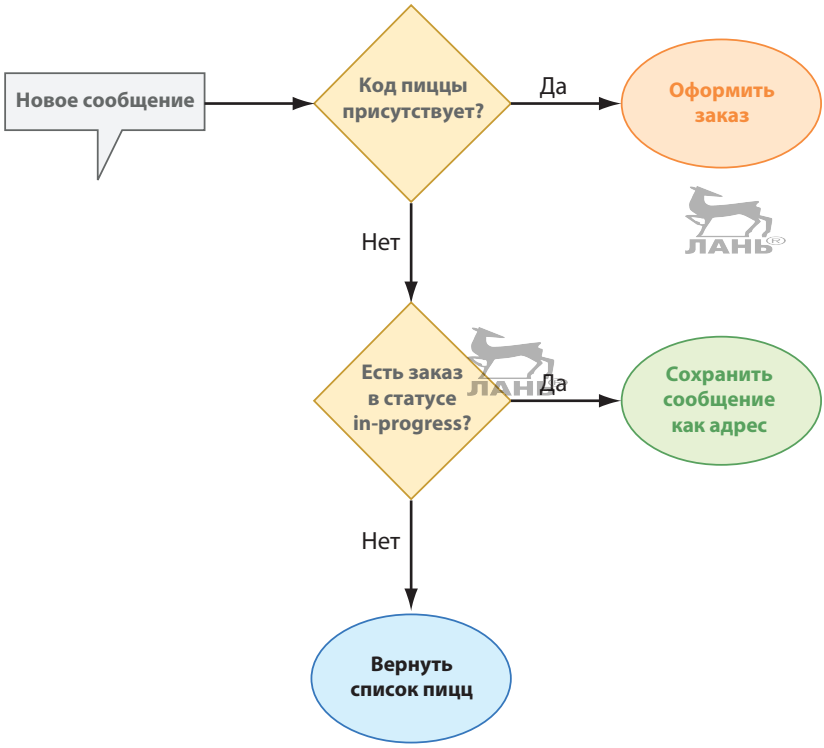


Рис. 10.1. Процесс парсинга сообщения в бессерверном SMS-чат-боте

Понять идею – важно, но не менее важно знать, как реализовать ее.

> **ПРИМЕЧАНИЕ.** В действующем приложении вы, вероятно, не должны сразу менять статус заказа, поскольку клиент мог ошибиться или забыл ответить и теперь может пожелать сделать новый заказ. Чтобы обработать эту ситуацию, можно запросить подтверждение у клиента. Если в ответ будет получено «YES» («ДА»), вы сможете изменить статус заказа на `pending`; если клиент ответит «NO» («НЕТ»), вы удалите заказ из базы данных. Однако мы не будем рассматривать эту процедуру в книге.

Сначала найдем заказ со статусом `in-progress`. С этой целью создадим отдельную функцию-обработчик `check-order-progress.js` в папке `handlers`. Внутри файла добавим логику сканирования таблицы DynamoDB, чтобы найти заказ, принадлежащий отправителю и имеющий статус `in-progress`. Поскольку команда `scan` в DynamoDB всегда возвращает массив найденных элементов, мы должны проверить, есть ли в результатах какие-либо элементы. Если есть,

вернем первый из них. Если нет, вернем значение `undefined` как признак, что ничего не найдено. Содержимое файла `check-order-progress.js` показано в листинге 10.6.

Листинг 10.6. Проверка заказа без адреса доставки

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function checkProgressOrder(sender) {

  return docClient.scan({
    ExpressionAttributeValues: {':user': sender, ':status': 'in-progress'},
    FilterExpression: 'user = :user and orderStatus = :status',
    Limit: 1,
    TableName: 'pizza-orders'
  }).promise()
  .then((result) => {
    if (result.Items && result.Items.length > 0) {
      return result.Items[0]
    } else {
      return undefined
    }
  })
  .catch((err) => {
    console.log(err)
      return [
        'Oh! Something went wrong. Can you please try again?'
      ]
    })
  });
}

module.exports = checkProgressOrder
```

Импортировать AWS SDK.

Создать экземпляр DocumentClient.

Определить фильтр для выбора записей.

Сканировать таблицу.

Инициировать фильтр для сканирования таблицы – нам нужно отыскать заказ по отправителю и статусу заказа (in-progress).

Ограничить число результатов, потому что нам нужно лишь узнать, есть ли заказ.

Задать таблицу DynamoDB для сканирования.

Вернуть заказ, соответствующий критериям поиска, или undefined, если такого заказа не найдено.

Вернуть дружественное сообщение в случае ошибки.

Экспортировать обработчик checkProgressOrder.

Теперь добавим изменения в основной файл `sms-bot.js`, чтобы проверить наличие заказа в статусе `in-progress` и сохранить в нем адрес для доставки. Если заказ не будет найден, вернем меню. Для начала импортируем обработчики `save-address.js` и `check-order-progress.js`. Затем используем их для проверки статуса заказа. Содержимое файла `sms-bot.js` должно выглядеть, как показано в листинге 10.7.

Листинг 10.7. Обновленный файл `sms-bot.js`

```
'use strict'

const botBuilder = require('claudia-bot-builder')
const pizzas = require('./data/pizzas.json')
const pizzaMenu = require('./handlers/pizza-menu'),
  orderPizza = require('./handlers/order-pizza'),
  checkOrderProgress = require('./handlers/check-order-progress'),
  saveAddress = require('./handlers/save-address')

const api = botBuilder((message, originalApiRequest) => {

  let chosenPizza
  pizzas.forEach(pizza => {
    if (message.indexOf(pizza.shortCode) != -1) {
      chosenPizza = pizza
    }
  })

  if (chosenPizza) {
    return orderPizza(chosenPizza, message.sender)
  }

  return checkOrderProgress(message.sender)
    .then(orderInProgress => {
      if (orderInProgress) {
        return saveAddress(orderInProgress, message)
      } else {
        return pizzaMenu()
      }
    })
}, { platforms: ['twilio'] })

module.exports = api
```

Импортировать обработчик check-order-progress.js.

Импортировать обработчик save-address.js.

Проверить наличие заказа в статусе in-progress, принадлежащем текущему отправителю.

Если такой заказ есть, сохранить сообщение клиента как адрес доставки.

Если такого заказа нет, вернуть список пицц.

Теперь нам не хватает только обработчика `save-address.js`. Создайте файл `save-address.js` в папке `handlers`, откройте и добавьте код, обновляющий заказ в таблице DynamoDB, используя указанный идентификатор заказа в качестве ключа. Также следует обновить адрес доставки и заменить статус `in-progress` на `pending`. Содержимое файла `save-address.js` показано в листинге 10.8.

Листинг 10.8. Обработчик `save-address`

```
'use strict'
```

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

function saveAddress(order, message) {

  return docClient.put({
    TableName: 'pizza-orders',
    Key: {
      orderId: order.id
    },
    UpdateExpression: 'set orderStatus = :o, address = :a',
    ExpressionAttributeValues: {
      ':n': 'pending',
      ':a': message.text
    },
    ReturnValues: 'UPDATED_NEW'
  }).promise()
}

module.exports = saveAddress
```

Задать идентификатор orderId заказа для изменения.

Задать выражение для изменения.

Задать значения для сохранения в заказе.

Определить значение, возвращаемое в случае успеха.

Экспортировать обработчик saveAddress.

Теперь выполните команду `claudia update` и пошлите сообщение на номер телефона чат-бота. Вот и все!

Вы только что создали свой первый бессерверный SMS-чат-бот с помощью Claudia.js и Twilio.

10.2. Эй, Алекса!

Наш SMS-чат-бот сделал свое дело, и теперь еще больше людей заказывают пиццу у тетушки Марии! Её пиццерия переполнена даже по понедельникам, и она начала подумывать о том, чтобы открыть второе заведение в другом районе. Дядюшка Фрэнк тоже счастлив, даже притом что после игр с SMS-чатом его счет за телефон поднялся до небес.

И вроде бы все хорошо, но вдруг к нам приходит Юлия, племянница тетушки Марии, и вручает нам подарок – Amazon Echo.

> **Amazon Echo**
>
> Amazon Echo – это домашнее устройство с голосовым управлением. Оно управляется голосовым помощником Алекса (Alexa). С ним можно разговаривать, давать команды и даже совершать покупки в интернете.

Юлия объясняет, что получила его в подарок на Рождество и оно ей порядком наскучило, пока она не поняла, что его можно использовать для заказа пиццы. Она хочет, чтобы тетушка Мария заняла лидирующие позиции на рынке и превзошла даже пиццерию Chess (возможно, потому что они не дают пиццу бесплатно, как тетушка Мария, но не будем углубляться в ее мотивы). Юлия считает, что, внедрив голосовые команды для заказа пиццы с помощью Echo раньше, чем это сделает пиццерия Chess, поможет в развитии бизнеса тетушки Марии и привлечет много новых клиентов. Это неплохая идея, поэтому мы решили помочь ей.

Но как работает Amazon Echo и как им пользоваться? Юлия показывает, что к устройству нужно обратиться по имени «Алекса».

Доступность Amazon Alexa

Впервые голосовой помощник Алекса начал использоваться в устройствах Amazon Echo и Amazon Echo Dot в 2014 году. Идеей для его реализации стала компьютерная голосовая и диалоговая система на борту космического корабля «*Enterprise*» из фантастического сериала «*Star Trek*» («Звездный путь»). Теперь голосовой помощник Алекса доступен на многих устройствах, включая семейство Amazon Echo и Amazon Fire TV, а также в мобильных приложениях для основных популярных платформ, таких как iOS и Android. Чтобы начать разговор с Алексой, необходимо специальное «пробуждающее слово», но на некоторых устройствах для этого требуется нажать кнопку.

Самой интересной и мощной особенностью Алексы является возможность создания своих *сценариев* (skills) – команд, распознаваемых Алексой, – которые можно опубликовать на Amazon Marketplace. На момент написания этих строк в Amazon Marketplace было зарегистрировано более 20 000 сценариев. Эти сценарии аналогичны компьютерным приложениям.

Создать свой сценарий довольно просто. Как показано на рис. 10.2, устройство с поддержкой голосового помощника Алекса пересылает аудиофайл в облако, где Алекса анализирует его, преобразует в *намерения* и *слоты* и затем передает результаты в формате JSON вашей функции Lambda или веб-обработчику. Намерения описывают, чего пытается добиться пользователь, а *слоты* – это переменные или динамические части данного намерения. В ответ функция Lambda или веб-обработчик должны вернуть файл JSON, описывающий голосовой ответ Алексы, который услышит пользователь. Прежде чем создать наш первый сценарий, посмотрим, как они работают и чем отличаются от чат-ботов для Facebook Messenger и Twilio.

Устройство сценария для голосового помощника Алекса

Алекса и другие голосовые помощники работают немного иначе, чем большинство платформ чат-ботов. Вот некоторые основные отличия:

- голосовое сообщение не просто передается веб-обработчику, а сначала попадает во встроенный механизм обработки естественного языка, который проанализирует аудиозапись и передаст результаты вашему веб-обработчику в формате JSON;
- общение с Алексой основано на командах и, в отличие от большинства платформ чат-ботов, не допускает бесплатных разговоров. Чтобы Алекса могла распознать и обработать команду, она должна быть определена заранее;
- обычно перед командой необходимо произнести специальное слово, активирующее голосового помощника и приводящее его в готовность принять команду.

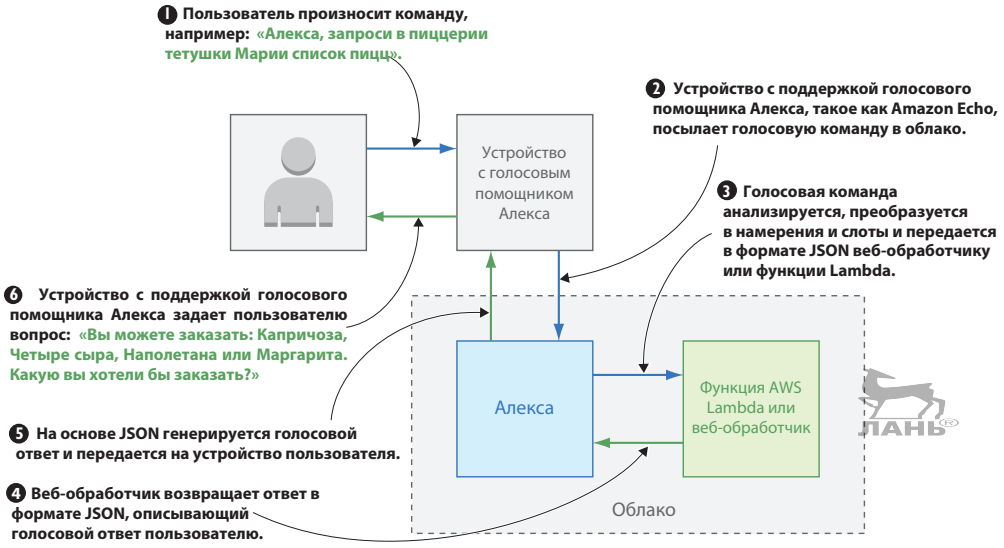


Рис. 10.2. Как Алекса выполняет сценарии

Как показано на рис. 10.3, типичная команда для Алексы включает следующие элементы:

- специальное слово, активирующее голосового помощника;
- фразу запуска;
- имя вызова;
- высказывание с дополнительными параметрами (слотами).

Рис. 10.3. Вызов сценария для Алексы

Другими примерами могут служить голосовые команды: «Алекса, начать работу с пиццерией тетушки Марии» и «Алекса, сообщи в пиццерию тетушки Марии заказ на пиццу».

По умолчанию используется активирующее слово «Алекса», но его можно изменить в настройках устройства. На момент написания этих строк можно было выбрать следующие слова: «Алекса», «Амазон», «Эхо» и «Компьютер».

Фраза запуска говорит Алексе активировать определенный сценарий. В число фраз запуска входят[1]: «ask» («запросить»), «launch» («запустить»), «start» («начать»), «show» («показать») и многие другие.

Имя вызова – это имя сценария, который требуется запустить. Конструируя свой сценарий, важно выбрать хорошее имя.

> **ПРИМЕЧАНИЕ.** Некоторые рекомендации по выбору имени вызова вы найдете на сайте http://mng.bz/T6ly.

Наконец, при использовании фразы запуска «начать» нужно сообщить Алексе, что должен сделать сценарий. Эти инструкции известны как *высказывания*. Наличие статических высказываний не дает большой гибкости, поэтому Алекса позволяет добавлять в инструкции динамические элементы, которые называют *слотами*.

Пользователь произносит команду, а Алекса анализирует ее и передает результаты функции AWS Lambda или веб-обработчику.

Как показано на рис. 10.4, голосовая команда преобразуется механизмом анализа естественного языка в намерение. Если в команде есть какие-либо слоты, они преобразуются в объекты, содержащие имя и значение слота. После успешного анализа голосовой команды Алекса создает объект JSON, который содержит тип запроса, имя намерения и значения слотов, а также другие данные, такие как атрибуты сеанса и метаданные.

Алекса может принимать запросы нескольких типов (табл. 10.1).

Таблица 10.1. Типы запросов, распознаваемые Алексой

Тип запроса	Описание
`LaunchRequest`	Посылается, когда сценарий вызывается фразой «start» («начать») или «launch» («запустить»), например: «Алекса, начать работу с пиццерией тетушки Марии»; не поддерживает дополнительных слотов
`IntentRequest`	Посылается, когда голосовая команда распознается как содержащая намерение
`SessionEndedRequest`	Посылается, когда пользователь завершает сеанс
`AudioPlayer or PlaybackController (префиксы)`	Посылается, когда пользователь использует аудиоплеер или функцию воспроизведения, такую как приостановка или переход к следующей песне

[1] На момент публикации книги (2019 год) голосовой помощник Алекса поддерживал только английский и немецкий языки. Имейте это в виду при опробовании примеров. – *Прим. перев.*

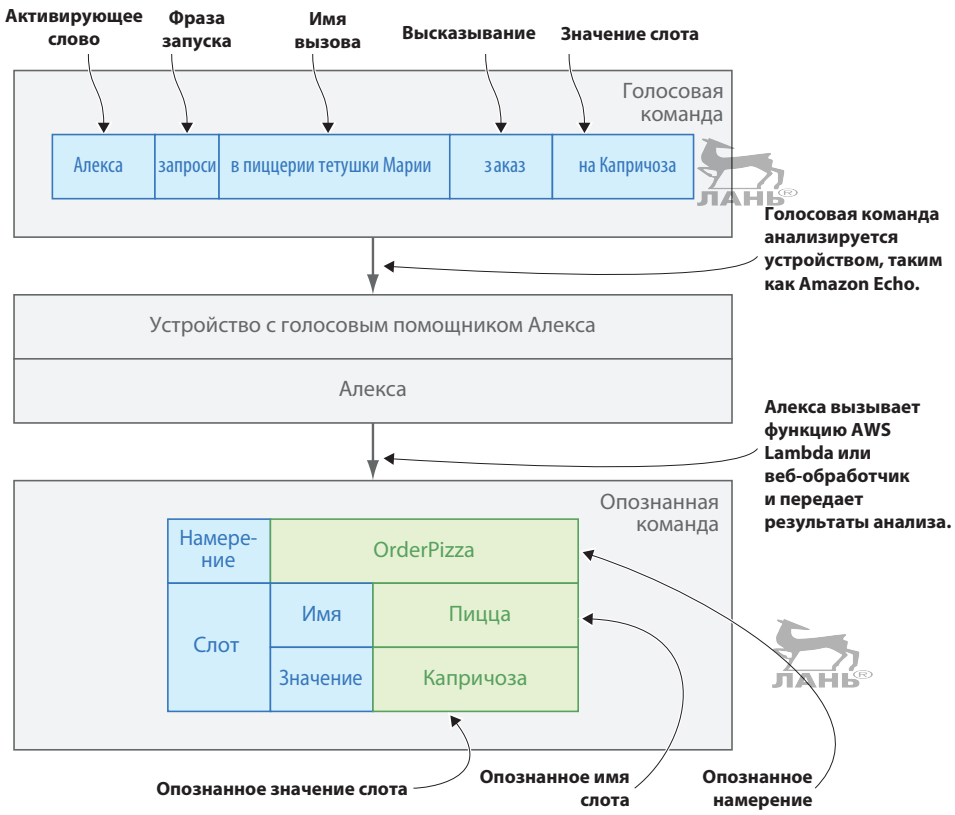


Рис. 10.4. Процесс передачи и анализа голосовой команды

Другой важной частью голосовых команд является *сеанс*. В отличие от Facebook Messenger, Алекса сохраняет некоторые данные между командами, но вы должны явно сохранять их в своем сеансе. Сеанс Алексы – это диалог с пользователем. Если сеанс активен, Алекса будет ждать следующей команды пользователя после ее ответа. Пока сеанс активен, последующие команды не обязательно должны начинаться с активирующего слова, потому что Алекса ждет ответа в течение следующих нескольких секунд.

Перед созданием сценария для Алексы его нужно спроектировать. Проектирование сценариев для голосового помощника, конечно же, связано не с пользовательским интерфейсом, а с проектированием взаимодействий и схемы намерений. Далее мы посмотрим, как это делается.

10.2.1. Подготовка сценария

Проектирование – самый важный этап в создании сценария. Голосовых помощников часто называют «умными помощниками», но на самом деле они все еще далеки от фантастического компьютера HAL 9000 из фильма «2001: Космическая одиссея», а возможности анализа естественного языка по-прежнему весьма ограничены.

Проектирование интерактивных взаимодействий выходит далеко за рамки этой книги, но в интернете есть много хороших ресурсов, посвященных данной теме. Хорошей отправной точкой может послужить официальное руководство по проектированию голосовых команд на сайте Amazon https://developer.amazon.com/designing-for-voice/.

Сценарий, который мы реализуем в этой главе, очень прост. Он:

1) даст пользователю возможность получить список пицц;
2) позволит заказать выбранную пиццу;
3) спросит у пользователя адрес доставки.

Базовый алгоритм работы сценария, который мы создадим, показан на рис. 10.5.

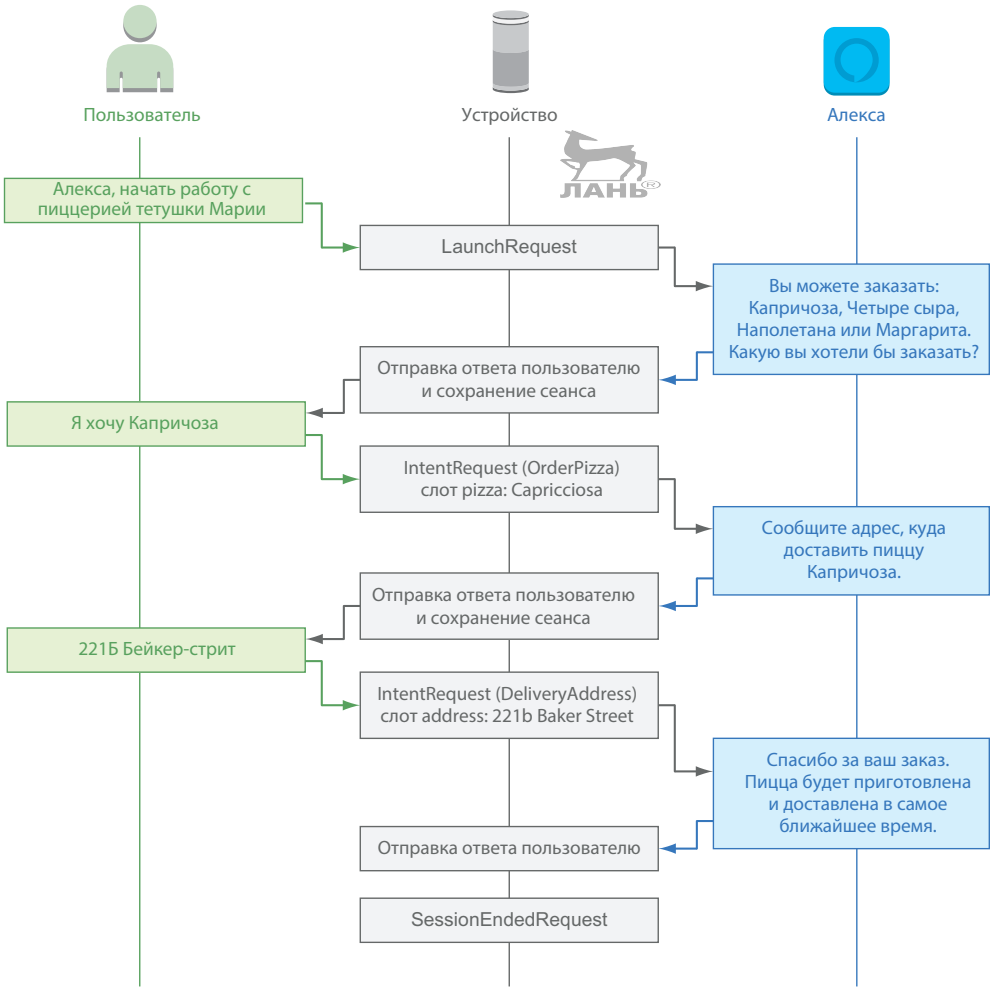


Рис. 10.5. Алгоритм сценария для оформления заказа в пиццерии тетушки Марии с помощью голосового помощника Алекса

Чтобы создать сценарий для Алексы, нам понадобится определить:

- схему намерений;
- типы слотов, если они существуют;
- список образцов выражений.

> **ПРИМЕЧАНИЕ.** Инструкции по настройке нового сценария для Алексы, его подключению к AWS Lambda и вводу схемы намерений, пользовательских слотов и образцов высказываний вы найдете в приложении В.

Схема намерений – это объект JSON, в котором перечислены все намерения (действия), соответствующие устному запросу пользователя. Каждое намерение может иметь слоты, и каждый слот должен иметь один тип. Типы слотов могут быть пользовательскими или встроенными. Компания Amazon предлагает множество встроенных типов слотов, таких как имена, даты и адреса. Полный список встроенных типов слотов можно найти по адресу https://developer.amazon.com/docs/custom-skills/slot-type-reference.html.

В дополнение к встроенным типам слотов есть возможность определить *свои типы*. Для этого нужно определить имя и список возможных значений. Список значений определяется как текстовый файл, в котором каждая строка представляет одно возможное значение слота.

Список *образцов высказываний* – это набор вероятных разговорных фраз, соответствующих намерениям. Он должен включать как можно больше репрезентативных фраз, которые Алекса будет использовать для обучения своего механизма анализа естественного языка. Подобно пользовательским типам слотов, образцы высказываний определяются в виде текстового файла, в котором каждый образец занимает отдельную строку. Каждая строка начинается с намерения, в которое должен быть преобразован текст, с последующим пробелом и текстом высказывания, как показано на рис. 10.6.

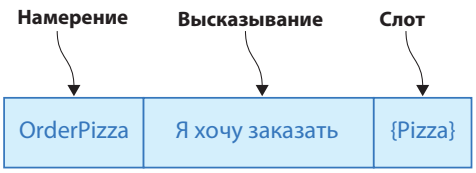


Рис. 10.6. Образцы высказываний в сценариях для голосового помощника Алекса

Теперь подготовим все, что нам понадобится, начав со схемы намерений. Как вы помните, это объект JSON, содержащий массив объектов намерений. Каждый объект намерения имеет ключ `intent` с именем намерения в качестве значения.

Наш сценарий должен включать намерения `OrderPizza` и `DeliveryAddress`, оба со своими слотами. Намерение `OrderPizza` должно иметь слот с названием пиццы, а `DeliveryAddress` – слот с адресом. Для представления адресов имеет-

ся встроенный тип слотов, но для названия пиццы нам придется определить свой тип. Мы определим его позже и дадим ему имя `LIST_OF_PIZZAS`.

Для поддержки слотов оба объекта намерений должны иметь ключ `slots` с массивом слотов в качестве значения. Массив `slots` в обоих случаях будет содержать только один объект с атрибутом, определяющим имя и тип слота.

Для намерения `OrderPizza` добавим слот `Pizza` с типом `LIST_OF_PIZZAS`. Для намерения `DeliveryAddress` добавим слот `Address` со встроенным типом `AMAZON.PostalAddress`, который определяет значения почтовых адресов.

Встроенные типы слотов

Пакет Alexa Skills Kit – коллекция классов, инструментов и документации для создания сценариев – поддерживает несколько встроенных типов слотов, которые определяют, как должны распознаваться и обрабатываться данные в слотах. Типы, входящие в пакет, делятся на следующие основные категории:

- числа, даты и время;
- списки.

К первой категории относятся типы слотов, помогающие распознавать числа, например `AMAZON.NUMBER` и `AMAZON.FOUR_DIGIT_NUMBER`, и значения даты и времени, например `AMAZON.DATE` и `AMAZON.DURATION`.

Все типы слотов во второй категории представляют списки элементов, таких как адреса, имена актеров, названия городов, виды животных и многое другое. Например, тип `AMAZON.Animal` распознает виды животных, тип `AMAZON.Book` распознает названия книг, а тип `AMAZON.PostalAddress` распознает адреса с номерами зданий.

За дополнительной информацией обращайтесь по адресу https://developer.amazon.com/docs/custom-skills/slot-type-reference.html.

Добавим дополнительно еще одно намерение: `ListPizzas`. У этого намерения не будет слотов – с его помощью пользователь сможет запрашивать у Алексы список пицц. Оно будет вызывать то же действие, что и `LaunchRequest`.

По завершении схема намерений должна выглядеть, как показано в листинге 10.9.

Листинг 10.9. Схема намерений

```
{
  "intents": [
    {
      "intent": "ListPizzas"
    }, {
      "intent": "OrderPizza",
```

Массив намерений.

Намерение ListPizzas.

Намерение OrderPizza.

```
      "slots": [
        {
          "name": "Pizza",
          "type": "LIST_OF_PIZZAS"
        }
      ]
    }, {
      "intent": "DeliveryAddress",
      "slots": [
        {
          "name": "Address",
          "type": "AMAZON.PostalAddress"
        }
      ]
    }
  ]
}
```

Слот Pizza для намерения OrderPizza.

Свой тип слота: LIST_OF_PIZZAS.

Намерение DeliveryAddress.

Слот Address для намерения DeliveryAddress.

Слот Address имеет встроенный тип AMAZON.PostalAddress.

Следующий шаг – определение типа слота `LIST_OF_PIZZAS`. Как уже отмечалось выше, свои типы слотов определяются в виде простых текстовых файлов, в которых перечисляются все возможные значения, по одному в строке. Наш слот `LIST_OF_PIZZAS` должен быть списком пицц, как показано в листинге 10.10.

Листинг 10.10. Тип слота `LIST_OF_PIZZAS`

```
Capricciosa
Quattro Formaggi
Napoletana
Margherita
```

Последний шаг – подготовка списка образцов высказываний. И снова этот список определяется как простой текстовый файл, где каждое высказывание занимает отдельную строку.

Каждая строка должна начинаться с имени намерения, за которым следует пробел и образец фразы; например, `ListPizzas Pizza menu`. Чем больше простых фраз будет определено, тем лучше, чтобы у Алексы была возможность проанализировать множество других простых фраз. Например, если определить высказывание `ListPizzas Pizza menu`, Алекса будет распознавать такие простые фразы, как «Show me the pizza menu» (Показать меню пиццы) или «What's on the pizza menu?» (Что есть в меню пиццы?).

Мы в нашем примере используем список высказываний, представленный в листинге 10.11. При желании вы можете оставить пустые строки для удобочитаемости.

Листинг 10.11. Образцы высказываний

```
ListPizzas Pizza menu
ListPizzas Which pizzas do you have
ListPizzas List all pizzas

OrderPizza {Pizza}
OrderPizza order {Pizza}
OrderPizza I want {Pizza}
OrderPizza I would like to order {Pizza}

DeliveryAddress {Address}
DeliveryAddress Deliver it to {Address}
DeliveryAddress address is {Address}
```

Образцы высказываний для намерения ListPizzas.

Образцы высказываний для намерения OrderPizza.

Образцы высказываний для намерения DeliveryAddress.

10.2.2. Оформление заказа с помощью Алексы

Теперь, имея схему намерений и список образцов высказываний, можно приступать к сценарию для Алексы.

Как упоминалось выше, Алекса может вызывать веб-обработчик или функцию AWS Lambda. Claudia Bot Builder поддерживает сценарии для голосового помощника Алекса, и у нас есть возможность повторно использовать функцию AWS Lambda, реализующую чат-бот для Facebook Messenger или Twilio. Но при этом придется добавить слой API Gateway между Алексой и функцией Lambda, что увеличивает сложность и ухудшает производительность. (Однако такое решение может упростить обслуживание за счет повторного использования части кода.)

Наш сценарий для Алексы очень прост, поэтому создадим для него отдельную функцию AWS Lambda. Создание дополнительной функции Lambda не увеличит первоначальную стоимость услуги, в отличие от традиционных серверов, когда нужно оплатить и настроить экземпляр, – затраты на установку и развертывание равны нулю.

Еще одно большое преимущество использования Claudia Bot Builder заключается в том, что этот пакет автоматически анализирует входные данные и передает их в простом формате, а также избавляет от необходимости писать шаблонный код для возврата ответа. Входные данные для сценария автоматически преобразуются в формат JSON, а для форматирования ответного сообщения можно использовать те же инструменты, которые использует Claudia Bot Builder: модуль `alexa-message-builder`, который доступен как отдельный модуль NPM и не требует импортировать весь пакет Claudia Bot Builder.

Создайте еще одну папку на одном уровне с папками `pizza-api` и `pizza-fb-bot`. Для единообразия назовем ее `pizza-alexa-skill`.

Затем перейдите в эту папку и инициализируйте проект NPM. Также установите `alexa-message-builder` как зависимость, выполнив команду `npm install`

`alexa-message-builder --save`. Затем создайте файл `skill.js` и откройте его в текстовом редакторе.

Файл `skill.js` будет содержать стандартную функцию AWS Lambda и экспортировать ее как функцию-обработчик с параметрами `event`, `context` и `callback`. Он также должен импортировать только что установленный модуль `alexa-message-builder`.

Поскольку мы решили не использовать Claudia Bot Builder, необходимо убедиться, что событие `event`, полученное функцией-обработчиком, является действительным запросом голосового помощника. Для этого можно проверить наличие атрибута `event.request` и его соответствие типу `LaunchRequest`, `IntentRequest` или `SessionEndedRequest`. Наш сценарий не будет управлять воспроизведением аудиофайлов, поэтому нам не требуется проверять `event.request` на соответствие этим типам запросов.

Если событие `event` недопустимое, нужно вернуть признак ошибки, передав его в функцию обратного вызова `callback`.

Затем нужно добавить инструкции `if...else` и с их помощью выяснить, какое намерение вызвало обработчик. Мы должны проверить следующие условия и вернуть соответствующие ответы:

1) если `event.request.type` соответствует `LaunchRequest` или `IntentRequest` с намерением `ListPizzas`, мы должны вернуть список пицц;
2) если получено намерение `OrderPizza` со слотом `Pizza`, содержащим название одной из пицц, мы должны запросить адрес доставки;
3) если получено намерение `DeliveryAddress` со слотом `Address`, мы должны сообщить пользователю, что его заказ принят к исполнению;
4) в любом другом случае следует сообщить пользователю, что произошла ошибка.

Если `request.type` имеет значение `IntentRequest`, имя намерения можно получить из `event.request.intent.name`. Если намерение имеет слоты, они будут доступны в объекте `event.request.intent.slots`.

Например, вот как можно проверить получение намерения `DeliveryAddress` и наличие слота `Address` в нем:

```
if (
  event.request.type === 'IntentRequest' &&
  event.request.intent.name === 'DeliveryAddress' &&
  event.request.intent.slots.Address.value
) { /* ... */ }
```

Перед инструкциями `if...else` можно создать экземпляр `AlexaMessageBuilder`, как показано ниже:

```
const AlexaMessageBuilder = require('alexa-message-builder')
```

Это позволит только один раз вызвать функцию `callback` после инструкций `if...else`:

```
callback(null, message)
```

В каждый блок `if...else` нужно добавить создание возвращаемого сообщения. Для намерений `LaunchRequest` и `ListPizzas` мы должны вернуть список всех пицц, предложить пользователю выбрать одну из них и сохранить сеанс открытым. Имейте в виду, что вопрос, задаваемый пользователю, должен быть простым и ясным, чтобы пользователь мог ответить однозначно, а голосовой помощник – правильно разобрать ответ. Ниже показано, как это можно реализовать:

```
const message = new AlexaMessageBuilder()
  .addText('You can order: Capricciosa, Quattro Formaggi, Napoletana, or
    Margherita. Which one do you want?')
  .keepSession()
  .get()
```

Это не лучший пример вопроса, потому что пользователь может ответить на него «the first one» («первую»), и тогда Алекса не поймет ответа. Однако он достаточно наглядно иллюстрирует, как работают сценарии для голосового помощника Алекса.

Подобно шаблонам в Facebook Messenger, `AlexaMessageBuilder` является классом, и его методы возвращают ссылку `this`, что позволяет объединять их в цепочки. Чтобы оставить сеанс открытым, можно вызвать метод `.keepSession`, а в конце следует вызвать метод `.get`, чтобы преобразовать ответ в простой объект JavaScript в формате, который понимает Алекса.

Ответ на намерение `OrderPizza` конструируется аналогично. Мы можем вернуть текст «What's the address where your pizza should be delivered?» («Сообщите адрес, куда доставить пиццу») и оставить сеанс открытым. Основное отличие от обработки предыдущего намерения состоит в том, что мы должны сохранить выбранную пиццу в атрибутах сеанса. Сделать это можно так:

```
.addSessionAttribute('pizza', event.request.intent.slots.Pizza.value)
```

По окончании содержимое файла `skill.js` должно выглядеть, как показано в листинге 10.12.

Листинг 10.12. Сценарий для голосового помощника Алекса

```
'use strict'

const AlexaMessageBuilder = require('alexa-message-builder')

function alexaSkill(event, context, callback) {
  if (
    !event ||
```

Импортировать библиотеку Alexa Message Builder.

Определение функции Lambda.

```
  !event.request ||
  ['LaunchRequest', 'IntentRequest', 'SessionEndedRequest'].indexOf(event.
    request.type) < 0
) {
  return callback('Not valid Alexa request')
}

const message = new AlexaMessageBuilder()

if (
  event.request.type === 'LaunchRequest' ||
  (event.request.type === 'IntentRequest' && event.request.intent.name ===
    'ListPizzas')
) {
  message
    .addText('You can order: Capricciosa, Quattro Formaggi, Napoletana, or
      Margherita. Which one do you want?')
    .keepSession()
} else if (
  event.request.type === 'IntentRequest' &&
  event.request.intent.name === 'OrderPizza' &&
  ['Capricciosa', 'Quattro Formaggi', 'Napoletana',
    'Margherita'].indexOf(event.request.intent.slots.Pizza.value) > -1
) {
  const pizza = event.request.intent.slots.Pizza.value

  message
    .addText(`What's the address where your ${pizza} should be delivered?`)
    .addSessionAttribute('pizza', pizza)
    .keepSession()
} else if (
  event.request.type === 'IntentRequest' &&
  event.request.intent.name === 'DeliveryAddress' &&
  event.request.intent.slots.Address.value
) {
  // Сохранить заказ

  message
    .addText(`Thanks for ordering pizza. Your order has been processed
        and the pizza should be delivered shortly`)
} else {
  message
    .addText('Oops, it seems there was a problem, please try again')
```

Проверить, является ли сообщение от Алексы действительным событием, и если условие не выполняется – вернуть сообщение об ошибке.

Создать экземпляр AlexaMessageBuilder.

Получено намерение LaunchRequest или ListPizzas?

Вернуть список пицц.

Получено намерение OrderPizza?

Запросить адрес доставки.

Сохранить выбранную пиццу в сеансе.

Получено намерение DeliveryAddress?

Сохранить заказ в DynamoDB.

Сообщить пользователю, что его заказ принят.

Иначе сообщить, что возникла ошибка.

```
  }

  callback(null, message.get())
}

export.handler = alexaSkill
```

Вернуть сообщение из функции AWS Lambda.

Экспортировать функцию-обработчик.

Далее следует развернуть функцию Lambda командой `claudia create`. На этот раз операция развертывания имеет две основные особенности:

- для развертывания поддерживаются только регионы `eu-west-1`, `us-east-1` и `us-west-1`;
- версия `latest`, используемая по умолчанию, не поддерживается, поэтому нужно выбрать какое-то другое имя версии, например `skill`.

В листинге 10.13 показана полная команда развертывания.

Листинг 10.13. Развертывание сценария для голосового помощника с помощью Claudia

```
claudia create \
  --region eu-west-1 \
  --handler skill.handler \
  --version skill
```

Создать функцию Lambda.

Определить регион (здесь используется eu-west-1, соответствующий Ирландии).

Определить путь к обработчику.

Определить версию функции AWS Lambda (здесь используется skill).

После развертывания функции Lambda нужно разрешить Алексе вызывать ее. Сделать это можно с помощью команды `claudia allow-alexa-skill-trigger`. Не забудьте указать версию, которую указали в команде `claudia create`, – в нашем примере это `skill`, то есть вы должны выполнить команду `claudia allow-alexa-skill-trigger --version skill`.

После того как вы загрузите свою функцию Lamda и разрешите Алексе вызывать сценарий, выполните настройки, как описано в приложении B. Закончив с настройками, вы сможете просто сказать «Alexa, start Aunt Maria's Pizzeria» («Алекса, начать работу с пиццерией тетушки Марии»)[2].

10.3. Опробование!

Чат-боты и голосовые помощники добавляют уникальные возможности! А теперь попробуйте усовершенствовать свой сценарий.

10.3.1. Упражнение

Ваша задача: отправить приветственное сообщение в ответ на намерение `LaunchRequest`, например такое: «Welcome to Aunt Maria's Pizzeria! You can order

[2] На момент публикации книги (2019 год) голосовой помощник Алекса поддерживал только английский и немецкий языки. Имейте это в виду при опробовании примеров. – *Прим. перев.*

pizza with this skill. We have Capricciosa, Quattro Formaggi, Napoletana, and Margherita. Which pizza do you want?» («Добро пожаловать в пиццерию тетушки Марии! С помощью голосового помощника вы можете заказать у нас пиццу. Мы можем предложить Капричоза, Четыре сыра, Наполетана и Маргарита. Какую вы предпочитаете?»)

Чтобы усложнить задачу и сделать ее немного интереснее, добавьте повторную отправку вопроса после получения намерений `LaunchRequest` и `ListPizzas`. Повторный вопрос посылается, когда сеанс все еще открыт, но клиент не ответил на первый вопрос в течение нескольких секунд.

Подсказки:

- разделите намерения `LaunchRequest` и `ListPizzas` на две инструкции `if...else`;
- проверьте, открыт ли сеанс;
- чтобы узнать, как организовать повторную отправку вопроса, прочитайте документацию для `alexa-message-builder` по адресу: https://github.com/stojanovic/alexa-message-builder.

10.3.2. Решение

Как показано в листинге 10.14, чтобы выполнить упражнение, достаточно изменить небольшой фрагмент кода в файле `skill.js`. Вы должны разделить намерения `LaunchRequest` и `ListPizzas` на два отдельных блока `if` и использовать метод `.addRepromptText` для повторной отправки вопроса.

Листинг 10.14. Изменения в файле `skill.js`

```
if (event.request.type === 'LaunchRequest') {
  message
    .addText('Welcome to Aunt Maria's Pizzeria! You can order pizza with
      this skill. We have: Capricciosa, Quattro Formaggi, Napoletana, or
      Margherita. Which pizza do you want?')
    .addRepromptText('You can order: Capricciosa, Quattro Formaggi,
      Napoletana, or Margherita. Which pizza do you want?')
    .keepSession()
} else if (event.request.type === 'IntentRequest' && event.request.intent.
      name === 'ListPizzas') {
  message
    .addText('You can order: Capricciosa, Quattro Formaggi, Napoletana, or
      Margherita. Which pizza do you want?')
    .addRepromptText('You can order: Capricciosa, Quattro Formaggi,
      Napoletana, or Margherita. Which pizza do you want?')
    .keepSession()
}
```

LaunchRequest теперь обрабатывается в отдельном блоке if...else.

Определить текст вопроса.

Настроить повторную отправку вопроса для LaunchRequest.

ListPizzas теперь обрабатывается в отдельном блоке if...else.

Настроить повторную отправку вопроса для ListPizzas.

Остальной код в файле не изменился.

Изменив код, разверните его командой `claudia update` – и ваш сценарий будет готов для тестирования.

10.4. Конец второй части: специальное упражнение

Мы подошли к концу второй части этой книги. Теперь, когда вы многое узнали о бессерверных приложениях и чат-ботах, пришло время объединить эти знания. Мы предлагаем вам специальное упражнение: подключить SMS-чат-бот и сценарий для голосового помощника Алекса к базе данных и службе доставки. Имейте в виду, что когда заказ оформляется с помощью голосового помощника, у вас не будет возможности уведомить пользователя об изменении статуса доставки пиццы.

ПРИМЕЧАНИЕ. Мы не даем подсказок к специальным упражнениям.

В заключение

- Claudia Bot Builder предлагает простой и быстрый способ создания SMS-чат-ботов с использованием Twilio.
- Из-за свойственных ограничений SMS-чат-боты должны предлагать пользователям простой и ясный способ вернуть ответ.
- Код чат-бота можно повторно использовать на разных платформах, но иногда проще разделить его на несколько функций Lambda.
- Claudia Bot Builder поддерживает сценарии для голосового помощника Алекса, но, так как Алекса способен вызывать функции Lambda самостоятельно, вы можете сэкономить деньги и уменьшить задержку, развернув сценарий без API Gateway.
- Несмотря на простоту разработки сценариев для голосового помощника Алекса, проектирование надежных голосовых взаимодействий – не самая простая задача.

Часть III

Дальнейшие шаги

Благодаря нашей работе пиццерия тетушки Марии снова процветает. Но, несмотря на то что все хорошо работает, частые изменения в коде начали вызывать появление случайных ошибок в приложении. Настало время узнать, как автоматизировать тестирование бессерверных приложений и как организовать тестирование Pizzeria API (глава 11). Кроме того, многие клиенты уже не раз спрашивали про возможность выполнения онлайн-платежей, поэтому мы должны внедрить поддержку платежной системы Stripe с помощью AWS Lambda (глава 12).

Однажды, когда вся наша большая семья собралась за одним столом, тетушка Мария похвасталась своим новым онлайн-бизнесом. Ее брат, дядюшка Роберто, поинтересовался, сможем ли мы перенести его существующее приложение на бессерверную основу. Он пользуется услугами Express.js, и у него нет претензий к качеству обслуживания, но платит за это гораздо больше, чем тетушка Мария, а кроме того, у него наблюдаются некоторые проблемы с масштабированием. Мы должны будем изучить и запустить его приложение Express.js в AWS Lambda (глава 13). Затем рассмотрим приемы миграции более сложных приложений в бессерверное окружение (глава 14).

В заключение посмотрим, как другие компании используют бессерверное окружение и как переносят в него свои приложения, и узнаем, какие преимущества они получили от этого (глава 15).

Глава 11

Тестирование, тестирование и еще раз тестирование

Эта глава охватывает следующие темы:

- подходы к тестированию бессерверных приложений;
- приемы разработки бессерверных функций, упрощающие их тестирование;
- автоматическое тестирование на локальной машине.

Разработка приложений – сложный и иногда болезненный процесс. Даже при тщательной проверке ошибки в программном обеспечении могут ускользнуть от взгляда проверяющего и подвергнуть риску вашу компанию или ваших пользователей. За последние два десятка лет предотвращение ошибок и тестирование программного обеспечения превратились в насущную необходимость. Как гласит старая пословица: *болезнь легче предупредить, чем лечить*.

Теперь, с появлением бессерверных окружений, при тестировании программного обеспечения приходится сталкиваться с новыми сложностями. Отсутствие сервера и использование AWS Lambda и API Gateway могут сделать тестирование приложений пугающим. Цель этой главы – показать, как с незначительными изменениями в подходе к тестированию приложений можно тестировать бессерверные приложения с той же легкостью, что и обычные, выполняющиеся на сервере.

11.1. Тестирование обычных и бессерверных приложений

Недавно тетушка Мария заметила, что у некоторых клиентов не получается заказать пиццу, а Пьер, ее разработчик мобильного приложения, сообщил о «странных» ошибках, иногда возникающих даже при отображении списка пицц. Тетушка Мария обеспокоена тем, что теряет клиентов, и попросила нас найти и устранить проблему. Мы можем попробовать отладить Pizza API, что-

бы выяснить причины проблемы, но ошибка также может быть на веб-сайте или в мобильном приложении. Тестировать все службы вручную каждый раз, когда возникает проблема, утомительно и требует слишком много времени. Поэтому было бы неплохо *автоматизировать* тестирование. Автоматическое тестирование требует первоначальных трудозатрат, чтобы написать код, который будет тестировать приложение, но затем его можно запускать снова и снова, чтобы проверить Pizza API после внесения изменений, добавления новых функций или устранения вновь обнаруженной проблемы.

Автоматизированное тестирование – обширная тема. Существует масса разных типов автоматических тестов, каждый из которых использует свой подход: от тестирования небольших фрагментов (или *модулей*) кода до полной проверки всех функций приложения.

Возьмем для примера Pizza API: маленькие модульные тесты будут проверять работу отдельных функций в обработчиках, тогда как тесты для всего приложения (также известные как *сквозные тесты*) проверят всю цепочку, от вывода списка пицц до обработки заказов.

Существует много других типов автоматических тестов. Они часто группируются в три слоя, в зависимости от используемого подхода, снизу вверх:

- *слой модульных тестов* – эти тесты проверяют небольшие фрагменты (модули) кода приложения, например отдельные функции;
- *слой интеграционных тестов* – эти тесты проверяют совместную работу небольших фрагментов кода, упомянутых в предыдущем пункте;
- *слой тестов пользовательского интерфейса* (ПИ) – эти тесты проверяют поведение приложения целиком с точки зрения пользователя.

В дополнение к этим трем слоям автоматизированного тестирования существует еще один слой – слой тестов, выполняемых вручную, – которые обычно выполняются группами контроля качества.

Этим уровням сопутствуют разные затраты на тестирование. Визуальное представление слоев с соответствующими затратами часто называют *пирамидой тестирования*. Обычно в пирамиду включают только три слоя, но чтобы лучше понять значение и стоимость каждого типа тестов, мы добавим в нее слой ручного тестирования. Со всеми четырьмя слоями пирамида тестирования выглядит, как показано на рис. 11.1. Стоимость тестирования на этом рисунке приводится для обычных приложений, размещаемых на сервере.

Пирамида автоматизированных тестов

Впервые трехуровневая пирамида автоматизированного тестирования была упомянута Майком Коном (Mike Cohn) в своей книге «Succeeding with Agile» (Addison Wesley, 2009). Мы настоятельно рекомендуем эту книгу желающим больше узнать об автоматизации тестирования. (*Кон Майк*. Scrum. Гибкая разработка ПО. М.: Вильямс, 2011. ISBN 978-5-8459-1731-7. – *Прим. перев.*)

На рис. 11.1 видно, что высокоуровневые тесты пользовательского интерфейса обходятся дороже модульных тестов, потому что проверяют поведение всего приложения с точки зрения пользователя, работу визуальных элементов, правильность ввода входных данных, отображение значений и т. д. Однако тесты пользовательского интерфейса не только более дорогие, но и более медленные из-за количества проверок и объема выполняемого кода.

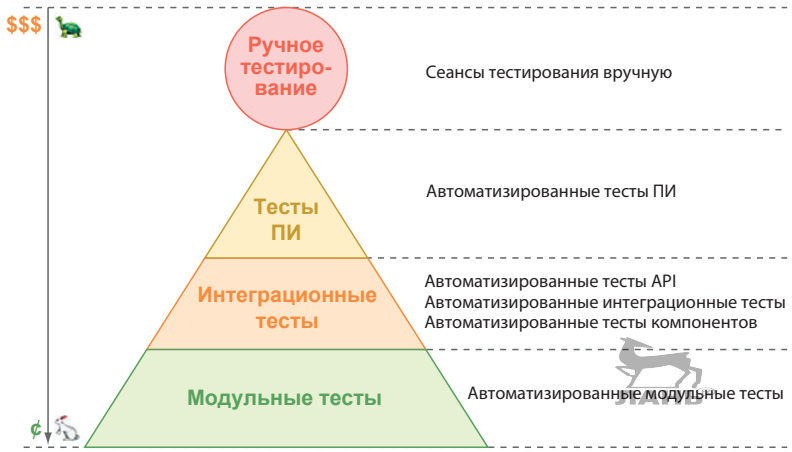


Рис. 11.1. Пирамида тестирования

В обычных приложениях, размещаемых на сервере, для запуска автоматизированных тестов чаще требуется использовать отдельный сервер тестирования, чтобы в ходе тестирования не уничтожить производственные данные. В результате значительная часть затрат на тестирование сопряжена с организацией тестовой инфраструктуры, включая настройку сервера, идентичного промышленному, импортирование базы данных, затраты на разработку и т. д.

В бессерверном окружении затраты на проведение испытаний существенно снижаются, в основном из-за отсутствия необходимости развертывать и настраивать отдельные серверы. В результате сокращаются трудозатраты разработчиков. Это сэкономленное время можно использовать для разработки дополнительных тестов и охвата большего объема кода. Обновленная пирамида тестирования для бессерверных приложений, показывающая разницу в стоимости тестирования, представлена на рис. 11.2. Назовем ее *пирамидой бессерверного тестирования*.

11.2. Подходы к тестированию бессерверных приложений

Выбор в пользу разработки бессерверного приложения – отличная идея, потому что отпадает необходимость в создании инфраструктуры. Но с точки зрения тестирования эта выгода превращается в проблему. Отсутствие контроля над инфраструктурой требует переосмысления способов тестирования. На

первый взгляд может показаться, что отсутствие контроля над инфраструктурой означает отсутствие ответственности за работу служб AWS или за сбои в работе сети. Но это неправильно. Отсутствие контроля над инфраструктурой не означает отсутствия ответственности, если она работает неправильно. Наши клиенты не понимают разницы между сбоем службы AWS и сбоем нашего приложения. Мы несем ответственность за все и обязаны проверить, насколько хорошо наше приложение обрабатывает эти ситуации.

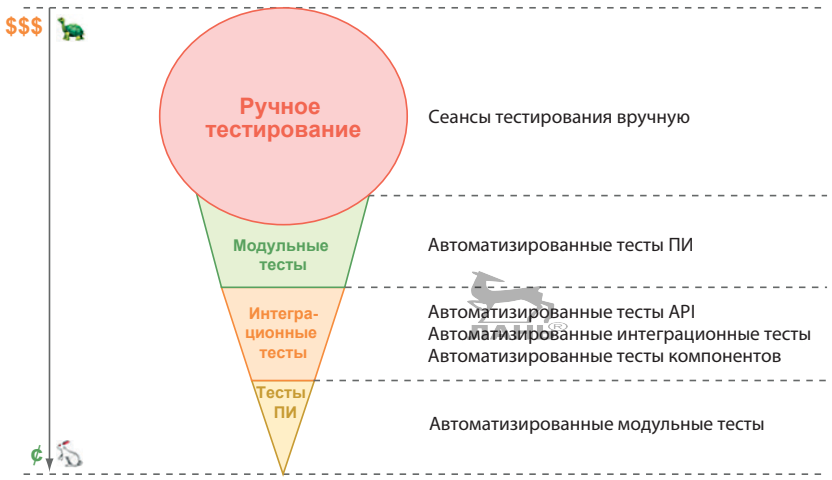


Рис. 11.2. Пирамида бессерверного тестирования

Следующие пошаговые инструкции помогут вам вспомнить эти ситуации при разработке тестов. Некоторые из вас, возможно, уже использовали их в какой-нибудь другой форме.

1. Составить список всех отдельных *задач*.
   Под задачей понимается одна функция или один фрагмент кода, отвечающий за одну операцию. В нашем примере это может быть вычисление скидки.
2. Протестировать каждую задачу в отдельности.
3. Протестировать, как эти задачи работают вместе (их интеграцию). Например, проверить, как скидка влияет на сумму, которую вы списываете с кредитной карты клиента.
4. Протестировать каждую *группу взаимодействующих задач* отдельно.
5. Составить список всех *сквозных процессов*.
   Под сквозным процессом понимается один полный процесс, выполняемый приложением. Примером может служить процесс, включающий загрузку сайта, вывод списка пицц, выбор одной из них, оформление заказа и оплату. Составление такого списка поможет вам получить более полное представление о приложении.
6. Протестировать каждый из процессов в списке.

Такой подход может показаться логичным, но количество ошибок в современных приложениях говорит нам, что логичные подходы не всегда используются в повседневной практике.

> **ПРИМЕЧАНИЕ.** Сквозное тестирование бессерверных приложений производится точно так же, как обычных приложений. Поэтому последние два шага оказываются за рамками этой книги. У нас нет доступа к веб-сайту тетушки Марии или мобильному устройству пользователя, соответственно, разработка комплексных сквозных тестов не входит в наши обязанности. Однако эти тесты играют очень важную роль, потому что проверяют работу бессерверного приложения в целом. Узнать больше о сквозном тестировании можно по адресу https://medium.freecodecamp.org/why-end-to-end-testing-is-important-for-your-team-cb7eb0ec1504.

11.3. Подготовка

Бессерверное приложение Node.js все еще остается приложением Node.js, а это означает, что для тестирования Pizza API можно использовать те же инструменты, которые применяются для тестирования приложений Node.js. В этой главе мы используем Jasmine, один из самых популярных фреймворков тестирования для Node.js, но вообще вы можете использовать любые другие инструменты, например Mocha, Tape или Jest.

> **Фреймворк тестирования Jasmine**
>
> Jasmine – это фреймворк тестирования для JavaScript. Он не зависит от других JavaScript-фреймворков и не требует использовать объектную модель документа (Document Object Model, DOM), поэтому его можно применять для тестирования сценариев в браузере и приложений Node.js. Jasmine имеет ясный и очевидный синтаксис, упрощающий тестирование. Узнать больше о нем можно по адресу https://jasmine.github.io.

Тесты для Jasmine называют *спецификациями*, поэтому мы тоже будем называть их так далее в этой главе. Спецификация (тест) – это функция на JavaScript, которая определяет, что должна вернуть проверяемая часть приложения. Спецификации объединяются в *наборы*, что позволяет организовать группы спецификаций. Например, для проверки поведения формы можно определить набор для проверки ввода и сгруппировать в нем спецификации, проверяющие ввод в отдельные поля.

Для запуска спецификаций в Jasmine используются *сценарии запуска* (runner). Запустить можно все спецификации, выбранные или входящие в определенный набор. Перед разработкой тестов необходимо подготовить проект

для модульного тестирования. Для этого нужно создать папку, где будут храниться спецификации, и написать сценарий, который будет запускать спецификации.

Следуя соглашениям об именах в Jasmine, создайте папку `spec` в проекте Pizza API. В ней будут храниться все спецификации, включая спецификации для модульного и интеграционного тестирований. Здесь же будет храниться конфигурация для сценария запуска и некоторых вспомогательных сценариев, например генерирующих фиктивные HTTP-запросы. Структура папки приложения со спецификациями, которые мы создадим в этой главе, показана на рис. 11.3.

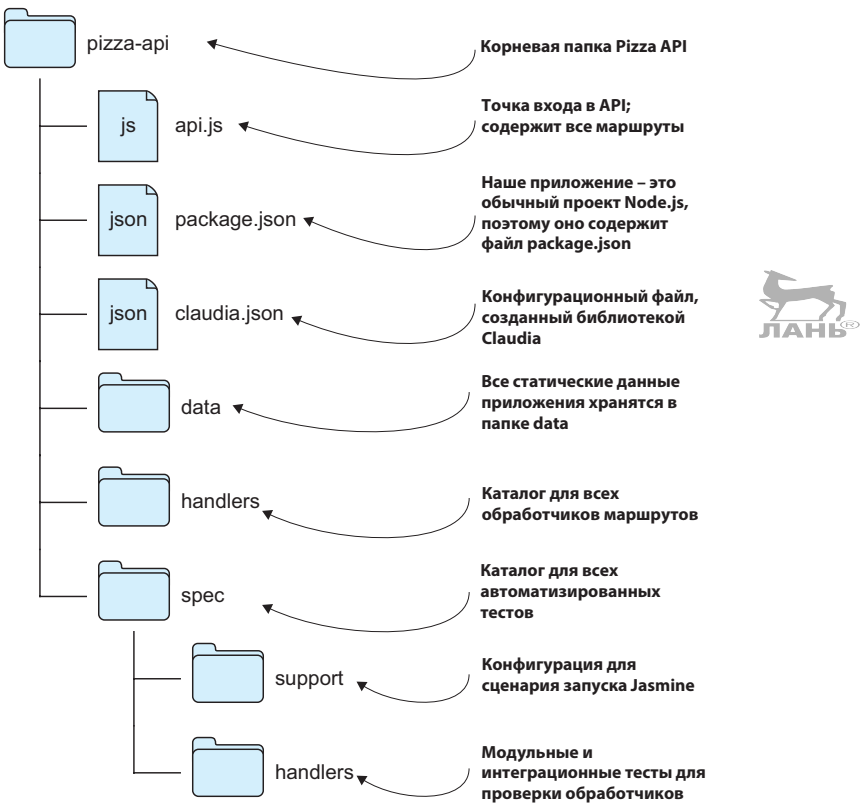


Рис. 11.3. Структура каталогов проекта Pizza API со спецификациями

Чтобы настроить сценарий запуска Jasmine, создайте папку `support` в папке `specs`. Внутри этой папки создайте файл `jasmine.json`. В нем будут храниться настройки для сценария запуска.

Как показано в листинге 11.1, мы должны определить местоположение спецификаций относительно корневой папки проекта и шаблон, который Jasmine будет использовать для поиска файлов спецификаций. В нашем случае это может быть любой файл с именем, оканчивающимся на «spec.js» или «Spec.js».

Листинг 11.1. Конфигурация Jasmine

```
{
  "spec_dir": "specs",
  "spec_files": [
    "**/*[sS]pec.js"
  ]
}
```

Путь к папке со спецификациями относительно корневой папки проекта.

Имена файлов всех спецификаций оканчиваются на «spec.js» или «Spec.js».

Теперь определим, как Jasmine будет выполнять тестирование. Нам нужно, чтобы фреймворк брал настройки из файла `jasmine.json` и позволял запускать отдельные спецификации или их наборы. Наконец, нам нужно, чтобы фреймворк работал в многословном режиме и в процессе работы выводил полную информацию о каждой конкретной спецификации.

Для этого создайте еще один файл с именем `jasmine-runner.js` в той же папке и откройте его в текстовом редакторе.

В начале файла импортируйте `jasmine` и `SpecReporter` из NPM-пакета `jasmine-spec-reporter`. Затем создайте экземпляр класса `Jasmine`.

Следующий шаг – цикл по аргументам командной строки. Мы можем пропустить первые два аргумента, потому что они определяют пути к Node.js и к текущему файлу сценария. Для всех остальных проверяем, является ли текущий аргумент строкой `'full'`, и в этом случае замещаем генератор отчетов по умолчанию генератором отчетов спецификации. Если аргумент является фильтром, запускаем только спецификации, содержащие указанный фильтр.

В заключение загружаем конфигурацию вызовом метода `loadConfigFile` и запускаем тестирование с указанными фильтрами.

Содержимое файла `jasmine-runner.js` показано в листинге 11.2.

Листинг 11.2. Сценарий запуска Jasmine

```
'use strict'

const SpecReporter = require('jasmine-spec-reporter').SpecReporter
const Jasmine = require('jasmine')
const jrunner = new Jasmine()
let filter

process.argv.slice(2).forEach(option => {
  if (option === 'full') {
    jrunner.configureDefaultReporter({ print() {} })
    jasmine.getEnv().addReporter(new SpecReporter())
  }

  if (option.match('^filter='))
```

Импортировать библиотеку SpecReporter.

Импортировать библиотеку jasmine.

Создать экземпляр Jasmine.

Создать переменную filter, которая будет использоваться позже.

Получить все аргументы командной строки, кроме двух первых, и выполнить цикл по ним.

Если получен аргумент full, заменить генератор отчетов по умолчанию генератором отчетов спецификации.

Если получен аргумент filter, сохранить значение фильтра в переменной filter.

```
    filter = option.match('^filter=(.*)')[1]
})

jrunner.loadConfigFile()
jrunner.execute(undefined, filter)
```

Загрузить конфигурацию из файла jasmine.json.

Запустить тестирование, передав полученные фильтры.

Теперь мы можем запустить свои спецификации командой `node spec/support/jasmine-runner.js`. Она выведет результаты работы спецификаций в окно терминала и добавит зеленую точку для каждой успешно выполнившейся спецификации. Чтобы вместо зеленых точек увидеть сообщения, генерируемые спецификациями, можно выполнить команду `node spec/support/jasmine-runner.js full`.

Чтобы упростить запуск спецификаций, можно добавить в файл `package.json` сценарий `test`. Это позволит запускать спецификации более короткой командой `npm test` или даже `npm t`. Добавьте следующий сценарий в файл `package.json`:

```
"test": "node specs/support/jasmine-runner.js"
```

Чтобы запустить спецификации в режиме вывода подробных сообщений, выполните команду `npm t-- full`. Два минуса (`--`) являются обязательными, и за ними должен следовать пробел, потому что следующий за ними параметр – в данном случае `full` – не является параметром NPM и должен передаваться непосредственно в Jasmine.

> **СОВЕТ.** Есть возможность усовершенствовать код, добавив еще два сценария NPM. Во-первых, если у вас установлена утилита `eslint`, ее можно автоматически запускать перед тестами, добавив в файл `package.json` сценарий `pretest`, например так:
>
> ```
> "pretest": "eslint lib spec *.js"
> ```
>
> Также, если вы пользуетесь отладчиком Node.js, вам очень даже может пригодиться сценарий `debug`:
>
> ```
> "debug": "node debug spec/support/jasmine-runner.js"
> ```
>
> Этот сценарий будет запускать тесты в отладчике Node.js. Более подробную информацию ищите по адресу https://nodejs.org/api/debugger.html.

11.4. Модульные тесты

В основании пирамиды тестирования лежит слой модульного тестирования, который состоит из модульных тестов. Цель модульного тестирования – изолировать каждую часть приложения и показать, что каждая в отдельности работает должным образом.

Размер единицы тестирования зависит от приложения; она может быть такой же маленькой, как функция, или большой, как класс или весь модуль.

Наименьшая единица кода в Pizza API, которую имеет смысл изолировать и тестировать, – функция-обработчик. Начнем с обработчика `getPizzas`.

Единственное, что связывает обработчик `getPizzas` с внешним миром, – это файл `pizzas.json`. Даже притом что это статический файл, он является частью внешнего мира и не должен участвовать в модульном тестировании. Чтобы подготовить обработчик к модульному тестированию, нужно разрешить функции получать произвольный список пицц, который будет использоваться взамен списка из `pizzas.json`. Тем самым мы гарантируем, что ваш модульный тест будет работать даже после изменения файла `pizzas.json`.

Как показано в листинге 11.3, для этого следует добавить параметр `pizzas` в обработчик `getPizzas`, который по умолчанию получает содержимое файла `pizzas.json`.

Листинг 11.3. Измененный обработчик getPizzas

```
'use strict'

const listOfPizzas = require('../data/pizzas.json')

function getPizzas(pizzaId, pizzas = listOfPizzas) {
```

Импортировать список пицц.

Передать список во втором аргументе и по умолчанию присвоить ему значение listOfPizzas.

Теперь обработчик готов к тестированию, и мы можем начать писать спецификации. Для этого создайте файл с именем `get-pizzas.spec.js` в папке `spec/handlers`.

В этом файле импортируйте обработчик и создайте массив пицц. Он должен содержать как минимум две пиццы с именами и идентификаторами и может выглядеть, как показано ниже:

```
const pizzas = [{
  id: 1,
  name: 'Capricciosa'
}, {
  id: 2,
  name: 'Napoletana'
}]
```

Теперь опишите спецификацию, используя функцию `describe`. Описание должно быть коротким и простым для понимания; например:

```
describe('Get pizzas handler', ()
        => { ... . })
```

> **СОВЕТ.** Фреймворк Jasmine не требует явно импортировать функции `describe`, `it` и `expect`, потому что они автоматически внедряются как глобальные переменные. Но если вы используете утилиту `eslint`, не забудьте сообщить ей, что функции

Jasmine являются глобальными и не надо выводить предупреждения, что они не определены.

Блок `describe` должен содержать несколько спецификаций. В случае с такой простой функцией, как `getPizzas`, мы должны проверить:

- получение списка всех пицц;
- получение одной пиццы по идентификатору;
- ошибку получения пиццы по несуществующему идентификатору.

Каждая спецификация – это отдельный блок, определяемый путем вызова функции `it`. Эта функция принимает два параметра: описание спецификации и функцию, определяющую эту спецификацию. Помните: описания должны быть короткими, но четкими, чтобы можно было легко понять, что тестируется.

Каждая спецификация содержит одно или несколько *ожиданий*, которые проверяют состояние кода. Ожидания проверяют соответствие текущего значения *ожидаемому*. Ожидания определяются с использованием инструкций `expect`.

ПРИМЕЧАНИЕ. Дополнительную информацию об использовании Jasmine вместе с Node.js можно найти в официальной документации по адресу https://jasmine.github.io/api/2.8/global.html.

В своей первой спецификации мы убедимся, что обработчик возвращает список всех пицц, когда идентификатор пиццы не указан. Для этого вызовем обработчик без первого аргумента, но при этом мы должны передать список пицц во втором аргументе. Сделать это можно, передав обработчику `undefined` и список пицц соответственно. Вот соответствующая спецификация:

```
it('should return a list of all pizzas if called without pizza ID', () => {
  expect(underTest(undefined, pizzas)).toEqual(pizzas)
})
```

Чтобы проверить получение пиццы по существующему идентификатору, нужно передать оба идентификатора (`1` и `2`) и список пицц и сопоставить результаты с ожидаемыми значениями – первой и второй пиццами из фиктивного массива пицц. Вот соответствующая спецификация:

```
it('should return a single pizza if an existing ID is passed as the first
    parameter', () => {
  expect(underTest(1, pizzas)).toEqual(pizzas[0])
  expect(underTest(2, pizzas)).toEqual(pizzas[1])
})
```

В последней спецификации для модульного тестирования обработчика `getPizzas` можно проявить творческий подход и передать любой несуществующий идентификатор. Например, некоторые крайние значения, то есть числа, которые меньше и больше любых существующих идентификаторов, а также проверить некоторые другие значения, такие как строки, массивы или объекты.

Ниже показано, как могла бы выглядеть такая спецификация:

```
it('should throw an error if nonexistent ID is passed', () => {
  expect(() => underTest(0, pizzas)).toThrow(
    'The pizza you requested was not found')
  expect(() => underTest(3, pizzas)).toThrow(
    'The pizza you requested was not found')
  expect(() => underTest(1.5, pizzas)).toThrow(
    'The pizza you requested was not found')
  expect(() => underTest(42, pizzas)).toThrow(
    'The pizza you requested was not found')
  expect(() => underTest('A', pizzas)).toThrow(
    'The pizza you requested was not found')
  expect(() => underTest([], pizzas)).toThrow(
    'The pizza you requested was not found')
})
```

В листинге 11.4 показан результат объединения всех этих спецификаций в модульные тесты для обработчика `getPizzas`.

Листинг 11.4. Модульные тесты для обработчика getPizzas

```
'use strict'

const underTest = require('../../handlers/get-pizzas')
const pizzas = [{
  id: 1,
  name: 'Capricciosa'
}, {
  id: 2,
  name: 'Napoletana'
}]

describe('Get pizzas handler', () => {
  it('should return a list of all pizzas if called without pizza ID', () => {
    expect(underTest(undefined, pizzas)).toEqual(pizzas)
  })

  it('should return a single pizza if an existing ID is passed as the first
```

Импортировать обработчик getPizzas.

Создать фиктивный список пицц.

Описание группы спецификаций.

Спецификация для проверки случая вызова getPizzas без идентификатора.

Ожидается, что в случае вызова без идентификатора getPizzas вернет список всех пицц.

```
      parameter', () => {
    expect(underTest(1, pizzas)).toEqual(pizzas[0])
    expect(underTest(2, pizzas)).toEqual(pizzas[1])
  })

  it('should throw an error if nonexistent ID is passed', () => {
    expect(() => underTest(0, pizzas)).toThrow(
      'The pizza you requested was not found')
    expect(() => underTest(3, pizzas)).toThrow(
      'The pizza you requested was not found')
    expect(() => underTest(1.5, pizzas)).toThrow(
      'The pizza you requested was not found')
    expect(() => underTest(42, pizzas)).toThrow(
      'The pizza you requested was not found')
    expect(() => underTest('A', pizzas)).toThrow(
      'The pizza you requested was not found')
    expect(() => underTest([], pizzas)).toThrow(
      'The pizza you requested was not found')
  })
})
```

Спецификация для проверки случая вызова getPizzas с действительным идентификатором.

Спецификация для проверки случая вызова getPizzas с недействительным идентификатором.

Перейдите в папку проекта и выполните команду `npm test` в терминале. Вывод этой команды, как показано листинге 11.5, указывает, что одна из спецификаций потерпела неудачу.

Листинг 11.5. Вывод команды npm test

```
> node spec/support/jasmine-runner.js

Started
..F

Failures:
1) Get pizzas handler should throw an error if nonexistent ID is passed
  Message:
    Expected function to throw an exception.
  Stack:
    Error: Expected function to throw an exception.
        at UserContext.it (~/pizza-api/spec/handlers/get-pizzas-spec.
      js:26:40)

3 specs, 1 failure
Finished in 0.027 seconds
```

Спецификация, потерпевшая неудачу, определяет появление ошибки в функции AWS Lambda, создающей проблему в работе. Всегда важно проверять крайние случаи в спецификациях, потому что это может сэкономить массу времени, расходуемого на отладку и исследование журналов CloudWatch.

Получив идентификатор с нулевым значением, обработчик `getPizzas` возвращает список всех пицц вместо ошибки, потому что 0 в JavaScript – это ложное значение и соответствует условию:

```
if (!pizzaId)
  return pizzas
```

Чтобы устранить эту проблему, нужно изменить условие и проверять в нем аргумент на равенство значению `undefined`, как показано ниже:

```
if (typeof pizzaId === 'undefined')
  return pizzas
```

Теперь, внеся изменение в обработчик `getPizzas`, повторно запустим спецификации командой `npm test`. На этот раз все они должны выполняться успешно, а вывод команды должен выглядеть, как показано в листинге 11.6.

Листинг 11.6. Результат выполнения спецификаций после внесения исправлений в обработчик

```
> node spec/support/jasmine-runner.js

Started
...

3 specs, 0 failures
Finished in 0.027 seconds
```

Имейте в виду, что успешное выполнение спецификаций не гарантирует отсутствия ошибок в коде, однако при достаточном количестве значимых спецификаций, охватывающих существенную часть кода, количество ошибок, перекочевавших в действующую версию, будет значительно меньше. Но как протестировать обработчики, которые нельзя легко изолировать, например подключающиеся к таблице в DynamoDB? В таких случаях вам помогут фиктивные функции.

11.5. Использование имитаций для тестирования бессерверных функций

В отличие от `getPizzas`, большинство других обработчиков в Pizza API взаимодействует с базой данных или отправляет HTTP-запросы. Чтобы протестировать эти обработчики изолированно, нужно смоделировать внешние взаимодействия.

Прием моделирования широко используется в модульном тестировании и заключается в создании фиктивных объектов, имитирующих поведение действительных объектов. Используя имитации вместо внешних объектов и функций, которые применяет тестируемый обработчик, мы можем изолировать обработчик и проверить его поведение.

Для примера попробуем протестировать более сложный обработчик, такой как `createOrder`. Для этого нам понадобятся две имитации:

- первое, что следует сымитировать, – это отправка HTTP-запроса службе доставки Some Like It Hot, потому что крайне нежелательно посылать настоящий запрос в процессе тестирования. Служба Some Like It Hot – это внешняя зависимость, неподконтрольная нам, и у нас нет доступа к ее тестовой версии. Любой запрос на доставку, который мы отправим в процессе тестирования, может вызвать реальные производственные проблемы;
- также, чтобы полностью изолировать `createOrder` от всех внешних зависимостей, нам нужно сымитировать класс `DocumentClient`. Если вы решите протестировать интегрированный обработчик, вам придется настроить тестовую базу данных.

Имитации важны, потому что модульные тесты должны выполняться быстрее интеграционных и сквозных тестов. На выполнение всего комплекта спецификаций должно уходить лишь несколько секунд, а не минут и тем более часов. Кроме того, модульное тестирование обходится намного дешевле, потому что не нужно платить за инфраструктуру, чтобы проверить, работает ли логика обработчиков, как ожидалось.

После подготовки имитаций HTTP-запросов и взаимодействий с DynamoDB тестирование обработчика будет выглядеть, как показано на рис. 11.4.

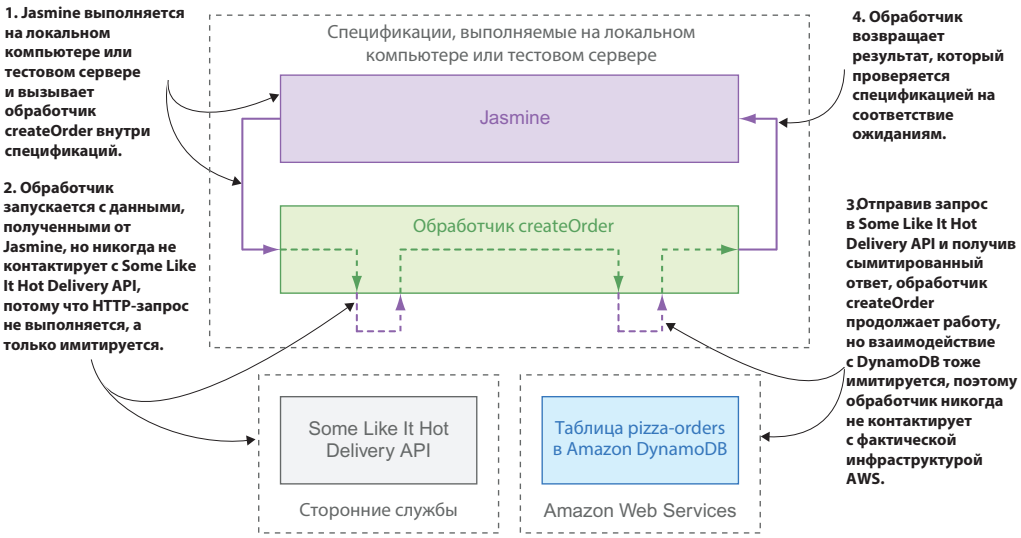


Рис. 11.4. Процесс модульного тестирования обработчика createOrder

Чтобы реализовать спецификацию для модульного тестирования обработчика `createOrder`, создайте файл `create-order.spec.js` в папке `specs/handlers`, внутри проекта Pizza API. Внутри файла импортируйте обработчик и добавьте блок `describe`, чтобы объединить нужные спецификации в группу.

Пока файл должен выглядеть, как показано ниже:

```
const underTest = require('../../handlers/create-order')

describe('Create order handler', () => {
  // Здесь будут находиться спецификации
})
```


Теперь сымитируем HTTP-запрос. В Node.js это можно сделать несколькими способами. Например, можно использовать полноценный модуль, такой как Sinon (http://sinonjs.org) или Nock (https://github.com/node-nock/nock), или даже написать свой модуль.

Занимающимся разработкой приложений на основе Node.js и бессерверных функций мы всегда рекомендуем использовать небольшие и специализированные модули. Модуль `fake-http-request` – как раз такой небольшой модуль Node.js, имитирующий отправку запросов HTTP и HTTPS. Его можно установить из NPM и сохранить как зависимость для разработки, выполнив команду `npm install fake-http-request --save-dev`.

В нашем новом модульном тесте нам также потребуется модуль `https`, потому что `fake-http-request` использует его для выявления имитируемых HTTP-запросов.

> **ПРИМЕЧАНИЕ.** Модуль `https` необходим для имитации HTTPS-соединения со службой доставки Some Like It Hot. Для имитации HTTP-запросов вместо HTTPS следует использовать модуль `http`.

Чтобы задействовать модуль `fake-http-request`, нужно использовать функции Jasmine `beforeEach` и `afterEach`, позволяющие выполнить операции до и после запуска каждой спецификации. Чтобы установить и удалить модуль, добавьте следующий фрагмент в блок `describe`:

```
beforeEach(() => fakeHttpRequest.install('https'))
afterEach(() => fakeHttpRequest.uninstall('https'))
```

Теперь, организовав тестирование HTTPS-запросов, смоделируем класс `DocumentClient`. Для этого импортируем `aws-sdk`, а затем заменим класс `DocumentClient` функцией-шпионом из Jasmine. Не забудьте связать функцию `Promise.resolve`, иначе будет использоваться другая ссылка `this`, что приведет к неудаче.

Так как для создания объектов `DocumentClient` в AWS SDK используется прототип, есть возможность заменить `DocumentClient` своей функцией-шпионом, добавив следующий код в блок `beforeEach`:

```
AWS.DynamoDB.DocumentClient.prototype = docClientMock
```

Функции-шпионы в Jasmine

Согласно документации к фреймворку Jasmine, «в Jasmine поддерживаются тестовые функции, которые называют шпионами. Функция-шпион может подменить любую функцию и отслеживать любые обращения к ней и ее аргументы. Функции-шпионы существуют только в блоках describe и it, где они определены, и автоматически исчезают по завершении выполнения спецификаций». Узнать больше о функциях-шпионах в Jasmine можно по адресу https://jasmine.github.io/2.0/introduction.html#section-Spies.

Теперь содержимое файла `create-order.spec.js` должно выглядеть, как показано в листинге 11.7.

Листинг 11.7. Основа модульного теста для обработчика `createOrder`

```
'use strict'

const underTest = require('../../handlers/create-order')
const https = require('https')
const fakeHttpRequest = require('fake-http-request')
const AWS = require('aws-sdk')
let docClientMock

describe('Create order handler', () => {
  beforeEach(() => {
    fakeHttpRequest.install('https')

    docClientMock = jasmine.createSpyObj('docClient', {
      put: { promise: Promise.resolve.bind(Promise) },
      configure() { }
    })
    AWS.DynamoDB.DocumentClient.prototype = docClientMock
  })

  afterEach(() => fakeHttpRequest.uninstall('https'))

    // Здесь будут находиться спецификации

  })
```

Импортировать обработчик.

Импортировать модули https и fake-http-request.

Импортировать aws-sdk.

Переменная для хранения имитации объекта DocumentClient.

Установить библиотеку fake-http-request для имитации запросов https.

Создать объект-шпион для имитации DocumentClient.

Имитации функций put и configure.

Заменить DocumentClient объектом-шпионом.

Удалить библиотеку fake-http-request.

Обработчик `createOrder` сложнее `getPizzas`, поэтому для его тестирования требуется больше спецификаций. Но мы ограничимся спецификациями, тестирующими наиболее важные аспекты поведения `createOrder`:

- отправка запроса `POST` службе доставки Some Like It Hot;
- реакция на успех и неудачу выполнения запроса к Some Like It Hot Delivery API;
- вызов `DocumentClient` для сохранения заказа только в случае успешного ответа от Some Like It Hot Delivery;
- завершение `Promise` с признаком успеха, если оба запроса – к Some Like It Hot Delivery API и `DocumentClient` – выполнились благополучно;
- завершение `Promise` с признаком ошибки, если какое-то из взаимодействий потерпело неудачу;
- проверка ввода.

Желающие могут добавить свои спецификации и проверить дополнительные крайние ситуации. Чтобы не увеличивать число страниц в этой главе до неразумных пределов, мы обсудим только наиболее важные моменты, а полный код `create-order.spec.js` вы найдете в примерах исходного кода к книге.

Первая спецификация, которую мы добавим в блок `it`, проверит отправку запроса POST службе доставки Some Like It Hot. Определим для нее короткое и понятное описание; например, «should send POST request to Some Like It Hot Delivery API» (в Some Like It Hot Delivery API должен быть отправлен запрос POST).

В этой спецификации вызовем обработчик `createOrder` с допустимыми данными и с помощью модуля `https` проверим, был ли отправлен запрос с ожидаемыми телом и заголовками.

Модуль `fake-http-request` добавляет метод `pipe` в `https.request`, который можно использовать для проверки всех свойств HTTPS-запроса. Например, можно убедиться, что число отправленных запросов равно 1, потому что службе доставки должен отправляться только один запрос. Также можно проверить правильность атрибутов в `https.request`, включая метод, путь, тело и заголовки.

> **ПРИМЕЧАНИЕ.** Имейте в виду, что тело запроса содержит самый обычный текст и его нужно преобразовать в строковый объект перед проверкой; иначе спецификация потерпит неудачу, попытавшись сравнить данные разных типов: объект и строку.

Спецификация, выполняющая эти проверки, показана в листинге 11.8.

> **СОВЕТ.** Если потребуется проверить лишь несколько свойств в большом объекте, вместо прямой проверки этих свойств используйте функцию `jasmine.objectContaining`, позволяющую проверить указанное подмножество свойств.

Листинг 11.8. Имитация запроса POST

Блок it с описанием спецификации.

```
it('should send POST request to Some Like It Hot Delivery API', (done) => {
```

```
  underTest({
    body: {
      pizza: 1,
      address: '221b Baker Street'
    }
  })

  https.request.pipe((callOptions) => {
    expect(https.request.calls.length).toBe(1)
    expect(callOptions).toEqual(jasmine.objectContaining({
      protocol: 'https:',
      slashes: true,
      host: 'some-like-it-hot-api.effortless-serverless.com',
      path: '/delivery',
      method: 'POST',
      headers: {
        Authorization: 'aunt-marias-pizzeria-1234567890',
        'Content-type': 'application/json'
      },
      body: JSON.stringify({
        pickupTime: '15.34pm',
        pickupAddress: 'Aunt Maria Pizzeria',
        deliveryAddress: '221b Baker Street',
        webhookUrl: 'https://g8fhlgccof.execute-api.eu-central-1.amazonaws.
com/latest/delivery'
      })
    }))
    done()
  })
})
```

Вызвать тестируемый обработчик.

Использовать https.request.pipe для проверки отправки запроса.

Убедиться, что отправлен только один запрос.

Сравнить параметры в запросе с ожидаемыми.

Сравнить параметры в запросе с ожидаемыми.

Сообщить фреймворку Jasmine, что выполнение асинхронной спецификации завершилось.

Следующая важная проверка – вызов `DocumentClient` должен состояться только после успешного завершения HTTP-запроса. Для этого нужно сымитировать получение сообщения об успехе от Some Like It Hot Delivery API, добавив строку `https.request.calls[0].respond(200, 'Ok', '{}')` в метод `https.request.pipe`.

Обработчик `createOrder` возвращает объект `Promise`, поэтому для проверки вызова имитации `DocumentClient` можно использовать `.then`.

Не забудьте добавить вызов `done()` после инструкции `expect` и вызвать `done.fail()`, если выполнение объекта `Promise` завершилось с ошибкой; иначе спецификация будет продолжать выполняться, пока не будет прервана фреймворком Jasmine по тайм-ауту.

Спецификация для проверки вызова `DocumentClient` представлена в листинге 11.9.

Листинг 11.9. Тестирование вызова DocumentClient

```
it('should call the DynamoDB DocumentClient.put
  if Some Like It Hot Delivery API request was successful', (done) => {
    underTest({
      body: { pizza: 1, address: '221b Baker Street' }
    })
    .then(() => {
      expect(docClientMock.put).toHaveBeenCalled()
      done()
    })
    .catch(done.fail)
      https.request.pipe((callOptions) => https.request.calls[0].respond(200,
          'Ok', '{}'))
})
```

← Вызвать тестируемый обработчик с допустимыми данными.

← Убедиться, что docClientMock.put вызывается в случае успешного выполнения Promise.

← Сообщить фреймворку Jasmine, что асинхронная спецификация завершилась успехом.

← Сообщить фреймворку Jasmine, что асинхронная спецификация завершилась ошибкой, если Promise завершился с ошибкой.

← Сымитировать успешное выполнение HTTP-запроса с кодом 200.

Другая похожая спецификация должна проверить, что `DocumentClient` никогда не вызывается, если HTTP-запрос потерпит неудачу. Вот основные отличия этой спецификации от предыдущей:

- спецификация должна завершаться с ошибкой, если выполнение `Promise` завершится успехом;
- спецификация должна проверить, что `docClientMock.put` не вызывается;
- библиотека `fake-http-request` должна вернуть ошибку (с кодом HTTP больше или равно 400).

Спецификация, выполняющая эти проверки, показана в листинге 11.10

Листинг 11.10. Проверка отсутствия обращений к DocumentClient в случае неудачного выполнения HTTP-запроса

```
it('should not call the DynamoDB DocumentClient.put
  if Some Like It Hot Delivery API request was not successful', (done) => {
    underTest({
      body: { pizza: 1, address: '221b Baker Street' }
    })
    .then(done.fail)
    .catch(() => {
      expect(docClientMock.put).not.toHaveBeenCalled()
      done()
    })

  https.request.pipe((callOptions) => https.request.calls[0].respond(500,
    'Server Error', '{}'))
})
```

← Сообщить фреймворку Jasmine, что асинхронная спецификация завершилась ошибкой, если Promise завершился успехом.

← Убедиться, что docClientMock.put не вызывался, если Promise завершился с ошибкой.

← Вернуть код ответа 500.

Если теперь выполнить команду `npm test` или `npm t`, спецификации должны выполниться благополучно.

ПРИМЕЧАНИЕ. Полный код спецификаций можно найти в примерах исходного кода к книге.

11.6. Интеграционные тесты

Интеграционные тесты – еще один вид тестов; они даже более важны для бессерверных функций, размер которых превышает несколько строк кода. Интеграционные тесты, в отличие от модульных, используют фактические связи с другими частями системы. Но при этом они все еще могут и должны использовать имитации сторонних библиотек, неподконтрольных вам. Например, едва ли кто-то из вас пожелает, чтобы ваши автоматизированные тесты взаимодействовали с платежной системой.

Как показано на рис. 11.5, интеграционные тесты для обработчика `createOrder` могут имитировать работу внешней службы Some Like It Hot, потому что отправка HTTP-запросов сторонней службе может отрицательно сказаться на реальных людях. Но фактическое взаимодействие с таблицей в DynamoDB вполне допустимо и даже желательно.

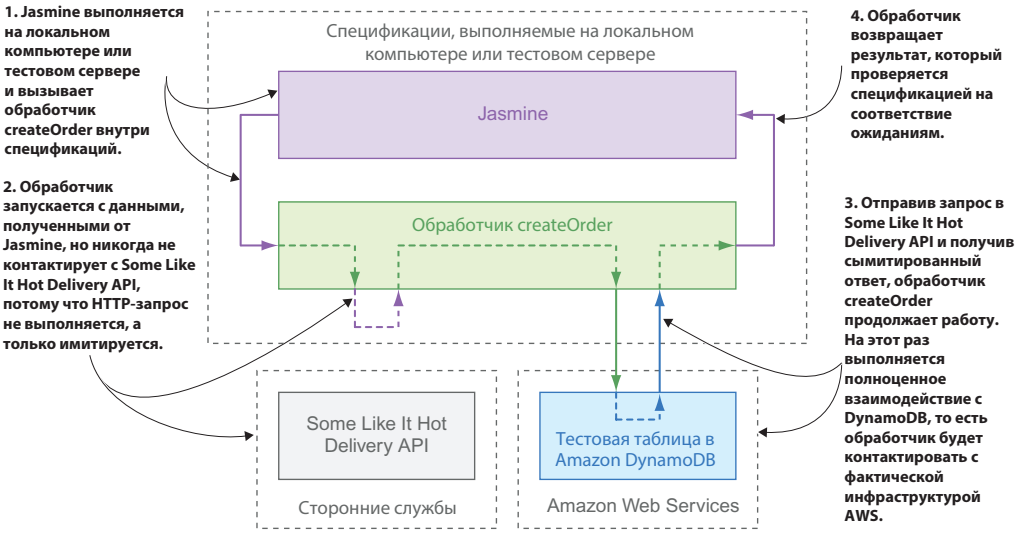


Рис. 11.5. Последовательность шагов интеграционного тестирования обработчика createOrder

Ниже перечислена последовательность шагов, которые могли бы выполняться в ходе интеграционного тестирования обработчика `createOrder`:

1) создать новую таблицу в DynamoDB перед запуском каждой спецификации;

2) создать имитацию соединения со службой доставки Some Like It Hot перед запуском каждой спецификации;
3) запустить спецификацию;
4) удалить имитацию соединения со службой доставки Some Like It Hot и выполнить следующую спецификацию (перейдя к шагу 2);
5) удалить тестовую таблицу из DynamoDB по завершении всех спецификаций.

> **СОВЕТ.** Создание и удаление таблицы в DynamoDB тоже можно выполнять до и после каждой спецификации, но поскольку эта операция может потребовать до нескольких секунд, для экономии времени предпочтительнее использовать одну и ту же таблицу для всех спецификаций из набора интеграционных тестов.

Поскольку имеется всего несколько обработчиков, модульные и интеграционные тесты можно хранить в одной папке. Просто давайте им такие имена, чтобы их легко можно было отличить друг от друга. Например, интеграционные тесты для обработчика `createOrder` могут находиться в файле `create-order-integration.spec.js`.

Как показано в листинге 11.11 ниже, подготовка к интеграционному тестированию обработчика `createOrder` выполняется в несколько этапов.

Первый этап – импортирование всех необходимых модулей: тестируемого обработчика, `aws-sdk` (чтобы получить доступ к классу DynamoDB), `https` и `fake-http-request`.

Затем вам нужно сгенерировать имя для тестовой таблицы в DynamoDB. Конечно, можно придумать и каждый раз использовать одно и то же имя, но случайно сгенерированное имя будет иметь больше шансов оказаться уникальным. Также нужно увеличить тайм-аут в Jasmine, хотя бы до одной минуты, потому что создание и удаление тестовой таблицы в DynamoDB может занять некоторое время, а пятисекундный тайм-аут по умолчанию недостаточно длинный.

> **ПРИМЕЧАНИЕ.** По умолчанию Jasmine ждет завершения асинхронной спецификации пять секунд, после чего возбуждает ошибку тайм-аута. Если время ожидания истечет до вызова `done`, текущая спецификация будет отмечена как потерпевшая неудачу и выполнение набора продолжится, как если бы была вызвана функция `done`.

Затем в функции Jasmine `beforeAll` нужно перед всеми тестами создать таблицу в DynamoDB. Имейте в виду, таблицы в DynamoDB создаются асинхронно, поэтому обязательно используйте функцию обратного вызова `done`, чтобы сообщить Jasmine о завершении операции. Если этого не сделать, выполнение спецификации начнется до того, как таблица будет готова.

Создать таблицу можно с помощью метода `createTable` класса `DynamoDB`. Ему нужно передать то же определение ключа, что и при создании таблицы `pizza-orders`, то есть использовать `orderId` в роли ключа.

Так как `createTable` возвращает объект `Promise`, который должен завершить выполнение до использования таблицы в DynamoDB, можно вызвать метод `waitFor` класса `DynamoDB` и дождаться создания таблицы до вызова функции `done`.

Удаление таблицы после тестирования должно производиться в функции `afterAll` вызовом метода `deleteTable` класса `DynamoDB` с последующим вызовом метода `waitFor`, чтобы дождаться завершения операции удаления таблицы. После этого можно вызвать функцию `done`.

Имитация HTTP-запросов к службе доставки Some Like It Hot выполняется точно так же, как в модульных тестах. Единственное отличие – имитировать следует только запросы к конкретному API; остальные запросы должны выполняться как есть, потому что класс `DynamoDB` использует их для взаимодействия с инфраструктурой AWS infrastructure. Для этого нужно передать в функцию `fakeHttpRequest.install` объект, содержащий тип запроса (в данном случае `https`), и объект регулярного выражения для сопоставления с доменным именем.

На данный момент содержимое файла `create-order-integration.spec.js` должно выглядеть, как показано в листинге 11.11.

Листинг 11.11. Интеграционный тест для обработчика `createOrder`

```
'use strict'

const underTest = require('../../handlers/create-order')
const AWS = require('aws-sdk')
const dynamoDb = new AWS.DynamoDB({
  apiVersion: '2012-08-10',
  region: 'eu-central-1'
})
const https = require('https')
const fakeHttpRequest = require('fake-http-request')

const tableName = `pizzaOrderTest${new Date().getTime()}`
jasmine.DEFAULT_TIMEOUT_INTERVAL = 60000

describe('Create order (integration)', () => {
  beforeAll((done) => {
    const params = {
      AttributeDefinitions: [{
        AttributeName: 'orderId',
        AttributeType: 'S'
      }],
      KeySchema: [{
        AttributeName: 'orderId',
        KeyType: 'HASH'
```

Импортировать тестируемый обработчик.

Импортировать aws-sdk.

Создать экземпляр класса DynamoDB.

Импортировать модули https и fake-http-request.

Сгенерировать имя для тестовой таблицы в DynamoDB.

Увеличить тайм-аут для Jasmine до одной минуты.

```
      }],
      ProvisionedThroughput: {
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1
      },
      TableName: tableName
    }

    dynamoDb.createTable(params).promise()
      .then(() => dynamoDb.waitFor('tableExists', {
        TableName: tableName
      }).promise())
      .then(done)
      .catch(done.fail)
  })

  afterAll(done => {
    dynamoDb.deleteTable({
      TableName: tableName
    }).promise()
      .then(() => dynamoDb.waitFor('tableNotExists', {
        TableName: tableName
      }).promise())
      .then(done)
      .catch(done.fail)
  })

  beforeEach(() => fakeHttpRequest.install({
    type: 'https',
    matcher: /some-like-it-hot-api/
  }))

  afterEach(() => fakeHttpRequest.uninstall('https'))

    // Здесь будут находиться спецификации

})
```

Создать новую таблицу в DynamoDB перед выполнением всех спецификаций.

Дождаться статуса tableExists.

Удалить таблицу в DynamoDB после выполнения всех спецификаций.

Дождаться статуса tableNotExists перед остановкой всех тестов.

Установить модуль fake-http-request для имитации запросов только к Some Like It Hot Delivery API.

Теперь, когда интеграционные тесты готовы к использованию, нужно обновить обработчик `createOrder`, чтобы организовать возможность динамического получения имени таблицы в DynamoDB. Сделать это можно, передавая имя таблицы во втором аргументе или определив имя в переменной окружения.

Проще всего передать имя таблицы во втором аргументе. Для этого измените обработчик `createOrder` так, чтобы он принимал имя таблицы, но не

забудьте назначить имя по умолчанию `pizza-orders`, чтобы не нарушить работоспособность существующего кода. Аргументы функции-обработчика `createOrder` должны выглядеть, как показано ниже:

```
function createOrder(request, tableName = 'pizza-orders') {
```

Последний и самый сложный шаг – добавление интеграционных спецификаций. Спецификации должны проверять все критические части интеграции обработчика с любыми другими компонентами системы или инфраструктуры.

Для экономии места в этой главе мы покажем только самую важную спецификацию, проверяющую запись данных в таблицу. Полный код спецификаций вы найдете в файле `create-order-integration.spec.js` в примерах исходного кода.

Как показано в листинге 11.12, чтобы проверить сохранение заказа в базе данных после получения положительного ответа от службы доставки Some Like It Hot, необходимо:

1) вызвать обработчик `createOrder` с допустимыми данными и именем тестовой таблицы в DynamoDB;
2) сымитировать положительный ответ от службы доставки Some Like It Hot и вернуть `deliveryId`;
3) когда объект `Promise`, возвращаемый обработчиком `createOrder`, завершится успехом, воспользоваться экземпляром класса `DynamoDB` и проверить присутствие в тестовой таблице записи с идентификатором, полученным от службы Some Like It Hot;
4) проверить правильность информации, возвращаемой методом `dynamoDb.getItem`;
5) отметить тест как выполнившийся успешно.

Листинг 11.12. Тестирование сохранения заказа в таблице DynamoDB

```
it('should save the order in the DynamoDB table
  if Some Like It Hot Delivery API request was successful', (done) => {
  underTest({
    body: { pizza: 1, address: '221b Baker Street' }
  }, tableName)
    .then(() => {
      const params = {
        Key: {
          orderId: {
            S: 'order-id-from-delivery-api'
          }
        },
        TableName: tableName
```

Вызвать обработчик с допустимыми данными и именем тестовой таблицы.

```
    }
    dynamoDb.getItem(params).promise()
      .then(result => {
        expect(result.Item.orderId.S).toBe('order-id-from-delivery-api')
        expect(result.Item.address.S).toBe('221b Baker Street')
        expect(result.Item.pizza.N).toBe('1')
        done()
      })
  })
  .catch(done.fail)
```

Получить запись из базы данных по идентификатору.

Проверить правильность данных и отметить тест как выполненный успешно.

Отметить тест как потерпевший неудачу, если объект Promise завершился с ошибкой.

```
https.request.pipe((callOptions) => https.request.calls[0].respond(200,
    'Ok', JSON.stringify({
  deliveryId: 'order-id-from-delivery-api'
})))
})
```

Сымитировать ответ от Some Like It Hot Delivery API и вернуть идентификатор deliveryID.

Если теперь выполнить команду `npm test`, вы заметите, что она выполняется довольно долго, но при этом все тесты, в том числе и интеграционные, должны завершиться успехом.

> **СОВЕТ.** При большом количестве интеграционных тестов можно создать тестовую таблицу DynamoDB заранее (до запуска тестов) и тем самым уменьшить время выполнения тестов.

Также можно заглянуть в веб-консоль AWS и убедиться, что таблица была успешно удалена из DynamoDB. Даже после добавления еще нескольких интеграционных тестов ваш ежемесячный счет на оплату услуг AWS для вашего приложения, созданного в этой книге, все равно должен составлять всего несколько центов.

11.7. Другие типы автоматизированных тестов

Выше вы видели, что модульные и интеграционные тесты в бессерверных приложениях аналогичны тестам в обычных, серверных приложениях Node.js. Как и ожидалось, основными отличительными чертами являются быстрота настройки тестовой инфраструктуры (настройка производится быстрее благодаря отсутствию необходимости настраивать сервер) и низкая стоимость услуг инфраструктуры (вам не придется платить за услуги, когда вы ими не пользуетесь).

Существует много других типов автоматизированных тестов, на которые также влияет бессерверный характер окружения. Например, нагрузочные и стресс-тесты теряют смысл в бессерверной архитектуре, потому что она автоматически масштабируется в установленных пределах. Такие тесты имеют

смысл, только если ваше приложение не является полностью бессерверным или вы не доверяете провайдеру услуг бессерверного окружения, но обсуждение этой проблемы выходит за рамки данной книги.

Еще один тип автоматизированных тестов, на которые может повлиять бессерверный характер окружения, – это тесты графического интерфейса пользователя. Хотя это и не очевидно, но бессерверная архитектура способна помочь ускорить тестирование графического интерфейса пользователя за счет применения *консольных браузеров*, таких как консольная версия Chrome и Phantom.js. Консольные браузеры – это обычные веб-браузеры, но не имеющие графического интерфейса; они запускаются из командной строки. Возможность запуска автоматических тестов графического интерфейса пользователя в Google Chrome на AWS Lambda уже привела к появлению множества новых инструментов, упрощающих тестирование графического интерфейса. Но, что еще более важно, эти инструменты на порядок ускоряют тесты и резко снижают их стоимость. Одним из инструментов, позволяющих запускать тесты графического интерфейса в AWS Lambda, является Appraise, использующий консольную версию Chrome для создания снимка экрана, а затем сравнивающий его с ожидаемым результатом. Узнать больше об этом инструменте можно на сайте http://appraise.qa.

11.8. В дополнение к тестам: приемы разработки бессерверных функций для упрощения их тестирования

Вы познакомились с основами тестирования бессерверных приложений, но это не значит, что вы познакомились со всеми возможными крайними случаями. Давайте вернемся к нашему обработчику сохранения заказов в базе данных.

Листинг 11.13. Текущий обработчик сохранения заказов в базе данных

```
function createOrder(request, tableName) {
  tableName = tableName || 'pizza-orders'

  const docClient = new AWS.DynamoDB.DocumentClient({
    region: process.env.AWS_DEFAULT_REGION
  })
  let userAddress = request && request.body && request.body.address;
  if (!userAddress) {
    const userData = request && request.context && request.context.authorizer
    && request.context.authorizer.claims;
    if (!userData)
      throw new Error()
    // console.log('User data', userData)
```

Загрузить DynamoDB.

```
    userAddress = JSON.parse(userData.address).formatted
  }

  if (!request || !request.body || !request.body.pizza || !userAddress)
    throw new Error('To order pizza please provide pizza type and address
      where pizza should be delivered')

  return rp.post('https://some-like-it-hot-api.effortless-serverless.com/
      delivery', {
    headers: {
      Authorization: 'aunt-marias-pizzeria-1234567890',
      'Content-type': 'application/json'
    },
    body: JSON.stringify({
      pickupTime: '15.34pm',
      pickupAddress: 'Aunt Maria Pizzeria',
      deliveryAddress: userAddress,
      webhookUrl: 'https://g8fhlgccof.execute-api.eu-central-1.amazonaws.com/
        latest/delivery',
    })
  })
    .then(rawResponse => JSON.parse(rawResponse.body))
    .then(response => {
      return docClient.put({
        TableName: tableName,
        Item: {
          cognitoUsername: userAddress['cognito:username'],
          orderId: response.deliveryId,
          pizza: request.body.pizza,
          address: userAddress,
          orderStatus: 'pending'
        }
      }).promise()
    })
    .then(res => {
      console.log('Order is saved!', res)
      return res
    })
    .catch(saveError => {
      console.log(`Oops, order is not saved :(`, saveError)
      throw saveError
    })
}
```

Получить userAddress для доставки пиццы.

Проверить наличие обязательных параметров в заказе.

Послать запрос на доставку в Some Like It Hot Delivery API.

Сохранить заказ в DynamoDB с помощью DocumentClient.

Вернуть ответ после сохранения.

Обработчик выглядит безупречно. Он хранится в отдельном файле, прост и понятен. Все операции он выполняет последовательно, друг за другом, но есть одна загвоздка. Как вы уже видели, автоматическое тестирование практически невозможно без вызова AWS DynamoDB. В общем и целом это хорошее решение, но мы не рассмотрели некоторые крайние случаи. Например, что, если какая-то часть службы AWS DynamoDB резко изменится и операция сохранения потерпит неудачу? Или что, если случится сбой в службе DynamoDB? Вероятность появления этих проблем очень низкая, но нам важно исключить данные риски из уравнения. Кроме того, есть еще много рисков, которые можно было бы учесть. Их можно разделить на четыре типа. Возможно, вы удивитесь, узнав, какие виды рисков охватывают эти типы. Например, вот краткий список таких рисков для примера сохранения одного заказа в DynamoDB:

- *конфигурационные риски* – сохранение производится в правильную таблицу? Обеспечивает ли роль, выбранная для функции Lambda, все необходимые права доступа к таблице в DynamoDB?
- *технические риски* – как производится парсинг входящих запросов? Правильно ли вы обрабатываете ответы, сообщающие об успехе и об ошибке?
- *риски бизнес-логики* – правильно ли структурирован заказ?
- *интеграционные риски* – правильно ли читается структура входящего запроса? Правильно ли сохраняется заказ в DynamoDB?

Вы можете протестировать каждый из этих рисков в своих интеграционных тестах, но настройка и подготовка службы перед каждым тестом – не самое оптимальное решение. Представьте, что было бы, если бы испытание автомобилей проводилось подобным образом. Каждый раз, чтобы проверить в машине один винт или даже зеркало, вам придется собирать, а затем разбирать весь автомобиль. Поэтому, чтобы упростить тестирование, бессерверную функцию следует разбить на несколько более мелких частей.

Тем, кто впервые начинает заниматься этой проблемой, трудно разбить службу на более мелкие функции. К счастью, многие разработчики уже миновали этот этап и разработали архитектурную практику, которая называется *гексагональная архитектура*, или шаблон *портов и адаптеров*.

Термин «гексагональная архитектура» кажется сложным и пугающим, но на самом деле он описывает довольно простой шаблон проектирования, в котором фрагменты кода, составляющие службу, взаимодействуют не с внешними ресурсами непосредственно, а со слоем граничных интерфейсов. Внешние службы подключаются к этим интерфейсам и преобразуют свои понятия в понятия вашего приложения. Например, обработчик `createOrder` в гексагональной архитектуре не будет напрямую получать запрос; он получит объект `OrderRequest` в конкретном прикладном формате, который содержит объекты `pizza` и `deliveryAddress`, описывающие пиццу и адрес доставки. За преобразование между форматом запроса и форматом `createOrder` будет отвечать адаптер. На рис. 11.6 наглядно показано, как будет выглядеть этот обработчик в гексагональной архитектуре.

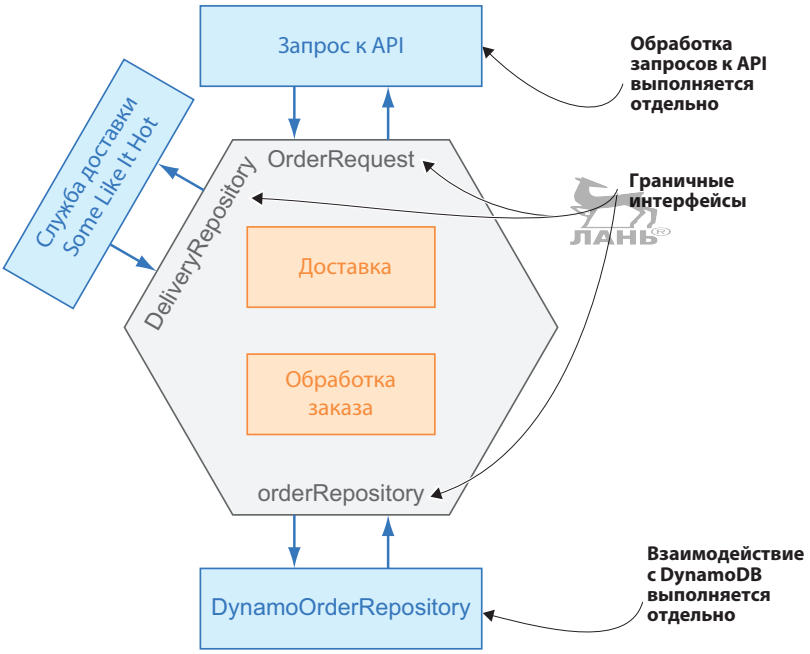


Рис. 11.6. Гексагональная архитектура

Применение этой архитектуры также означает, что функция `createOrder` не будет вызывать DynamoDB непосредственно. Вместо этого она будет обращаться к граничным интерфейсам, отвечающим вашим потребностям. Например, вы можете определить интерфейс `OrderRepository` с функцией `put` и затем определить отдельный объект `DynamoOrderRepository`, реализующий этот интерфейс и взаимодействующий с базой данных DynamoDB. То же самое можно организовать в отношении службы доставки Some Like It Hot.

Эта архитектура позволяет тестировать интеграцию с внешними API и DynamoDB, не беспокоясь о том, как в действительности ваша служба взаимодействует с DynamoDB или службой доставки. Даже если DynamoDB полностью изменит свой API или вы решите заменить DynamoDB какой-либо другой службой баз данных в AWS, основной код вашего обработчика не изменится, изменится только объект `DynamoOrderRepository`. Это упрощает проверку ответов и обработку ошибок, а также обеспечивает безопасность и согласованность кода в вашем приложении. Кроме того, такой подход помогает яснее понять, что нужно смоделировать в интеграционных тестах.

Для реализации этой архитектуры нужно разбить обработчик `createOrder` на несколько функций. Здесь мы рассмотрим только одну из них – взаимодействующую с DynamoDB. Прежде всего мы должны передать в функцию `createOrder` объект `orderRepository` как дополнительный параметр. Вместо непосредственного использования `DocumentClient` из AWS DynamoDB мы будем вызывать метод `put` объекта `orderRepository`. Необходимые изменения в обработчике `createOrder` показаны в листинге 11.14.

Листинг 11.14. Добавление в обработчик сохранения заказа поддержки объекта orderRepository

```
function createOrder(request, orderRepository) {   ← Добавлен параметр orderRepository.


  // мы удалили код инициализации AWS DynamoDB, потому что он
  // перекочевал в orderRepository   ← Код инициализации AWS DynamoDB DocumentClient был удален.
  let userAddress = request && request.body && request.body.address;
  if (!userAddress) {
    const userData = request && request.context && request.context.authorizer
                     && request.context.authorizer.claims;
    if (!userData)
      throw new Error()
    // console.log('User data', userData)
    userAddress = JSON.parse(userData.address).formatted
  }

    // остальной код не изменился
    .then(rawResponse => JSON.parse(rawResponse.body))
    .then(response => orderRepository.createOrder({   ← Вместо docClient.put теперь вызывается orderRepository.createOrder.
          cognitoUsername: userAddress['cognito:username'],
          orderId: response.deliveryId,
          pizza: request.body.pizza,
          address: userAddress,
          orderStatus: 'pending'
        })
      ).promise()
    })
    // остальной код не изменился
}
```

Листинг 11.14 наглядно показывает, как изменился обработчик `createOrder`. Теперь, если вы решите выполнить рефакторинг или заменить службу базы данных, вам не придется править код обработчика `createOrder`. Кроме того, смоделировать `orderRepository` намного проще, чем `DocumentClient`. Осталось только реализовать `orderRepository`. Его лучше определить в отдельном модуле, чтобы получить возможность использовать в других обработчиках. Код `orderRepository` показан в листинге 11.15.

Листинг 11.15. Определение объекта orderRepository

```
var AWS = require('aws-sdk')   ← Импортировать aws-sdk.
```

```
module.exports = function orderRepository() {
  var self = this
  const tableName = 'pizza-orders',
    docClient = new AWS.DynamoDB.DocumentClient({
      region: process.env.AWS_DEFAULT_REGION
    })
  self.createOrder = function (orderData) {
    return docClient.put({
      TableName: tableName,
      Item: {
        cognitoUsername: orderData.cognitoUsername,
        orderId: orderData.orderId,
        pizza: orderData.pizza,
        address: orderData.address,
        orderStatus: orderData.orderStatus
      }
    })
  }
}
```

← Настройка объекта orderRepository.

← Назначить имя таблицы pizza-orders.

Инициализация класса DocumentClient из AWS DynamoDB.

Объявление метода createOrder для объекта orderRepository.

Вызов docClient.put для сохранения orderData.

Реализация граничных интерфейсов, таких как `orderRepository`, помогает отделить специфическую логику взаимодействия с AWS DynamoDB от логики сохранения заказов. Теперь вы можете попробовать сами реализовать другие граничные интерфейсы (для обработки запросов к службе доставки и API нашего приложения).

Реализация бессерверных функций с учетом возможности их тестирования делает код проще и легче для чтения и отладки, а также устраняет потенциальные риски изменения внешних служб. Думая о тестировании перед началом разработки, можно избавиться от многих потенциальных проблем и создавать высококачественные бессерверные приложения.

Мы надеемся, что в этой главе вы почерпнули достаточно сведений, чтобы хотя бы начать тестировать ваши бессерверные функции. Теперь пришло время перейти к упражнениям!

11.9. Опробование!

Автоматизированные тесты – важнейшая часть любого приложения. Бессерверные приложения не являются исключением. Мы подготовили для вас небольшое упражнение, но призываем вас не останавливаться на этом. Пойдите дальше и напишите другие тесты, превратив тестирование ваших бессерверных приложений в обычный этап рабочего процесса.

11.9.1. Упражнение

В приложениях Node.js разработчики часто тестируют маршруты API. То же самое можно реализовать с использованием Claudia API Builder. Итак, ваша

задача – проверить правильность настройки маршрутов библиотекой Claudia API Builder. Вот несколько советов, которые могут вам пригодиться:

- используйте метод `.apiConfig` из библиотеки Claudia API Builder, чтобы получить конфигурацию API с массивом `routes`;
- есть возможность динамического создания спецификаций, перебирая содержимое массива `routes` в цикле.

Для тех, кому это упражнение покажется недостаточно сложным, мы предлагаем изменить Pizza API в соответствии с рекомендациями по организации гексагональной архитектуры, а затем протестировать оставшиеся части Pizza API. Эта дополнительная задача не обсуждается в следующем разделе, но вы можете заглянуть в исходный код примеров, чтобы увидеть наше решение.

11.9.2. Решение

Для проверки маршрутов API создайте файл `api.spec.js` в папке `specs`. Обратите внимание: этот файл не должен находиться в папке `handlers`, потому что он не тестирует обработчики.

В этом файле импортируйте главный файл `api.js` и используйте функцию `describe` из фреймворка Jasmine, чтобы добавить описание, например такое: `"API"` или `"API routes"`.

Затем определите массив объектов с маршрутами и соответствующими им методами. Маршруты должны определяться без начального слеша (`/`), потому что именно так их хранит Claudia API Builder.

Далее организуйте обход элементов массива в цикле и для каждого вызовите функцию `it` из фреймворка Jasmine. С помощью `underTest.apiConfig().routes` проверьте наличие всех маршрутов и соответствие их методов.

Содержимое файла `api.spec.js` приводится в листинге 11.16.

Листинг 11.16. Тестирование маршрутов API

```
'use strict'     ← Импортировать обработчик.

const underTest = require('../api')

describe('API', () => {
  [                ← Определить массив существующих маршрутов.
    {
      path: '',
      methods: ['GET']
    }, {
      path: 'pizzas',
      methods: ['GET']
    }, {
```

```
      path: 'orders',
      methods: ['POST']
    }, {
      path: 'orders/{id}',
      methods: ['PUT', 'DELETE']
    }, {
      path: 'delivery',
      methods: ['POST']
    }, {
      path: 'upload-url',
      methods: ['GET']
    }
  ].forEach(route => {
    it(`should setup /${route.path} route`, () => {
      expect(Object.keys(underTest.apiConfig().routes[route.path])).
        toEqual(route.methods)
    })
  })
})
```

Вызвать функцию it для каждого маршрута из массива.

Проверить присутствие маршрута и соответствие его метода.

Если теперь выполнить команду `npm test`, все тесты должны пройти успешно. Если вы пожелаете протестировать только маршруты, выполните команду `npm t filter="should setup"`.

В заключение

- Автоматизированные тесты являются важнейшей частью любого бессерверного приложения.
- Бессерверная архитектура способна положительно повлиять на традиционно медленные и дорогостоящие интеграционные тесты и тесты графического интерфейса пользователя, увеличивая скорость их выполнения за счет распараллеливания и снижения затрат на организацию тестовой инфраструктуры.
- Модульное тестирование бессерверных приложений Node.js осуществляется практически так же, как обычных приложений.
- Интеграционные тесты в бессерверных архитектурах могут подключаться к действующим службам AWS, потому что стоимость услуги бессерверных вычислений достаточно низкая.
- Взаимодействия с некоторыми сторонними службами все еще необходимо имитировать. К таким сторонним службам можно отнести платежные системы или, в случае с Pizza API, службу доставки Some Like It Hot.

- Бессерверная архитектура меняет способ тестирования программного обеспечения, потому что затраты на инфраструктуру и риски смещаются в область интеграции бессерверных компонентов.
- При проектировании бессерверных функций важно учитывать возможность и простоту последующего их тестирования, и гексагональная архитектура поможет вам в этом.

Глава 12

Получение платы за пиццу

Эта глава охватывает следующие темы:

- обработка платежей в бессерверных приложениях;
- реализация приема платы в нашем бессерверном API;
- основы безопасности данных в обработке платежей.

Введите номер вашей карты и срок ее действия. Теперь введите секретный код карты. Все знают эту последовательность. Оплата товаров или услуги кредитной картой является наиболее ценным шагом практически для любого бизнеса. До сих пор мы в основном изучали особенности разработки бессерверных приложений, которые предоставляют полезные услуги, такие как заказ пиццы и ее доставка. Но вы также должны знать, как получать платежи от клиентов тетушки Марии.

В этой главе мы посмотрим, как реализовать возможность приема онлайн-платежей в программном обеспечении для пиццерии тетушки Марии. Вы увидите, как платеж поступает в наш обработчик платежей, а затем на счет компании тетушки Марии. Затем вы узнаете, как реализовать прием платежей для тетушки Марии. А потом познакомитесь с основами безопасности приема платежей в бессерверных приложениях и узнаете, как в этом помогает соблюдение стандартов.

12.1. Платежные транзакции

По словам тетушки Марии, «все должно вращаться вокруг потребностей клиентов». Ее бизнес начал расширяться, и она получила от клиентов более ста предложений организовать возможность онлайн-оплаты в мобильном и веб-приложении. Поэтому она попросила нас помочь ей в этом.

ПРИМЕЧАНИЕ. Кому-то из вас реализация онлайн-платежей может показаться сложной и пугающей задачей, потому что вы никогда не делали этого раньше,

а также из-за высокой ответственности за возможные ошибки. Цель этой главы состоит в том, чтобы развеять эти страхи и рассказать, как происходит обработка платежей, как взаимодействовать с платежной системой и как создать бессерверную функцию, принимающую онлайн-платежи.

Перед реализацией приема платежей в нашем приложении кратко рассмотрим, как выполняются платежные транзакции.

Платеж – это финансовая операция между покупателем и продавцом. Покупатель платит продавцу деньги за товары или услуги. Если у клиента нет денег, операция невозможна. Если у покупателя есть необходимая сумма, денежные средства переводятся продавцу, после чего приобретенный товар или услуга передается или оказывается покупателю. Ход операции показан на рис. 12.1.

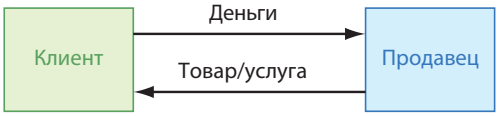


Рис. 12.1. Процесс оплаты наличными

Процесс оплаты кредитной или дебетовой картой немного отличается от процедуры оплаты наличными. Клиент вставляет карту в считывающее устройство продавца. Устройство проверяет действительность карты, считывает ее номер, отображает сумму платежа и предлагает клиенту ввести конфиденциальный пин-код, чтобы подтвердить согласие на выполнение операции. Затем устройство отправляет запрос в банк клиента на перевод средств со счета клиента на счет продавца. Если на карте достаточно свободных средств, банк резервирует сумму со счета клиента. Процесс «резервирования» также называют «взиманием» платы с клиента. Вместо немедленного списания средств со счета клиента банк сначала резервирует нужную сумму, чтобы организовать задержку на случай возникновения ошибок с обеих сторон. Ход операции по кредитной или дебетовой карте показан на рис. 12.2.

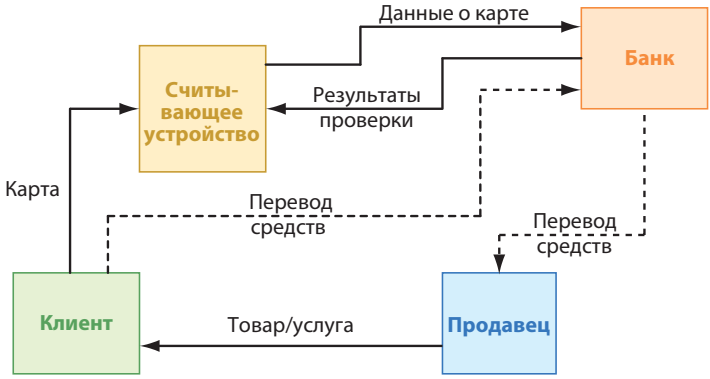


Рис. 12.2. Процесс оплаты кредитной картой с использованием считывающего устройства продавца

Онлайн-платежи отличаются от оплаты картой с использованием считывающего устройства. Во-первых, из-за невозможности использовать физическое считывающее устройство необходимо использовать платежную систему, осуществляющую обработку онлайн-платежей. Во-вторых, опять же из-за невозможности использовать физическое считывающее устройство нужно проверить карту в платежной системе. Процесс проверки необходим, потому что важно убедиться, что карта действительна и принадлежит действующему органу (например, банку). Поэтому конфиденциальные данные клиента необходимо отправить в платежную систему для проверки. Если данные верны, выполняется запрос на оплату. Третье отличие заключается в том, что теперь есть возможность проследить возможные изменения статуса платежа, так как некоторые платежи могут проверяться банком клиента и отклоняться в течение нескольких минут (рис. 12.3), но мы не будем рассматривать этот аспект ради экономии места.

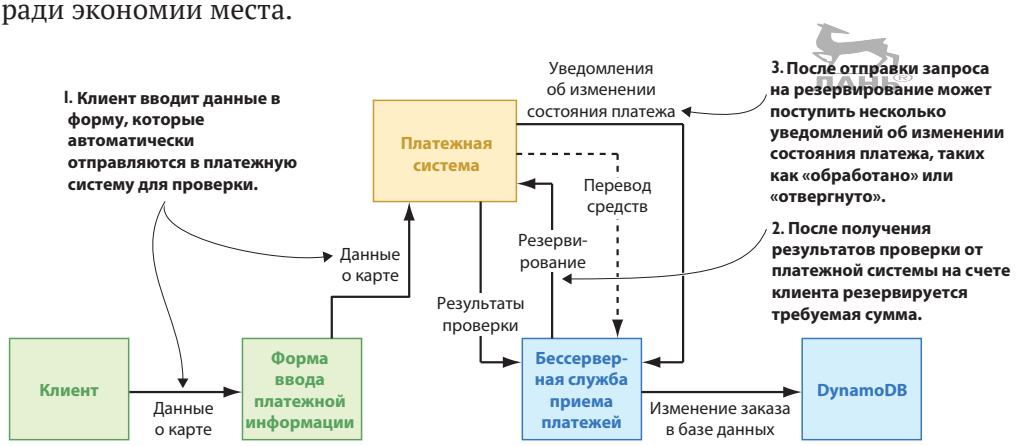


Рис. 12.3. Процедура онлайн-оплаты кредитной картой

Обработка онлайн-платежа сложнее, но с нашей точки зрения она выглядит простой. Мы должны:

1) обеспечить безопасную передачу платежной информации в систему обработки платежей;
2) зарезервировать средства на карте;
3) после резервирования обновить информацию о состоянии платежа.

Как видите, все довольно просто. А теперь, когда вы получили общее представление о процессе, давайте посмотрим, как его реализовать в пиццерии тетушки Марии.

12.1.1. Реализация онлайн-платежей

Как объяснила тетушка Мария, сейчас клиент может заплатить, только когда получит пиццу. Большинство клиентов было бы радо иметь возможность внести предоплату с помощью карты. Чтобы реализовать эту возможность, в

веб-приложение нужно добавить страницу с платежной формой, куда клиент сможет ввести необходимую информацию. После заполнения формы клиент нажимает кнопку **Оплатить**, веб-страница отправляет данные в платежную систему, и та списывает с его кредитной карты необходимую сумму.

Попробуйте представить шаги, реализующие операцию оплаты в приложении:

1) показать клиенту платежную форму с причитающейся суммой;
2) после того как клиент нажмет кнопку **Оплатить**, вызвать функцию для резервирования средств со счета клиента с помощью платежной системы;
3) после резервирования средств изменить состояние заказа в базе данных.

ПРИМЕЧАНИЕ. Имейте в виду, что мы создаем минимально работоспособный продукт, поэтому логика оплаты максимально упрощена. В действующем приложении реализация оплаты должна учитывать множество других особенностей, возможно, даже хранить историю платежей.

Этот процесс изображен на рис. 12.4.

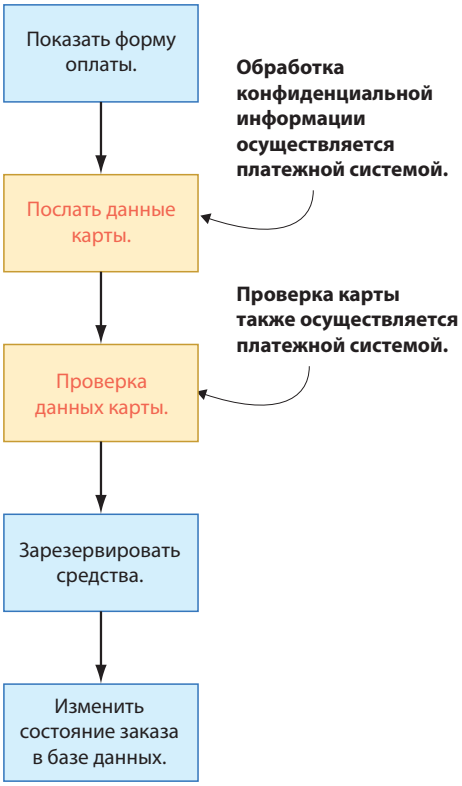


Рис. 12.4. Диаграмма, иллюстрирующая процедуру выполнения онлайн-платежа

Прежде чем приступить к реализации службы оплаты, нужно выбрать платежную систему. На выбор есть множество таких систем; в некоторых случаях ваш банк может ограничивать ваш выбор. Но наиболее известными и наиболее используемыми являются *Stripe* и *Braintree*. Мы могли бы использовать любую из них, но решили остановиться на *Stripe*: это быстрая и простая в настройке система, и она обеспечивает поддержку нескольких платформ.

> **ПРИМЕЧАНИЕ.** Тех из вас, кто предпочел бы систему Braintree, мы уверяем, что различия в реализации минимальны, они заключаются только в именах используемых библиотек и параметров. Если вы решите выбрать эту систему, просто мысленно подставляйте название Braintree вместо Strip – в остальном процесс будет тем же самым.

Теперь, определившись с выбором платежной системы, нам нужно настроить свою учетную запись в Stripe и получить ключи Stripe API (как описано в приложении C, в разделе «Настройка учетной записи Stripe и получение ключей Stripe API»). Если у вас уже есть учетная запись Stripe, продолжайте чтение; если нет, создайте ее прямо сейчас.

После настройки учетной записи Stripe войдите в нее и откройте в браузере страницу **Stripe Test** (или введите в адресную строку браузера `https://dashboard.stripe.com/test/dashboard`). Щелкните на ссылке **Accept your first payment** («Принять свой первый платеж»), чтобы открыть документацию с описанием Stripe, объясняющую, как настроить платежи по карте.

Как описывается на этой странице, мы должны выполнить два важных шага:

1) безопасно собрать информацию о платеже и получить ключ;
2) использовать ключ для резервирования средств.

> **ПРИМЕЧАНИЕ.** Описание этих шагов может отличаться, но суть остается той же: безопасно собрать платежную информацию и использовать ее для взимания платы с клиента.

Эти шаги фактически изменяют процедуру оплаты, описанную ранее. Мы не будем запрашивать платежные реквизиты клиента и отправлять их в Stripe, а просто покажем ему форму оплаты Stripe, которая отправит конфиденциальную платежную информацию прямо в Stripe. После проверки и сохранения информации о платеже Stripe вернет вам ключ – шестнадцатеричную строку, представляющую эту информацию. Срок действия ключа ограничен несколькими минутами, и его можно использовать только один раз. Мы не будем хранить одноразовый ключ, но используем его для взимания платы. Измененная процедура показана на рис. 12.5.

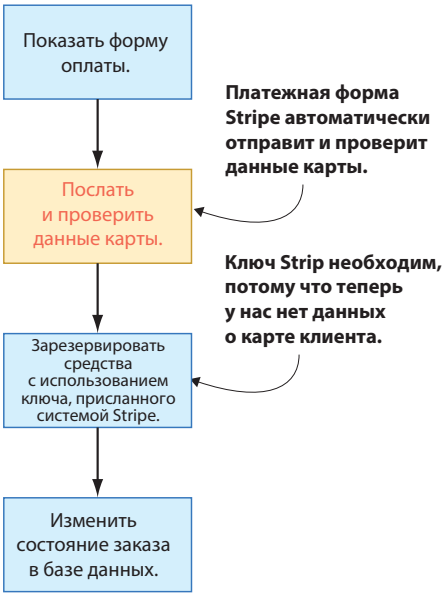


Рис. 12.5. Процедура приема онлайн-платежа с использованием платежной системы Stripe

Как видите, наша ответственность сведена к минимуму. Теперь перечислим, что мы должны реализовать.

1. *Показать платежную форму и отправить информацию в Stripe* – это означает, что нам не нужно отображать свою HTML-страницу, а значит, не нужно создавать свой HTML-документ.

   Обычно это делается в пользовательском интерфейсе веб-приложения, но мы напишем свою простенькую страницу, чтобы можно было протестировать свою службу и увидеть, как она действует. Stripe предлагает несколько способов создать свою платежную форму – с использованием:

   - Mobile SDK;
   - Checkout;
   - Stripe.js and Elements.

   Поскольку нам нужно, чтобы форма отображалась в браузере, вариант с использованием Mobile SDK не подходит. Checkout – это готовая HTML-форма, быстрая и простая в применении, а Stripe.js и Elements позволяют создать форму в своем стиле. Поскольку форма нам нужна только для тестирования, выберем вариант с использованием Checkout.
2. *Получить секретный ключ от системы платежей Stripe* – платежная форма Stripe, в нашем случае Checkout, требует настройки конечной точки веб-службы. Это означает, что кроме платежной формы нам также потребуется реализовать бессерверный API для приема конфиденциальных данных от Stripe. Поэтому мы создадим бессерверную функцию, которая будет принимать ключ.

3. *Послать запрос с ключом для взимания платы* – получив ключ, наша бессерверная платежная функция должна вызвать Stripe API, чтобы списать деньги с карты клиента, послав сумму, валюту и ключ.
4. *Изменить информацию о заказе, исходя из состояния платежа* – если списание прошло успешно, необходимо отыскать заказ в таблице DynamoDB и изменить состояние заказа (рис. 12.6).

Теперь, зная, что мы должны реализовать, приступим к работе.

Рис. 12.6. Диаграмма, иллюстрирующая процесс оплаты с использованием платежной системы Stripe

12.2. Реализация платежной службы

Для приема платежей мы должны написать платежную службу и HTML-документ. То есть мы сразу оказываемся перед выбором – с чего начать. Начнем с платежной службы, потому что нам потребуется указать URL развернутой службы в форме Stripe Checkout, находящейся в HTML-документе.

Прежде всего создайте папку `pizzeria-payments` для нового проекта и перейдите в нее в терминале.

> **ПРИМЕЧАНИЕ.** Возможно, вам интересно, почему мы создаем новый отдельный проект. При использовании бессерверного окружения AWS Lambda мы советуем оформлять службы как независимые функции или компоненты. Отсутствие тесной связи между службами дает следующие преимущества:
>
> - *высокая надежность* – если одна из служб выйдет из строя, остальные продолжат работу, тогда как в монолитном приложении сбой одной службы часто приводит к сбою всего приложения;

- *простота сопровождения* – чем меньше служба, тем уже круг решаемых ею задач, благодаря чему она проще для понимания и сопровождения;
- *возможность повторного использования* – любую функцию можно позднее немного изменить и повторно использовать в других проектах.

Внутри папки создайте NPM-файл `package.json`, выполнив команду `npm init -y`. Затем установите библиотеки Claudia API Builder и AWS SDK командой `npm install -S claudia-api-builder aws-sdk`. Поскольку мы решили использовать платежную систему Stripe, также выполните команду `npm install -S stripe`, чтобы установить Stripe SDK для Node.js – библиотеку, упрощающую отправку запросов в Stripe.

Наша платежная служба подготовит запрос к системе Stripe, добавив в него ключ, полученный ранее, и другую информацию, необходимую для взимания платы с клиента, такую как вид валюты, сумма и идентификатор заказа. Затем отправит этот запрос в Stripe с помощью Stripe SDK и получит в ответ информацию о состоянии платежа. Если оплата прошла успешно, наша служба должна выполнить запрос к базе данных DynamoDB и обновить состояние заказа в таблице `pizza-orders`. После этого служба должна вернуть сообщение об успехе.

Прежде чем приступить к фактической реализации, потратим немного времени и подумаем о применении гексагональной архитектуры, чтобы потом нам проще было протестировать нашу бессерверную платежную службу. Какие граничные объекты вам понадобятся?

Немного поразмыслив, вы придете к выводу, что нужны три граничных объекта:

- `PaymentRequest` – объект с данными, возвращаемыми в ответ на запрос о списании средств;
- `PaymentRepository` – объект с методом `createCharge` для создания запроса на списание;
- `PizzaOrderRepository` – объект с методом `updateOrderStatus` для обновления заказа в таблице `pizza-orders`.

Разместим реализацию службы в четырех файлах:

- `payment.js` – главный запускаемый файл службы, содержащий конечную точку для приема запросов `POST`, созданную с помощью Claudia API Builder;
- `create-charge.js` – файл с бизнес-логикой создания запроса на списание;
- `payment-repository.js` – файл, реализующий взаимодействия с платежной системой Stripe и определяющий единственный метод `createCharge`;
- `order-repository.js` – файл, реализующий взаимодействия с базой данных AWS DynamoDB и обновляющий состояние заказа с учетом информации о состоянии платежа.

Сначала создайте файл `payment.js`. Определите в нем объект запроса к Stripe на оплату, содержащий ключ Stripe, сумму и валюту, а также идентификатор заказа из атрибута `metadata`. Это необходимо, потому что Stripe не поддерживает отправку дополнительных параметров через свои вызовы, но дает возможность послать свойство `metadata` со строковым значением. Stripe не использует это свойство. Затем нужно вызвать функцию `createCharge`, импортированную из файла `create-charge.js`. Если все прошло успешно, отправьте сообщение об успехе. Иначе отправьте сообщение с описанием ошибки. Содержимое файла `payment.js` показано в листинге 12.1.

Листинг 12.1. Файл `payment.js` с конечной точкой для запросов POST, посылаемых платежной системой Stripe

```
'use strict'

const ApiBuilder = require('claudia-api-builder')
const api = new ApiBuilder()
const createCharge = require('./create-charge')

api.post('/create-charge', request => {

  let paymentRequest = {
    token: request.body.stripeToken,
    amount: request.body.amount,
    currency: request.body.currency,
    orderId: request.body.metadata
  }

  return createCharge(paymentRequest)
    .then(charge => {
      return { message: 'Payment Initiated!', charge: charge }
    }).catch(err => {
    return { message: 'Payment Initialization Error', error: err }
  })
})

module.exports = api
```

Импортировать экземпляр Claudia API Builder.

Импортировать файл create-charge.js с бизнес-логикой.

Определить конечную точку /create-charge.

Создать граничный объект paymentRequest.

Инициализировать атрибут orderId значением атрибута metadata.

Вызвать функцию createCharge.

В случае успеха вернуть сообщение об успехе.

В случае ошибки вернуть сообщение с ее описанием.

Экспортировать API платежной службы.

Затем создайте файл `create-charge.js` в корневой папке проекта. Он должен сначала импортировать файлы `payment-repository.js` (для работы с Stripe API) и `order-repository.js` (для обновления заказов в AWS DynamoDB), а затем определять функцию для приема запроса на оплату. Мы должны определить описание платежа, а затем вызвать функцию `paymentRepository.createCharge` с

указанными ключом, суммой и валютой, чтобы фактически выполнить платеж. После этого нужно вызвать метод `orderRepository.updateOrderStatus` с полученным идентификатором оплаченного заказа `orderId`. Содержимое файла `create-charge.js` показано в листинге 12.2.

Листинг 12.2. Файл create-charge.js с бизнес-логикой

```
'use strict'

const paymentRepository = require('./repositories/payment-repository.js')
const orderRepository = require('./repositories/order-repository.js')

module.exports = function (paymentRequest) {
  let paymentDescription = 'Pizza order payment'
  return paymentRepository.createCharge(paymentRequest.token,
        paymentRequest.amount,
      paymentRequest.currency, paymentDescription)
    .then(() => orderRepository.updateOrderStatus(paymentRequest.orderId))
}
```

Импортировать граничный объект paymentRepository.

Импортировать граничный объект orderRepository.

Определить описание платежа.

Вызвать метод createCharge.

Вызвать метод updateOrderStatus.

Теперь перейдем к методу `createCharge` в файле `payment-repository.js`. Но перед этим немного реорганизуем проект: создайте папку `repositories` в корневом каталоге проекта и перейдите в нее. Затем создайте файл `payment-repository.js`. Внутри файла определите объект с единственным методом, который создает запрос на списание средств вызовом метода `stripe.charges.create`. Методу `stripe.charges.create` необходимо передать `stripeToken` (ключ, соответствующий транзакции клиента), сумму для списания со счета клиента, валюту (сумма `amount` указывается в центах, если в параметре вида валюты `currency` передается значение `usd` или `eur`) и описание транзакции. Содержимое файла показано в листинге 12.3.

Листинг 12.3. Файл payment-repository.js, определяющий метод createCharge

```
'use strict'

const stripe = require('stripe')(process.env.STRIPE_SECRET_KEY)

module.exports = {
  createCharge: function (stripeToken, amount, currency, description){
    return stripe.charges.create({
      source: stripeToken,
      amount: amount,
      currency: currency,
      description: description
    })
```

Создать экземпляр Stripe SDK с ключом доступа из STRIPE_SECRET_KEY.

Вызвать функцию stripe.charges.create, чтобы создать запрос.

```
  }
}
```

Реализацию протокола Stripe в файле `payment-repository.js` легко заменить реализацией для Braintree или любой другой платежной системы. Оставляем это читателю в качестве упражнения.

Теперь добавим последнюю часть: файл `order-repository.js`, реализующий обновление состояния заказа в таблице `pizza-orders`.

Листинг 12.4. Файл `order-repository.js`, обновляющий состояние заказа в базе данных DynamoDB

```
'use strict'

const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient()

module.exports = {
  updateOrderStatus: function (orderId) {
    return docClient.put({
      TableName: 'pizza-orders',
      Key: {
        orderId: orderId
      },
      UpdateExpression: 'set orderStatus = :s',
      ExpressionAttributeValues: {
        ':s': 'paid'
      }
    }).promise()
  }
}
```

Импортировать класс DocumentClient.

Определение функции updateOrderStatus, принимающей идентификатор заказа.

Изменить состояние заказа с указанным идентификатором.

Записать в атрибут значение paid (оплачен).

Применив принципы гексагональной архитектуры, мы не только упростили тестирование этой платежной службы, но и облегчили возможность замены DynamoDB на Amazon Aurora или даже Amazon Relational Database Service (Amazon RDS).

Последний шаг в реализации нашей платежной службы – ее развертывание с помощью Claudia. В терминале, находясь в корневой папке проекта, выполните следующую команду, чтобы получить URL недавно созданного API:

```
claudia create --region us-east-1 --api-module payment \
  --set-env STRIPE_SECRET_KEY=<ваш-ключ-доступа-к-stripe>
```

Скопируйте и сохраните URL во временном файле, чтобы потом можно было вставить его в HTML-форму, которую еще предстоит создать. Единственное, что нам осталось, – создать HTML-документ с формой.

Заключив, что эта бессерверная платежная служба должна быть универсальной и решать единственную задачу, мы теперь не можем просто вставить в нее HTML-документ. Поэтому создайте отдельную папку для нового проекта с именем `payment-form` и внутри нее создайте HTML-документ `payment-form.html`. В элемент `body` добавьте элемент `form` с атрибутом `action`, ссылающимся на URL недавно созданной бессерверной платежной службы. Внутри элемента `form` создайте элемент `script`, загружающий форму Stripe Checkout. В элементе `script` нужно определить следующие атрибуты:

- `data-key` (начинающийся с `pk_test_`);
- `data-amount`, представляющий сумму платежа (в центах для валют USD и EUR);
- `data-name` с текстом для отображения в заголовке окна оформления платежа;
- `data-description` с текстом описания платежа;
- `data-image`, если вы решите включить URL своего логотипа или изображения для вывода в форме;
- `data-locale` с языковыми настройками (можно присвоить значение `auto`, чтобы автоматически выбрать язык, настроенный в браузере);
- `data-zip-code` с признаком необходимости запросить у пользователя почтовый индекс (логическое значение);
- `data-currency` с кодом валюты.

Содержимое файла `payment-form.html` показано в листинге 12.5.

Листинг 12.5. Файл `payment-form.html` со страницей для оплаты

```
<html>
<head>
</head>
<body>
<form action="<paste-your-function-url-here>" method="POST">
  <script
    src="https://checkout.stripe.com/checkout.js" class="stripe-button"
    data-key="<ваш-ключ-доступа-к-stripe>"
    data-amount="100"
    data-name="Demo Site"
    data-description="2 widgets"
    data-image="https://stripe.com/img/documentation/checkout/marketplace.png"
    data-locale="auto"
    data-zip-code="true"
    data-currency="usd">
  </script>
```

Элемент script для загрузки формы Stripe Checkout.

Элемент form.

← Ваш ключ доступа к Stripe.

← Сумма платежа.

← Текст для вывода в заголовке окна.

← Описание платежа.

← Выбор языка.

← Признак необходимости ввода почтового индекса.

← Код валюты.

URL-адрес логотипа или изображения для вывода в форме.

```
</form>
</body>
</html>
```

Создав файл `payment-form.html`, откройте его в браузере. Вы увидите кнопку **Pay** (Оплатить) в форме, загруженной из платежной системы Stripe. Щелкните на ней, и появится форма оплаты Stripe с суммой $1, как определено в примере (`data-amount = "100"`). Форма должна выглядеть примерно так, как показано на рис. 12.7.

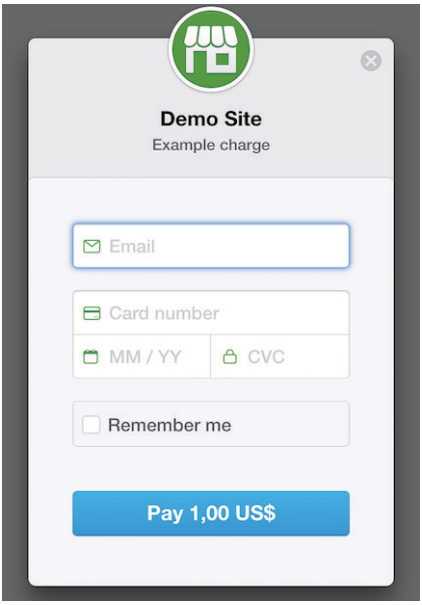


Рис. 12.7. Платежная форма

ПРИМЕЧАНИЕ. Если вы не увидите кнопку **Pay** (Оплатить), проверьте еще раз все шаги, описанные в этой главе. Если вы еще не получили ключи доступа к Stripe API, обратитесь к разделу «Настройка учетной записи Stripe и получение ключей Stripe API» в приложении C. Если у вас уже есть учетная запись, войдите в панель мониторинга Stripe и перейдите на страницу «API keys» по адресу https://dashboard.stripe.com/account/apikeys.

Для проверки платежной системы введите следующие данные:

- тестовый номер карты 4242 4242 4242 4242 (загляните на страницу https://stripe.com/docs/testing#cards);
- дату (месяц и год) истечения срока действия карты в будущем;
- любое трехзначное число в качестве секретного кода;
- любой случайный почтовый индекс;
- любой адрес электронной почты.

Затем нажмите кнопку **Pay** (Оплатить).

Вот и все. Через несколько секунд ваш платеж должен быть обработан. Заглянув в таблицу `pizza-orders`, вы увидите, что заказ получил статус `paid` (оплачено). Также обязательно загляните в панель мониторинга Stripe Dashboard (https://dashboard.stripe.com/test/dashboard), чтобы увидеть платеж. Еще можно просмотреть журналы в CloudWatch, чтобы увидеть, прошел или не прошел платеж.

Как видите, использование Claudia.js и Claudia API Builder существенно упрощает разработку и поддержку бессерверной платежной службы. Но как насчет безопасности?

12.3. Можно ли взломать нашу платежную службу?

Отсутствие полного контроля над инфраструктурой или окружением выполнения может вызывать беспокойство. Как знать, может быть, в фоновом режиме работает какая-то вредоносная служба, которая крадет данные о кредитных картах ваших клиентов? А как насчет риска взлома или мошенничества, которые могут уничтожить наш бизнес?

Мы не можем знать, что происходит на серверах поставщика услуг бессерверных вычислений. Эти опасения оправданны, потому что успешные попытки взлома провайдера могут нанести ущерб нашему предприятию. Но мы часто упускаем из виду два фактора, играющих важную роль в обеспечении безопасности:

- стандарты;
- компетентность.

12.3.1. Стандарты

Безопасность и надежность службы обработки платежей имеют большое значение не только для нас, но и для наших клиентов. Поэтому безопасность – один из главных приоритетов почти в каждой компании, по крайней мере на бумаге. Безопасность постоянно укрепляется, каждый день обнаруживаются новые проблемы. Естественно, что с течением времени многие передовые практики образовали единый стандарт, а также появился орган по стандартизации.

Органом по стандартизации является совет по стандартам безопасности в индустрии платежных карт (Payment Card Industry Security Standards Council, PCI SSC), отвечающий за определение и обеспечение безопасности платежей и методов обработки данных клиентов. Основным стандартом обеспечения безопасности платежей является стандарт безопасности данных индустрии платежных карт (Payment Card Industry Data Security Standard, PCI DSS,).

Услуги, соответствующие стандарту, называют PCI DSS-совместимыми.

Что такое PCI DSS-совместимость?

Стандарт PCI DSS устанавливает требования к организациям и продавцам по безопасному и надежному приему, хранению, обработке и передаче данных о держателе карты во время транзакций с кредитной картой, чтобы предотвратить мошенничество и утечку данных. Совместимость с PCI DSS означает, что вы безопасно обрабатываете данные держателя карты во время транзакции.

Для соответствия стандарту PCI DSS необходимо выполнить множество требований, таких как настройка брандмауэра, шифрование транзакций, ограничение физического доступа к данным, реализация внутренних политик безопасности компании и т. д. Текст стандарта можно найти по адресу https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-1.pdf.

В настоящее время почти все поставщики услуг бессерверных вычислений обеспечивают совместимость с PCI DSS, в том числе:

- AWS Lambda;
- Microsoft Azure Functions;
- Google Cloud Functions;
- IBM OpenWhisk.

Узнать больше о совместимости с PCI DSS можно на главном портале https://www.pcisecuritystandards.org.

> **ПРИМЕЧАНИЕ.** Несмотря на то что AWS Lambda совместима с PCI DSS, это не означает, что ваша служба автоматически становится совместимой с PCI DSS. Совместимость провайдера со стандартом означает лишь, что вам не нужно думать о соответствии с PCI DSS на уровне инфраструктуры. Но вам все равно нужно позаботиться о совместимости своей кодовой базы и способов обработки конфиденциальной информации при выполнении платежей.

12.3.2. Компетентность

Уязвимости в системе безопасности возможны почти всегда. Многие компании и инженеры ставят под сомнение компетентность поставщиков инфраструктуры или, в данном случае, поставщиков услуг бессерверных вычислений. Некоторые даже пытаются сами разрабатывать свои системы безопасности, несмотря на строгость, необходимую для достижения соответствия PCI.

В некоторых случаях эти усилия могут давать определенный эффект, но если у вас возникнет искушение сделать это, подумайте – действительно ли вы или ваша компания более компетентны в защите ваших данных, чем известные поставщики услуг бессерверных вычислений, такие как Amazon AWS, Microsoft Azure, Google Cloud или другие.

Ответственность за обеспечение безопасной обработки платежей огромна, и неправильная реализация может нанести большой урон вам или вашим

клиентам. Поэтому наличие компетентного поставщика услуг бессерверных вычислений, обеспечивающего совместимость с PCI, должно быть одним из главных приоритетов при разработке бессерверных приложений.

12.4. Опробование!

Как было показано в этой главе, реализация бессерверной платежной службы не требует много труда и времени. А теперь пришло время проверить ваши знания!

12.4.1. Упражнение

Ваша задача: создать новую бессерверную функцию, возвращающую список ранее созданных платежей. Реализуйте ее с применением гексагональной архитектуры. Но прежде чем приступить к выполнению упражнения, познакомьтесь с дополнительной информацией о Stripe API:

- чтобы получить все ранее выполненные платежи, используйте метод `listCharges`; дополнительную информацию о нем ищите по адресу https://stripe.com/docs/api#list_charges;
- вам потребуется настроить в своей бессерверной функции использование ключа доступа `STRIPE_SECRET_KEY` и развернуть ее командой `claudia create --set-env`;
- в отсутствие платежей служба `charge-listing` должна возвращать пустой список.

Если этой информации вам покажется недостаточно, попробуйте найти дополнительные сведения самостоятельно. Вот несколько советов, которые могут вам пригодиться:

- с помощью Claudia API Builder создайте конечную точку API с именем `GET /charges`;
- создайте объект `ChargeRepository`.

Если этого вам будет недостаточно, подсмотрите готовое решение в следующем разделе.

12.4.2. Решение

А теперь рассмотрим наше решение. Сначала обсудим общий алгоритм.

Когда запрос поступает в конечную точку `GET /charges`, мы должны проанализировать его и вызвать метод `getAllCharges` объекта `ChargesRepository`. Метод `getAllCharges` должен вызвать `stripe.charges.create` без параметров, затем проанализировать объект с ответом системы Stripe и вернуть список из атрибута `data`. Этот список должен отправляться клиенту в виде массива.

Прежде всего создайте папку проекта `charges`. Внутри этой папки выполните команду `npm init -y` и затем команду `npm install -S claudia-api-builder stripe`. Потом создайте два файла:

- `payment.js` в корневой папке проекта;
- `payment-repository.js` в папке `repositories` внутри папки проекта.

В следующих двух листингах приводится полный код службы `charge-listing`. В листинге 12.6 приводится содержимое файла `payment.js` с конечной точкой `GET /charges` для приема входящих запросов Stripe. Обработчик конечной точки должен вызвать метод `paymentRepository.getAllCharges`, чтобы получить все платежи. В случае успеха он должен вернуть список без какой-либо дополнительной обработки. В случае ошибки отправьте клиенту сообщение со свойством `error`, содержащим описание ошибки.

Листинг 12.6. Файл `payment.js`

```
'use strict'

const ApiBuilder = require('claudia-api-builder')
const api = new ApiBuilder()
const paymentRepository = require('./repositories/payment-repository')

api.get('/charges', request => {

  return paymentRepository.getAllCharges()
    .catch(err => {

      return { message: 'Charges Listing Error', error: err }
    })
})

module.exports = api
```

Импортировать файл paymentrepository.js.

Определение конечной точки GET /charges.

Импортировать экземпляр Claudia API Builder.

Вызвать метод paymentRepository.getAllCharges.

В случае неудачи вернуть сообщение об ошибке.

Экспортировать службу charge-listing.

В листинге 12.7 показано содержимое файла `payment-repository.js`, отвечающего за получение всех платежей, произведенных пользователем. Он реализует метод `getAllCharges`, который вызывает метод `stripe.charges.list` без параметров.

Листинг 12.7. Файл `payment-repository.js`

```
'use strict'

const stripe = require('stripe')(process.env.STRIPE_SECRET_KEY)

module.exports = {
  getAllCharges: function (){
    return stripe.charges.list()
      .then(response => response.data)
```

Создать экземпляр Stripe SDK с ключом доступа из STRIPE_SECRET_KEY.

Вызвать метод stripe.charges.list.

Вернуть response.data со списком платежей.

```
  }
}
```

В заключение

- Знать, как осуществлять платежи, необходимо для разработки любых приложений, независимо от того, являются оно бессерверными или нет.
- Обработку платежей лучше реализовать в виде независимой бессерверной службы, чтобы она работала стабильно и не зависела от других служб в вашем приложении.
- Интегрировать платежную систему Strip в платежную службу на основе AWS Lambda очень просто.
- Тщательно проработанная и независимая платежная служба может впоследствии использоваться другими вашими продуктами или услугами.
- Отсутствие контроля над инфраструктурой не является оправданием для снижения уровня безопасности.
- Хорошим показателем безопасности вашей платежной службы может быть совместимость вашего поставщика услуг бессерверных вычислений со стандартом PCI DSS.
- Совместимость с PCI – обязательное требование при выборе поставщика услуг бессерверных вычислений, потому что она обеспечивает необходимый уровень безопасности.

Глава 13

Миграция существующих приложений Express.js в окружение AWS Lambda

Эта глава охватывает следующие темы:

- запуск приложений Express.js в бессерверной экосистеме AWS Lambda;
- обслуживание статического контента приложений Express.js;
- подключение бессерверных приложений Express.js к MongoDB;
- ограничения и риски использования приложений Express.js в бессерверной экосистеме.

Express.js – наиболее важный и широко используемый фреймворк в экосистеме Node.js. И тому есть веские причины: Express.js прост в использовании и имеет обширную экосистему вспомогательного программного обеспечения, способствующего в создании серверных API и веб-приложений. Но для использования Express.js по-прежнему требуется сервер, на котором будет размещаться приложение, а это значит, что мы вернулись к проблемам, которые пытается решить эта книга с помощью бессерверных технологий. Есть ли способ сохранить существующее приложение Express.js и при этом пользоваться всеми преимуществами бессерверных вычислений?

Веб-фреймворк Express.js по сути является HTTP-сервером. Бессерверным приложениям не нужны HTTP-серверы, потому что HTTP-запросы обрабатываются шлюзом API Gateway. Но, к счастью, AWS Lambda предлагает возможность выполнять существующие приложения Express.js с небольшими изменениями. В этой главе рассказывается, как это сделать, а также описываются некоторые наиболее существенные ограничения для выполнения приложений Express.js в бессерверном окружении.

13.1. Приложение для таксомоторной компании дядюшки Роберто

На последнем семейном совете тетушка Мария похвасталась своим новым онлайн-бизнесом. По ее словам, она получила то, о чем и мечтать не смела, и самое замечательное, что новое приложение просто работает – оно отлично справляется с любым количеством заказов.

Ее брат, дядюшка Роберто, отметил, что ей здорово повезло, потому что сам давно страдает от проблем в приложении для его таксомоторной компании. Пока клиентов немного, оно работает, но когда заявок поступает больше, чем обычно, например в дождь, приложение падает. К сожалению, его IT-команда не может справиться с проблемами, и он теряет клиентов и деньги.

Роберто спросил, как нам удалось создать такое чудесное приложение для тетушки Марии и можно ли что-то сделать для увеличения надежности его приложения. Мы объяснили ему, что многое зависит от технологии, которую использует приложение.

Через несколько дней мы получили сообщение, что приложение в таксомоторной компании использует Express.js и MongoDB. Оно действует на небольшом виртуальном частном сервере, который обслуживает RESTful API для мобильного приложения, а его административная часть реализована в виде набора HTML-страниц. То есть это самое обычное приложение Express.js. Мы согласились провести исследования и через несколько дней сообщить дядюшке Роберто, сможем ли чем-нибудь помочь ему.

13.2. Запуск приложения Express.js в AWS Lambda

Прежде чем приступать к исследованиям, создадим простое приложение Express.js – мы используем его для проверки работы Express.js в AWS Lambda. Для этого создайте новую папку проекта `simple-express-app`. Затем инициализируйте в ней новый проект NPM и установите Express.js как зависимость, выполнив команду `npm i express -S`.

Первым делом создадим один файл с маршрутом Express.js и попробуем запустить его под управлением AWS Lambda. Итак, создайте файл `app.js` в папке проекта `simple-express-app`.

Внутри файла импортируйте модуль `express` и с его помощью создайте новое приложение Express. Затем добавьте маршрут `GET /`, возвращающий текст «Hello World». Наконец, определите порт приложения и запустите сервер вызовом функции `server.listen`.

На данный момент содержимое файла `app.js` должно выглядеть, как показано в листинге 13.1.

Листинг 13.1. Приложение Express.js

```
'use strict'

const express = require('express')
const app = express()

app.get('/', (req, res) => res.send('Hello World'))

const port = process.env.PORT || 3000
app.listen(port, () => console.log(`App listening on port ${port}`))
```

Создать маршрут GET, возвращающий текст «Hello World».

Создать приложение Express.js.

Настроить порт, указанный в переменой окружения PORT, или 3000, если эта переменная не определена.

Запустить приложение на указанном порту.

Теперь запустите это простое приложение Express.js командой

```
node app.js
```

Эта команда запустит локальный сервер, который будет прослушивать порт 3000, если переменная окружения `PORT` не определена. Открыв страницу `http://localhost:3000` в веб-браузере, вы должны увидеть текст «Hello World».

Самый простой способ запустить существующее приложение Express.js в окружении AWS Lambda – воспользоваться модулем `aws-serverless-express` из Node.js. Для использования этого модуля не требуется вносить существенных изменений в приложение Express.js.

Итак, чтобы подготовить приложение к выполнению под управлением AWS Lambda и API Gateway, откройте файл `app.js` и замените вызов функции `app.listen` простой инструкцией импортирования, как показано в листинге 13.2. Это позволит обертке Express.js в AWS Lambda загрузить ваше приложение.

Листинг 13.2. Приложение Express.js, подготовленное для выполнения в окружении AWS Lambda

```
'use strict'

const express = require('express')
const app = express()

app.get('/', (req, res) => res.send('Hello World'))

module.exports = app
```

Вместо вызова функции app.listen нужно экспортировать экземпляр приложения.

Но теперь это приложение не получится запустить на локальном компьютере командой `node app.js`.

Чтобы исправить эту проблему, создайте в папке проекта файл `app.local.js`. Этот файл должен импортировать приложение Express.js из файла `app.js` и вызывать функцию `app.listen` для запуска локального сервера, прослушивающего указанный порт.

Содержимое файла `app.local.js` приводится в листинге 13.3.

Листинг 13.3. Выполнение приложения Express.js, подготовленного для работы в среде AWS Lambda, на локальном компьютере

```
'use strict'

const app = require('./app')
const port = process.env.PORT || 3000

app.listen(port, () => console.log(`App listening on port ${port}`))
```

Импортировать приложение из файла app.js.

Настроить порт и запустить приложение.

Чтобы убедиться, что локальная версия приложения Express.js по-прежнему работает в соответствии с нашими ожиданиями, выполните команду

```
node app.local.js
```

После этого, открыв страницу `http://localhost:3000` в веб-браузере, вы должны увидеть все тот же текст «Hello World».

Теперь, убедившись, что локальная версия работает, сгенерируем обертку для приложения Express.js. Для этого достаточно выполнить команду `claudia generate-serverless-express-proxy`. Эта команда требует передать ей параметр `--express-module` и указать путь к главному файлу без расширения `.js`. Например, для файла `app.js` команда должна выглядеть так:

```
claudia generate-serverless-express-proxy --express-module app
```

> **ПРИМЕЧАНИЕ.** Для опробования примеров в этой главе у вас должна быть установлена версия Claudia не ниже 3.3.1.

Эта команда сгенерирует файл `lambda.js` и установит модуль `aws-serverless-express` как зависимость времени разработки.

Файл, созданный командой, – это обертка, которая запускает приложение Express.js в окружении AWS Lambda. Она использует функцию `awsServerlessExpress.createServer`, чтобы запустить приложение Express.js внутри функции Lambda. Затем вызывает функцию `awsServerlessExpress.proxy`, чтобы преобразовать запрос API Gateway в HTTP-запрос и передать его в приложение Express.js, а потом преобразовать и вернуть ответ в API Gateway.

Содержимое файла показано в листинге 13.4.

Листинг 13.4. Обертка AWS Lambda для приложений Express.js

```
'use strict'
const awsServerlessExpress = require('aws-serverless-express')
const app = require('./app')
```

Импортировать модуль aws-serverless-express.

Импортировать приложение из файла app.js.

```
const binaryMimeTypes = [
    'application/octet-stream',
    'font/eot',
    'font/opentype',
    'font/otf',
    'image/jpeg',
    'image/png',
    'image/svg+xml'
]
const server = awsServerlessExpress.createServer(app, null, binaryMimeTypes)
exports.handler = (event, context) => awsServerlessExpress.proxy(server,
    event, context)
```

Список разрешенных MIME-типов запросов, которые будут преобразовываться и передаваться в приложение Express.js.

Создать HTTP-сервер.

Экспортировать функцию-обработчик, пересылающую запросы в приложение Express.js.

Следующий шаг: развертывание API в AWS Lambda и API Gateway. Сделать это можно с помощью команды `claudia create`, но здесь есть одно важное отличие от команд, которые использовались для развертывания API в предыдущих главах: мы должны использовать параметр `--handler` вместо `--api-module`, а также добавить параметр `--deploy-proxy-api`. Эти параметры обеспечат интеграцию с оберткой и непосредственную передачу всех запросов к API Gateway в нашу функцию Lambda.

Чтобы развернуть приложение Express.js, выполните следующую команду:

```
claudia create --handler lambda.handler --deploy-proxy-api --region eu-central-1
```

В случае успеха эта команда должна вывести отчет, как показано в листинге 13.5.

Листинг 13.5. Результат развертывания

```
{
  "lambda": {
    "role": "simple-express-app-executor",
    "name": "simple-express-app",
    "region": "eu-central-1"
  },
  "api": {
    "id": "8qc6lgqcs5",
    "url": "https://8qc6lgqcs5.execute-api.eu-central-1.amazonaws.com/latest"
  }
}
```

Адрес URL обертки.

Как видите, обычный ответ команды развертывания дополнен параметром `url`. И если открыть этот URL в браузере (в данном случае https://8qc6lgqcs5.execute-api.eu-central-1.amazonaws.com/latest), то вы должны увидеть текст «Hello World».

13.2.1. Интеграция с оберткой

Как рассказывалось в главе 2, API Gateway можно использовать в двух режимах:

- с моделями и шаблонами запросов и ответов;
- в интеграции с оберткой.

Первый режим больше подходит для типизированных языков, таких как Java и .Net, но поскольку библиотека Claudia ориентирована исключительно на JavaScript, она всегда использует второй режим. В этом случае API Gateway всегда передает запросы непосредственно в функцию AWS Lambda, которая сама должна обеспечить маршрутизацию и обработку запросов.

Когда выполняется развертывание обертки для приложения Express.js, библиотека Claudia выполняет следующие операции:

- создает ресурс обертки с переменной пути `{proxy+}`;
- настраивает метод `ANY` для ресурса обертки;
- интегрирует ресурс и метод с помощью функции Lambda.

Узнать больше об интеграции обертки можно по адресу https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-set-up-simple-proxy.html.

13.2.2. Как работает serverless-express

Фактически приложение Express.js – это небольшой HTTP-сервер внутри функции AWS Lambda, а модуль `serverless-express` действует в роли прокси между API Gateway и этим HTTP-сервером.

Когда пользователь посылает HTTP-запрос, API Gateway передает его в функцию AWS Lambda. Внутри функции модуль `serverless-express` запускает сервер Express.js и кеширует его для обработки повторных запросов, а затем преобразует событие API Gateway в HTTP-запрос, который передает приложению Express.js.

> **Не является ли запуск HTTP-сервера внутри AWS Lambda антишаблоном?**
>
> Бессерверные приложения все еще считаются новинкой, поэтому шаблоны и методы программирования пока сформировались не полностью. Они меняются с появлением каждой новой особенности. Запуск HTTP-сервера внутри AWS Lambda выглядит как антишаблон, и этот подход имеет несколько недостатков, например увеличение времени выполнения и размера функций. Но он имеет также множество положительных сторон, таких как возможность использования существующей кодовой базы и избежание привязки к поставщику. Еще одна причина, по которой этот подход нельзя назвать антишаблоном, заключается в том, что AWS Lambda со средой выполнения GoLang использует аналогичный подход для запуска функции.

После этого приложение Express.js выполняет обычную для него последовательность действий – маршрутизатор выбирает обработчик и применяет все промежуточные функции. Когда приложение Express.js посылает ответ, модуль `serverless-express` преобразует его в формат, понятный API Gateway, и затем он возвращается пользователю. Этот процесс изображен на рис. 13.1.

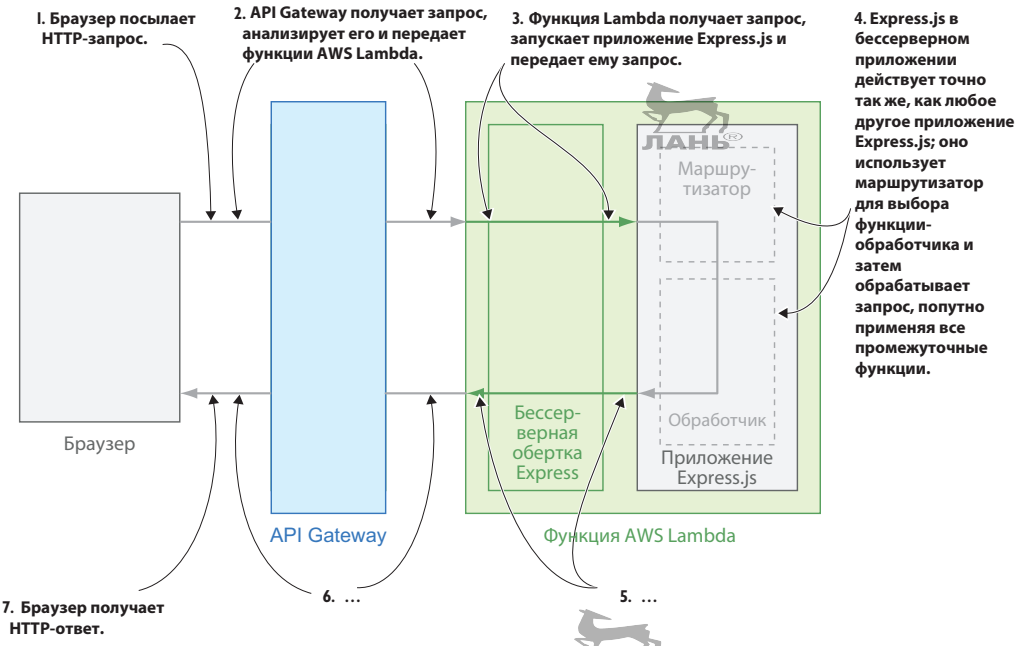


Рис. 13.1. Последовательность действий, выполняемых в бессерверном приложении Express.js

13.3. Обслуживание статического контента

Еще один сценарий, который мы должны проверить, – это обслуживание статического контента из приложения Express.js, потому что именно так организована работа административного раздела в приложении дядюшки Роберто.

Для проверки нам понадобится простая статическая HTML-страница. Подойдет любая страница, включающая хотя бы одно изображение и простой файл CSS, – это позволит нам проверить работу с файлами разных типов.

Прежде всего создадим новую папку `static` внутри проекта Express.js. Затем добавим в эту папку файл `index.html`, который загружает `style.css`, отображает некоторый заголовок и изображение, например логотип Claudia (`claudiajs.png`). Оба файла, `style.css` и `claudiajs.png`, будут загружаться из папки `static`.

В листинге 13.6 показано содержимое файла `index.html`.

Листинг 13.6. Файл `index.html`

```
<!doctype html>
<html>
  <head>
    <title>Static site</title>
    <link rel="stylesheet" href="style.css">  ← Загрузить файл CSS.
  </head>
  <body>
    <h1>Hello from serverless Express.js app</h1>  ← Это заголовок.
    <img src="claudiajs.png" alt="Claudia.js logo" />  ← Показать на странице изображение claudiajs.png.
  </body>
</html>
```

Теперь добавьте в папку `static` логотип Claudia (его можно найти в примерах исходного кода к книге или на веб-сайте проекта Claudia) и файл `style.css`.

Нет нужды добавлять в файл CSS что-то необычное, но при желании вы можете проявить творческий подход. Однако в книге мы ограничимся определением простого стиля, оформляющего заголовок синим цветом, добавляющего тень и располагающего текст по центру страницы. Содержимое нашего файла CSS показано в листинге 13.7.

Листинг 13.7. Файл `style.css`

```
body {
  margin: 0;
}

h1 {
  color: #71c8e7;
  font-family: sans-serif;
  text-align: center;
  text-shadow: 1px 2px 0px #00a3da;
}

img {
  display: block;
  margin: 40px auto;
  width: 80%;
  max-width: 400px;
}
```

Далее добавим в файл `app.js` обслуживание статического содержимого из папки `static`. Для этого следует использовать промежуточную функцию `express.static`, как показано в листинге 13.8.

Листинг 13.8. Обслуживание статического контента из приложения Express.js

```
'use strict'

const express = require('express')
const app = express()

app.use('/static', express.static('static'))

app.get('/', (req, res) => res.send('Hello World'))

module.exports = app
```

Извлекать статический контент из папки static.

Теперь можно проверить работу локальной версии приложения Express.js, выполнив команду `node app.local.js` и открыв в браузере страницу `http://localhost:3000/static`.

Если локальная версия работает нормально, обновите приложение командой `claudia update`.

Дождитесь завершения команды и откройте страницу https://8qc6lgqcs5.execute-api.eu-central-1.amazonaws.com/latest/static/. Вы должны увидеть свою статическую HTML-страницу с логотипом Claudia, как показано на рис. 13.2.

> **ПРИМЕЧАНИЕ.** Завершающий слеш (/) в этом URL является обязательным. Если опустить его (ввести адрес https://8qc6lgqcs5.execute-api.eu-central-1.amazonaws.com/latest/static), попытка открыть страницу завершится неудачей.

Рис. 13.2. Статическая HTML-страница из приложения Express.js в AWS Lambda

13.4. Подключение к MongoDB

До сих пор все шло как по маслу – нам понадобилось внести лишь весьма незначительные изменения. Но получится ли у нас так же легко и просто связать приложение с базой данных MongoDB?

Функцию AWS Lambda можно связать с любой базой данных, но если база данных не является бессерверной, могут возникнуть проблемы, когда при масштабировании функции будет выполнена попытка установить с базой данных слишком много соединений (не забывайте, что база данных не масштабируется автоматически).

Чтобы обеспечить нормальную работу базы данных с функцией AWS Lambda, можно:

- выбрать базу данных, которая легко и быстро масштабируется;
- ограничить масштабируемость функции AWS Lambda до пределов, не превышающих возможностей базы данных;
- использовать управляемую базу данных.

Первый вариант требует большого опыта практической работы и хорошего знания особенностей разных баз данных, однако и то, и другое находится за рамками этой книги.

Второй вариант имеет право на жизнь, но ограничение масштабируемости не позволит приложению справляться с наплывом пользователей в пиковые периоды. Больше узнать об управлении масштабируемостью в окружении AWS можно по адресу https://docs.aws.amazon.com/lambda/latest/dg/concurrent-executions.html.

Последний вариант самый простой и, пожалуй, самый лучший, поэтому остановим свой выбор на нем. Для поддержки базы данных MongoDB, которую использует приложение дядюшки Роберто, можно использовать проект MongoDB Atlas, предлагаемый MongoDB, Inc. Он размещает базу данных на одном из нескольких облачных ресурсов, включая AWS. Более подробную информацию о MongoDB Atlas можно найти по адресу https://www.mongodb.com/cloud/atlas.

13.4.1. Использование управляемой базы данных MongoDB с бессерверным приложением Express.js

Прежде всего вам нужно зарегистрировать учетную запись в MongoDB Atlas и создать базу данных, как описывается в приложении C.

Также нужно настроить подключение к базе данных в файле `app.js`. Для этого установите NPM-модули `mongodb` и `body-parser` как зависимости в своем проекте Express.js. Первый позволит устанавливать соединения с базой данных MongoDB, а второй даст приложению Express.js возможность анализировать запросы `POST`.

Установив модули, создадим соединение с базой данных. Функции AWS Lambda на самом деле способны сохранять состояние между вызовами, потому

что если функция будет повторно вызвана в течение следующих нескольких минут, для ее выполнения может использоваться тот же самый контейнер. Это означает, что все, что находится за пределами функции-обработчика, будет сохранено, и вы сможете повторно использовать то же соединение MongoDB.

Например, сохранив соединение с базой данных за пределами функции-обработчика, вы сможете проверить его активность, вызвав:

```
cachedDb.serverConfig.isConnected()
```

Если соединение все еще активно, вы должны использовать его. Если соединение неактивно, можно создать новое, вызвав функцию `MongoClient.connect` и сохранив установленное соединение в переменной перед дальнейшим использованием. Затем нужно активировать модуль `body-parser`.

Старайтесь повторно использовать активное соединение, потому что любая база данных имеет ограничение на максимальное число активных соединений. Например, бесплатный экземпляр MongoDB Atlas поддерживает не более 100 активных соединений. Это означает, что одновременно к нему может обратиться не более 100 функций Lambda. Повторное использование активного соединения поможет вам не превысить это ограничение и одновременно уменьшить задержки, потому что для установки нового соединения с базой данных требуется некоторое время.

Процесс подключения функции Lambda к базе данных MongoDB изображен на рис. 13.3.

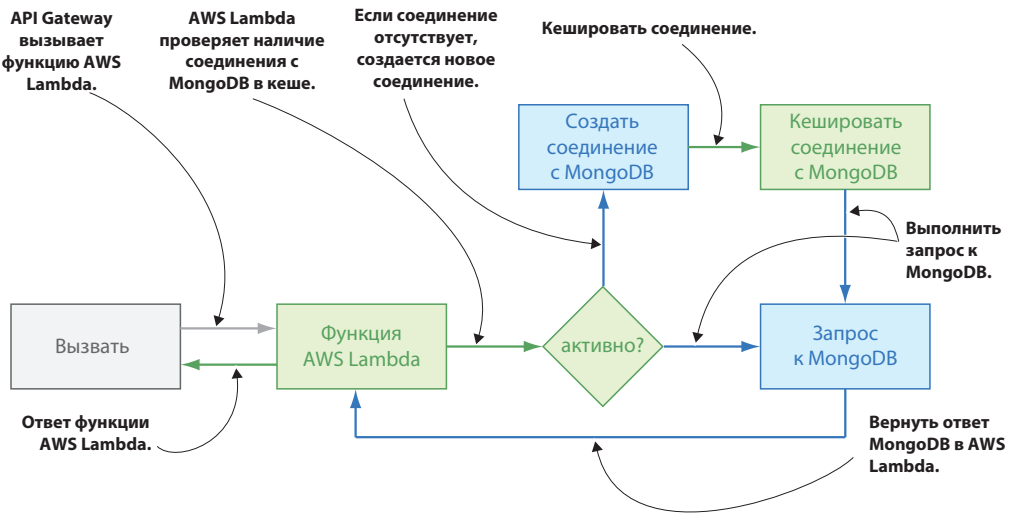


Рис. 13.3. Процесс подключения к базе данных MongoDB и кеширования активного соединения

На данный момент начало файла `app.js` должно выглядеть, как в листинге 13.9.

Листинг 13.9. Начало файла `app.js`

```
const express = require('express')
const app = express()
const { MongoClient } = require('mongodb')   ← Импортировать модуль mongodb.
const bodyParser = require('body-parser')    ← Импортировать модуль body-parser.

let cachedDb = null   ← Кеш для соединения с базой данных.

function connectToDatabase(uri) {   ← Соединиться с базой данных.
  if (cachedDb && cachedDb.serverConfig.isConnected()) {   ← Проверить наличие соединения с базой данных в кеше, и если оно имеется и активно – вернуть его.
    console.log('=> using cached database instance')
    return Promise.resolve(cachedDb)
  }

  return MongoClient.connect(uri)   ← Иначе создать новое соединение и сохранить в кеше.
    .then(client => {
      cachedDb = client.db('taxi')
      console.log('Not cached')
      return cachedDb
    })
}

app.use(bodyParser.json())   ← Активировать модуль body-parser.
```

Теперь проверим соединение функции Express.js с базой данных MongoDB. Самый простой способ сделать это – записать что-нибудь в коллекцию в базе данных, а затем прочитать содержимое коллекции, чтобы убедиться, что данные действительно были записаны. С этой целью добавим два маршрута: один будет выполнять запись в базу данных MongoDB, а другой – чтение. Например:

- маршрут `POST /orders`, который будет добавлять новый заказ;
- маршрут `GET /orders`, который будет возвращать список всех имеющихся заказов.

Вот как будет выглядеть процесс выполнения этих двух новых маршрутов.

1. Запрос `POST /orders` достигнет API Gateway и будет передан в функцию AWS Lambda.
2. Функция Lambda запустит приложение Express.js.
3. Затем функция Lambda преобразует запрос API Gateway в HTTP-запрос и отправит его в приложение Express.js.
4. Приложение Express.js проверит наличие соединения с MongoDB и при необходимости создаст новое соединение.

5. Обработчик в приложении Express.js сохранит заказ в MongoDB и вернет ответ.
6. Функция Lambda преобразует ответ приложения Express.js в формат, который ожидает получить API Gateway.
7. API Gateway вернет ответ пользователю.
8. Получив ответ, пользователь тут же отправит запрос `GET /orders`, и API Gateway передаст его в функцию Lambda.
9. Функция Lambda преобразует запрос API Gateway в HTTP-запрос и отправит его в уже существующий экземпляр приложения Express.js.
10. Приложение Express.js проверит наличие соединения с MongoDB и, поскольку оно установлено, использует его для получения всех заказов из базы данных.
11. Функция Lambda преобразует ответ приложения Express.js и передаст его в API Gateway.
12. Пользователь получит ответ от API Gateway со списком всех заказов.

ПРИМЕЧАНИЕ. Соединение с MongoDB и экземпляр приложения Express.js сохраняются в кеше. Это происходит один раз, в момент холодного запуска функции.

Описанный процесс изображен на рис. 13.4.

Для подключения обработчиков новых маршрутов к базе данных MongoDB используем функцию `connectToDatabase`, созданную выше. Передадим ей строку соединения с MongoDB, хранящуюся в переменной окружения.

Далее, чтобы получить все записи из коллекции `orders` и преобразовать результат в простой массив JavaScript, маршрут `GET /orders` должен вызвать функцию `db.collection('orders').find().toArray()`. Эта команда вернет объект `Promise`, и когда он завершит выполнение, мы сможем вызвать функцию `res.send` из фреймворка Express.js, чтобы послать результат или сообщение об ошибке.

Маршрут `POST /orders` отличается от `GET /orders` только тем, что должен записать новый элемент в базу данных, а не прочитать элементы из нее. Чтобы выполнить запись, используем функцию `db.collection('orders').insertOne` и передадим ей объект JSON, содержащий только адрес доставки.

Фактическая реализация маршрутов показана в листинге 13.10.

Листинг 13.10. Маршруты для чтения и записи новых заказов на такси

```
app.get('/orders', (req, res) => {                    ← Добавить маршрут GET.
  connectToDatabase(process.env.MONGODB_CONNECTION_STRING)
    .then((db) => {                                   ← Получить соединение с базой данных.
      return db.collection('orders').find().toArray()
    })                                                ← Получить все заказы и преобразовать их в массив.
    .then(result => {                                 ← В случае успеха вернуть результат.
      return res.send(result)
```

```
    })
    .catch(err => res.send(err).status(400))
})
```

Если что-то пошло не так, вернуть ошибку с кодом 400.

```
app.post('/orders', (req, res) => {
  connectToDatabase(process.env.MONGODB_CONNECTION_STRING)
    .then((db) => {
      return db.collection('orders').insertOne({
        address: req.body.address
      })
    })
    .then(result => res.send(result).status(201))
    .catch(err => res.send(err).status(400))
})
```

Добавить маршрут POST.

Получить соединение с базой данных.

Записать заказ в базу данных.

В случае успеха вернуть результат.

Если что-то пошло не так, вернуть ошибку с кодом 400.

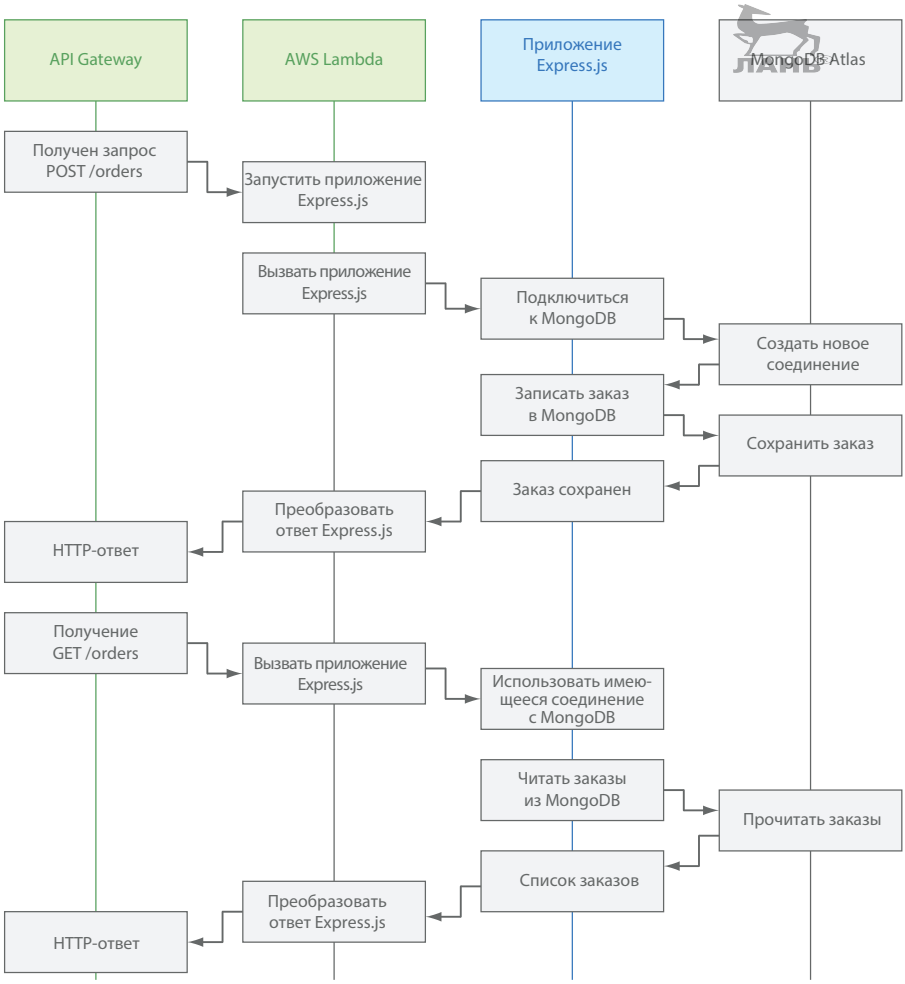


Рис. 13.4. Процесс записи и чтения заказов из MongoDB

Добавив обработчики, проверим подключение к MongoDB на локальном компьютере командой `node app.local.js`, но не забудьте перед этим инициализировать переменную окружения `MONGODB_CONNECTION_STRING`. Например:

```
MONGODB_CONNECTION_STRING=mongodb://localhost:27017 node app.local.js
```

Если проверка на локальном компьютере прошла успешно, выполните команду `claudia update` с параметром `--set-env` или `--set-env-from-json`, в котором передайте переменную `MONGODB_CONNECTION_STRING`. Например, вот как могла бы выглядеть такая команда:

```
claudia update --set-env MONGODB_CONNECTION_STRING=mongodb://<пользователь>:
<пароль>@robertostaxicompany-shard-00-00-rs1m4.mongodb.net:27017,
robertostaxicompany-shard-00-01-rs1m4.mongodb.net:27017,robertostaxicompany
-shard-00-02-rs1m4.mongodb.net:27017/taxi?ssl=true&replicaSet=
RobertosTaxiCompany-shard-0&authSource=admin
```

> **ПРИМЕЧАНИЕ.** В примерах исходного кода для этой книги мы использовали другой способ, чтобы показать доступные возможности: строка соединения с MongoDB определена в файле `env.json` и передается функции AWS Lambda командой `claudia update --set-env-from-json env.json`.
>
> У нас определена только одна переменная окружения, поэтому подойдут оба варианта. Но на будущее, если вам понадобится несколько переменных, мы советуем создавать файл JSON с их определениями, потому что такой подход уменьшает длину команды и вероятность ошибиться при ее вводе.

После развертывания приложения можете попробовать добавить заказ, послав запрос `POST` на адрес https://8qc6lgqcs5.execute-api.eu-central-1.amazonaws.com/latest/orders. Аналогично можно посетить тот же адрес https://8qc6lgqcs5.execute-api.eu-central-1.amazonaws.com/latest/orders и увидеть в браузере список заказов.

13.5. Ограничения бессерверных приложений Express.js

Теперь, проверив все важные стороны работы приложения, можно сообщить дядюшке Роберто, что его приложение Express.js вполне можно перенести на платформу AWS Lambda. Уверены, он будет рад услышать это, а мы получим в награду множество бесплатных поездок на такси.

Но прежде давайте поговорим о некоторых наиболее важных ограничениях, стоящих на пути приложений Express.js к бессерверному окружению.

Первое и, пожалуй, самое очевидное ограничение – бессерверные приложения Express.js не смогут использовать веб-сокеты. Если в приложении дядюшки Роберто используются веб-сокеты для общения с клиентами в режиме реального времени, оно будет работать не так, как ожидалось. Некоторую

ограниченную поддержку веб-сокетов в AWS Lambda может обеспечить AWS IoT MQTT через протокол WebSockets. Более подробную информацию о протоколе MQTT ищите по адресу https://docs.aws.amazon.com/iot/latest/developerguide/protocols.html#mqtt. Пример проекта на основе Claudia вы найдете по адресу https://github.com/claudiajs/serverless-chat.

Другое ограничение связано с выгрузкой файлов. Если приложение попытается выгрузить файл в любую папку, кроме `/tmp`, оно потерпит неудачу, потому что все дисковое пространство в AWS Lambda, кроме этой папки, доступно только для чтения. Но даже если вы выгрузите файл в папку `/tmp`, он просуществует в ней очень недолгое время. Чтобы организовать выгрузку файлов, используйте AWS S3.

Следующее ограничение – аутентификация. Аутентификацию в бессерверных приложениях Express.js можно реализовать, например, используя библиотеку Passport.js, но вам придется обеспечить сохранение сеансов вне локальной файловой системы. Или, если вы пользуетесь библиотеками Node.js, вам придется упаковать их в статические файлы с использованием машины EC2, выполняющейся под управлением Amazon Linux. Больше информации о таких библиотеках вы найдете по адресу https://nodejs.org/api/addons.html.

Кроме того, API Gateway накладывает свои ограничения для приложений Express.js. Например, Node.js и Express.js позволяют послать запрос `GET` с телом; API Gateway не поддерживает такой возможности.

Помимо всего перечисленного, существуют определенные ограничения времени выполнения, например максимальное время ожидания API Gateway составляет 30 секунд, а максимальное время выполнения функции AWS Lambda ограничено 5 минутами. Если приложению Express.js требуется больше 30 секунд для ответа, запрос не будет выполнен. Кроме того, если приложение Express.js должно ответить на HTTP-запрос и продолжить выполнение, из этого тоже ничего не выйдет, потому что выполнение функции AWS Lambda будет остановлено сразу после отправки HTTP-ответа. Это поведение зависит от свойства `callbackWaitsForEmptyEventLoop` контекста Lambda; по умолчанию оно имеет значение `true`. Это означает, что обратный вызов будет ждать, пока цикл событий не исчерпает все события, прежде чем остановить процесс и вернуть результаты вызывающей стороне. В это свойство можно записать значение `false`, чтобы запросить AWS Lambda приостановить процесс сразу после обратного вызова, даже если в буфере еще есть события.

Если ни одно из этих ограничений не нарушается, приложение дядюшки Роберто вполне сможет работать в окружении AWS Lambda.

13.6. Опробование!

А теперь выполните небольшое упражнение.

13.6.1 Exercise

Добавьте маршрут `DELETE /order/:id`, который удалит заказ по идентификатору, указанному в URL.

Вот несколько советов, которые могут вам помочь:

- параметры URL определяются в стиле Express.js (например, `:id`), а не в стиле API Gateway и Claudia API Builder (например, `{id}`);
- удалить запись из MongoDB можно вызовом функции `collection.deleteOne`;
- не забудьте преобразовать идентификатор заказа в идентификатор MongoDB вызовом новой функции `mongodb.ObjectID`.

Если вам это задание покажется слишком простым, попробуйте реализовать аутентификацию в приложении Express.js. (Можете также попробовать запустить существующее приложение Express.js в AWS Lambda, если у вас оно есть. Для этого дополнительного задания мы не дадим никаких советов.)

13.6.2. Решение

Решение этого упражнения напоминает реализацию маршрута `POST /orders`.

Вы должны добавить новый маршрут `DELETE` в файл `app.js` в виде метода `app.delete`. Внутри этого метода нужно подключиться к базе данных и с помощью функции `db.collection('orders').collection.deleteOne` удалить запись из коллекции `orders`.

Так как идентификатор заказа передается в виде строки, его следует преобразовать в идентификатор MongoDB с помощью функции `mongodb.ObjectID(req.params.id)`.

Новый маршрут должен выглядеть, как показано в листинге 13.11.

Листинг 13.11. Маршрут удаления заказа

```
app.delete('/orders/id', (req, res) => {
  connectToDatabase(process.env.MONGODB_CONNECTION_STRING)
    .then((db) => {
      return db.collection('orders').collection.deleteOne({
        _id: new mongodb.ObjectID(req.params.id)
      })
    })
    .then(result => res.send(result))
    .catch(err => res.send(err).status(400))
})
```

Подключиться к базе данных.

Добавить маршрут DELETE /order/:id.

Удалить запись из базы данных.

Преобразовать идентификатор заказа в идентификатор MongoDB.

Вернуть результат.

Или код ошибки 400, если что-то пошло не так.

После развертывания функции командой `claudia update` проверьте работу метода `delete` с помощью `curl` или Postman.

> **ПРИМЕЧАНИЕ.** Не забудьте настроить строку соединения с MongoDB командой `claudia update` с параметром `--set-env` или `--set-env-from-json`.

В заключение

- Приложения Express.js можно запускать в окружении AWS Lambda с помощью библиотеки Claudia и модуля `serverless-express`.
- Для поддержки статического контента в бессерверных приложениях Express.js не требуется никаких модификаций.
- Используйте управляемый экземпляр MongoDB, если только не собираетесь реализовать свой механизм управления масштабированием.
- Сохраняйте соединение с базой данных в переменной за пределами функции-обработчика.
- Для приложений Express.js в окружении AWS Lambda существуют определенные ограничения, например ограничена возможность использования веб-сокетов, а запросы не должны обрабатываться дольше 30 секунд.

Глава 14

Миграция в бессерверное окружение

Эта глава охватывает следующие темы:

- порядок миграции в бессерверное окружение;
- приведение структуры приложения в соответствие с характеристиками провайдера услуг бессерверных вычислений;
- организация архитектуры приложения с учетом требований бизнеса и возможности дальнейшего развития;
- архитектурные различия между бессерверными и традиционными серверными приложениями.

Рано или поздно наступит момент, когда вам придется задуматься об изменении своих бессерверных приложений, о переносе существующих приложений в бессерверное окружение и о том, как такой перенос скажется на вашем бизнесе.

Вас начнут волновать вопросы организации и поддержки большого количества бессерверных функций. Вы можете также задуматься об ограничениях вашего провайдера услуг бессерверных вычислений, таких как «холодный запуск», и как они могут повлиять на приложение. В этой главе мы сначала поговорим об архитектуре бессерверных приложений, а затем рассмотрим некоторые из этих проблем, чтобы вы могли понять, как правильно переносить приложения в бессерверное окружение и как запускать их в эксплуатацию.

14.1. Анализ текущего бессерверного приложения

Перед миграцией в бессерверное окружение желательно сначала рассмотреть существующее бессерверное приложение и особенности организации его

основных служб. На протяжении всей книги вы помогали тетушке Марии и способствовали процветанию ее пиццерии, создав следующие бессерверные службы:

- *прикладной интерфейс* (API) – этот прикладной интерфейс возвращает список доступных пицц, принимает заказы и сохраняет заказы в бессерверной базе данных. Он соединяется со службой доставки, хранит изображения пицц в бессерверном хранилище, а также поддерживает авторизацию;
- *служба обработки изображений* – уменьшает фотографии пицц, создавая миниатюры для отображения в веб- или мобильном приложении;
- *чат-бот для Facebook Messenger* – чат-бот способен по запросу клиента представить список пицц, принять заказ и отправить запрос в службу доставки. Он также предусматривает упрощенную обработку естественного языка, что позволяет ему вести короткие диалоги с клиентами;
- *чат-бот для Twilio SMS* – этот чат-бот тоже может вернуть список пицц и принять заказ;
- *сценарии для голосового помощника Alexa* – поддержка голосового помощника Alexa позволяет клиенту с помощью его устройства Echo запросить список пицц и заказать выбранную пиццу;
- *платежная служба* – эта независимая платежная служба связывается с платежной системой Stripe и позволяет клиенту оплатить заказ;
- *приложение для таксомоторного парка дядюшки Роберто* – мы рассмотрели возможность миграции приложения Express.js дядюшки Роберто в бессерверное окружение. Это приложение никак не связано с тетушкой Марией, но для нас было полезно познакомиться с одним из возможных решений миграции существующих приложений.

Получился довольно обширный список, но чтобы лучше понять приложение и взаимоотношения между его службами, предпочтительнее иметь перед глазами нарисованную диаграмму. Полная схема служб, разработанных для тетушки Марии, показана на рис. 14.1. Поскольку приложение дядюшки Роберто не имеет отношения к системе тетушки Марии, мы опустили его.

Диаграмма наглядно показывает, как работают и как разделены наши бессерверные службы. Но вам может быть интересно узнать, почему службы тетушки Марии организованы именно так, а не иначе, потому что это знание поможет вам перенести в бессерверное окружение свои существующие приложения.

14.2. Миграция существующего приложения в бессерверное окружение

Создание бессерверных приложений с нуля требует изменить свой взгляд на задачу. Но как только вы начинаете думать в терминах бессерверных вычис-

лений, все быстро становится на свое место. С помощью таких инструментов, как Claudia, циклы разработки и развертывания становятся короткими и простыми.

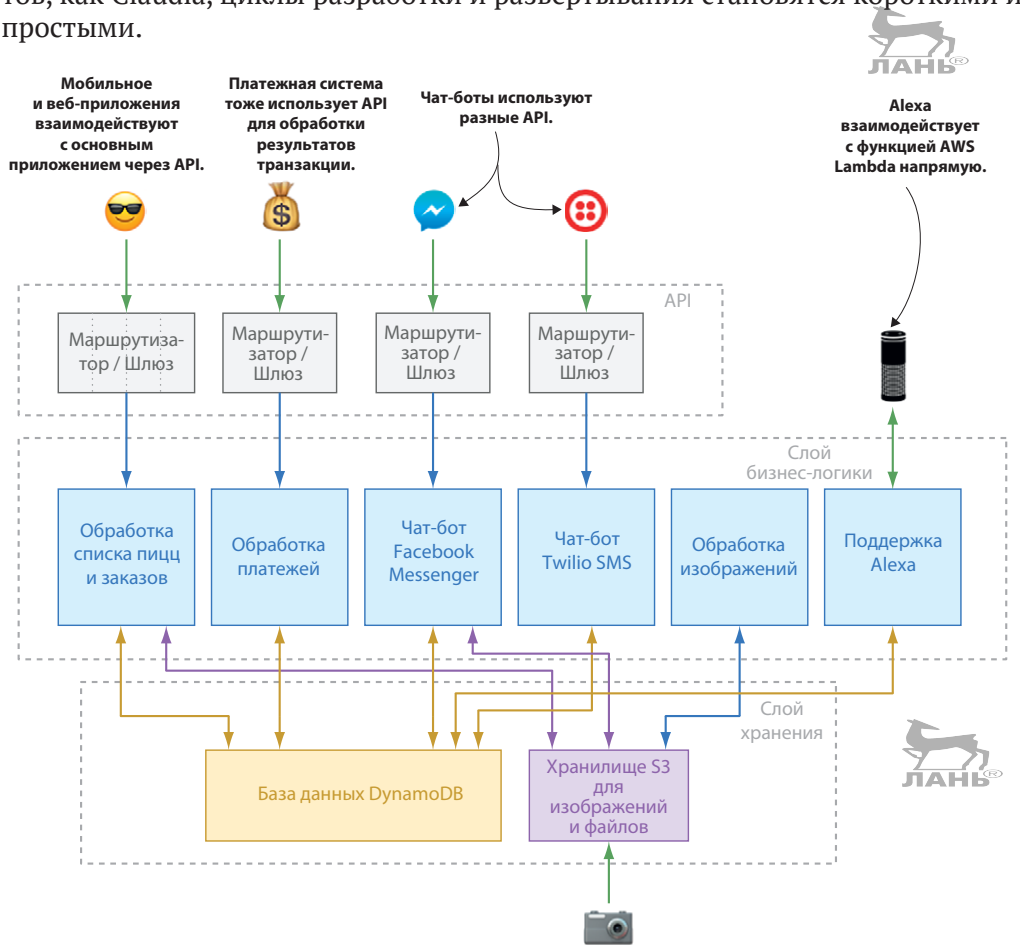


Рис. 14.1. Диаграмма, иллюстрирующая бессерверные службы в приложении для тетушки Марии и отношения между ними

Если у вас уже есть приложение, которое работает и обслуживает клиентов, вы, скорее всего, захотите использовать существующий код, чтобы не начинать с нуля. У вас уже есть приложение с тысячами активных пользователей и тысячами строк кода, сформировавшихся под воздействием бизнес-требований и других проблем.

Можно ли и должны ли вы переносить такое приложение в бессерверное окружение? Ответить на этот вопрос непросто, потому что многое зависит от специфики приложения, структуры вашей команды и многого другого. Но в большинстве случаев миграция в бессерверное окружение может положительно сказаться на приложении.

После миграции бессерверная архитектура будет подталкивать вас к поддержанию приложения в хорошей форме. Она поощряет дальнейшую реорга-

низацию с целью уменьшения затрат; надежный и эффективный код станет хорошим бизнес-решением.

После принятия решения об использовании бессерверных вычислений возникает следующий вопрос: как правильно перенести приложение в бессерверное окружение. Прежде всего начните с малого, с наименее важных компонентов приложения, которые легко отделить от монолита.

У одного из наших клиентов имелась служба, которая преобразовывала каталоги PDF в изображения JPG, чтобы их можно было подписывать, связывать и обслуживать в мобильных приложениях. Служба была частью большего монолитного приложения. После загрузки файла PDF служба обрабатывала его, для каждой страницы генерировала изображение JPG и рассылала мобильные push-уведомления почти 100 000 пользователей, сообщая о доступности нового каталога.

Проблема возникала при попытке выгрузить второй большой каталог PDF сразу вслед за первым. Пользователи, получившие push-уведомление, открывали приложение, и в результате один и тот же сервер вынужден был решать сразу две задачи: обслуживать запросы пользователей и преобразовывать файлы. Так как преобразование документов PDF в изображения JPG является довольно ресурсоемкой процедурой, а процесс автоматического масштабирования действовал с интервалом в две-три минуты, запросы пользователей часто терялись в самый неподходящий момент – когда пользователь нажимал на push-уведомление.

У клиента имелось несколько вариантов решения проблемы, в том числе установить отдельный сервер для обработки PDF (который будет простаивать большую часть времени) или запускать автоматическое масштабирование до того, как это понадобится. Но стоимость инфраструктуры и без того была слишком высока, поэтому клиент решил перенести эту службу на AWS Lambda и сделать ее бессерверной на 100 %.

Всего несколько дней спустя они получили полностью работоспособную службу преобразования PDF в JPG, не зависящую от сервера API. Они могли загружать PDF-файлы прямо в AWS Simple Storage Service (S3) – бессерверную службу хранения статических файлов – из своей панели инструментов. После этого S3 запускала функцию AWS Lambda, которая преобразовывала PDF-файлы в изображения JPG с помощью ImageMagick. Больше об интеграции S3 с AWS Lambda можно прочитать в главе 7, а дополнительную информацию о службе S3 можно узнать на официальном сайте https://aws.amazon.com/s3/.

Поскольку преобразование PDF в JPG происходит довольно медленно, а некоторые каталоги PDF содержат по нескольку сотен страниц, они использовали *шаблон проектирования «Ветвление»*: первая функция Lambda получает запрос и загружает файл PDF, а затем для каждой страницы запускает другую функцию Lambda, используя рассылку событий через службу простых уведомлений (Simple Notification Service, SNS). После преобразования всех страниц первая функция соединялась с API для отправки push-уведомлений всем пользователям. Порядок работы службы преобразования показан на рис. 14.2.

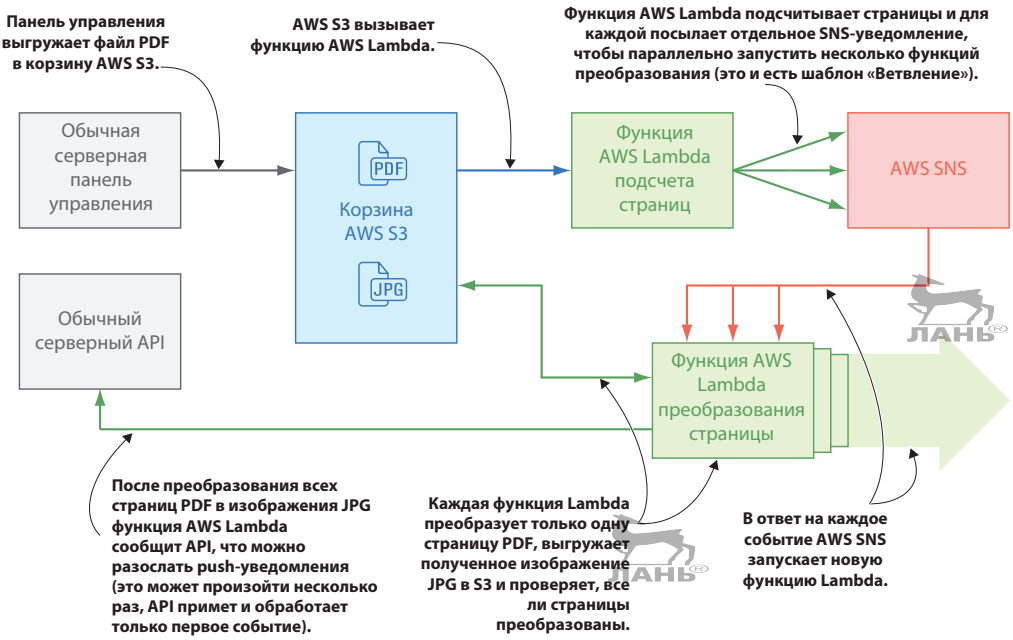


Рис. 14.2. Перенос малой части приложения в бессерверное окружение: преобразование PDF в JPG с использованием функций Lambda и шаблона проектирования «Ветвление»

Шаблон проектирования «Ветвление»

В бессерверной архитектуре функции – это узкоспециализированные компоненты, имеющие ограниченную область ответственности и выполняющие одно или небольшое количество действий. Они не подходят для выполнения продолжительных операций или фоновых процессов.

Поскольку нашим приложениям часто требуется обрабатывать большие объемы данных или выполнять продолжительные операции, вокруг бессерверных функций начал развиваться новый набор шаблонов проектирования. Одним из наиболее полезных является шаблон «Ветвление». Ветвление ускоряет продолжительные операции и может моделировать фоновые процессы, распределяя работу между многими функциями. Идея заключается в том, что одна функция получает запрос, а затем вызывает несколько других функций и каждой делегирует свою порцию работы.

Этот шаблон может пригодиться для реализации медленных операций, таких как преобразование файлов PDF в изображения JPG или для пакетной обработки данных и во многих других случаях. Инициировать ветвление в AWS можно с помощью AWS SDK или другой службы, например AWS SNS.

Если перенос одной службы в бессерверное окружение прошел успешно, следуйте дальше тем же путем, разделяя монолитное приложение на отдельные службы шаг за шагом. Также можно использовать прием, описанный в главе 13, и попробовать запустить все приложение Express.js в AWS Lambda. Это хороший способ начать использовать бессерверные вычисления, но не стоит рассматривать такой шаг как окончательное решение. Перенос монолитного приложения в AWS Lambda целиком не сделает его быстрее и дешевле, все может получиться с точностью до наоборот. Миграция на бессерверную платформу потребует от вас изменить свои привычки. И в этой главе мы рассмотрим некоторые из основных сложностей, таких как холодный запуск.

Другой подход заключается в том, чтобы поместить API Gateway перед приложением и заменять маршруты по одному с использованием функций AWS Lambda или других бессерверных компонентов, соответствующих вашим потребностям (рис. 14.3). После этого вы сможете понаблюдать за своими службами и оптимизировать их.

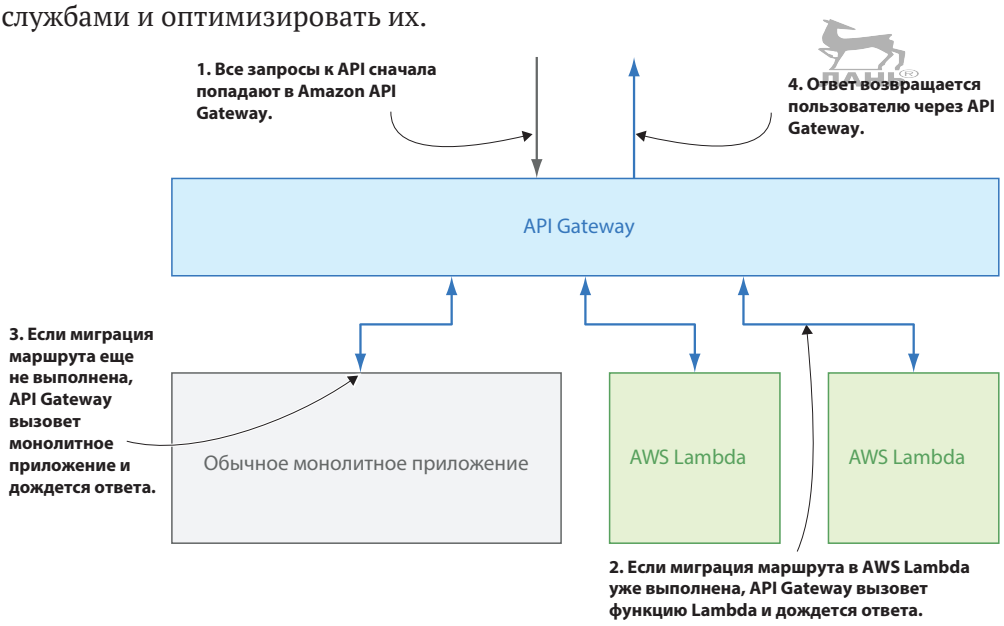


Рис. 14.3. Пошаговая миграция API в бессерверное окружение

Переход от маршрутов к функциям AWS Lambda относительно прост; труднее реализовать другие части приложения, такие как аутентификацию и авторизацию или доступ к базам данных. Чтобы перевести все ваше приложение на бессерверную основу, потребуется охватить все его составляющие, а не только функции.

14.3. Общий взгляд на платформу

Бессерверная архитектура обещает определенные выгоды, такие как низкая стоимость, скорость и стабильность. Но чтобы получить эти выгоды, недоста-

точно использовать ограниченное подмножество бессерверных технологий и продолжать применять те же принципы, что применяются при разработке традиционных серверных приложений. Вам придется пойти ва-банк и организовать приложение так, чтобы оно включало только бессерверные службы и позволяло пользователям подключаться к ним напрямую.

Если пользователь подключается напрямую к базе данных или к хранилищу файлов, это антишаблон. Но в бессерверной среде в сочетании с другими службами, такими как Cognito, этот подход становится шаблоном, способным значительно снизить стоимость вашей инфраструктуры.

В этом разделе обсуждаются некоторые вопросы, которые мы часто слышим от людей, пытающихся перенести свои существующие приложения в бессерверное окружение.

14.3.1. Обслуживание статических файлов

Так же как традиционные серверные приложения (как было показано в главе 13), API Gateway и AWS Lambda способны обслуживать статические файлы, такие как документы HTML и изображения. Но статические файлы значительно увеличивают стоимость (и иногда задержки) услуги бессерверных вычислений из-за масштабирования, потому что каждый раз, когда пользователь запрашивает файл, вы будете платить за использование API Gateway для получения запроса и возврата ответа, а также за использование AWS Lambda для обработки запроса и передачи данных.

Стоимость может показаться небольшой, но API Gateway обходится намного дороже, чем Amazon S3. Кроме того, обработка статических файлов посредством API Gateway и AWS Lambda может повлиять на ваши ограничения и помешать обработке более важных запросов. Итак, как лучше обслуживать статические файлы в бессерверной архитектуре?

Вы должны дать пользователю возможность напрямую взаимодействовать с Amazon S3, когда это возможно. Если потребуется ограничить доступ для определенных пользователей, используйте Cognito. Чтобы позволить выгружать файлы только определенным пользователям, применяйте предварительно подписанный URL (см. главу 7).

14.3.2. Сохранение состояния

Другой важный вопрос касается управления состоянием в бессерверных приложениях. Существует распространенное заблуждение, что функции AWS Lambda не имеют состояния. Но это не так, и обращение с ними как с компонентами, не имеющими состояния, может привести вас к дополнительным затратам как с точки зрения времени выполнения, так и величины оплаты.

По словам Гойко Адзича (Gojko Adzic), создателя Claudia и MindMup, популярного инструмента для создания интеллект-карт, бессерверное окружение следует рассматривать не как окружение без состояния, а как архитектуру *без совместно используемых ресурсов*. Под каждой бессерверной функцией есть

виртуальная машина (ВМ), но вы не знаете, как долго она будет существовать и будет ли эта же ВМ обрабатывать ваш следующий запрос.

> **Архитектура без совместно используемых ресурсов**
>
> Архитектура без совместно используемых ресурсов (shared-nothing (SN) architecture) – это архитектура распределенных вычислений, в которой каждый узел является независимым и самодостаточным и нет единственной точки, где возникает конкуренция. Проще говоря, никакие узлы не используют общую память или дисковое пространство. Люди часто сравнивают SN-системы с системами, хранящими большое количество информации о состоянии в централизованном хранилище – в базе данных, на сервере приложений или в любой другой подобной точке конкуренции. Узнать больше об архитектуре без совместно используемых ресурсов можно по адресу https://en.wikipedia.org/wiki/Shared-nothing_architecture.

Служба AWS Lambda не позволяет хранить состояние, ее главная задача – оптимизация выполнения. Например, в главе 13 мы запускали приложение Express.js за пределами функции-обработчика и тем самым повысили производительность обработки запросов, которые повторно попадают в одну и ту же виртуальную машину. Для долговременного хранения состояния следует использовать другую службу, такую как DynamoDB или даже S3, в зависимости от сложности состояния, которое требуется сохранить.

Если вы реализуете машину, действующую в зависимости от текущего состояния (конечный автомат), вам может пригодиться AWS Step Functions. С помощью этой службы вы легко сможете координировать работу компонентов распределенных приложений и микросервисов, используя визуальные инструменты, и иметь возможность сохранения промежуточных состояний. Узнать больше об AWS Step Functions можно по адресу https://aws.amazon.com/step-functions/.

14.3.3. Журналы

Как рассказывалось в главе 5, CloudWatch имеет встроенную интеграцию с другими компонентами бессерверного окружения, такими как AWS Lambda и API Gateway. Но CloudWatch не самое лучшее решение для журналирования.

К счастью, есть и другие варианты, улучшающие работу с журналами в бессерверном окружении, например сторонние решения или запуск функций Lambda или Elasticsearch из CloudWatch.

В числе наиболее популярных сторонних решений можно назвать IOpipe (https://www.iopipe.com), службу мониторинга, которая позволяет видеть параметры производительности функций, рассылать оповещения в реальном времени и выполнять трассировку распределенного стека. Настроить IOpipe довольно просто: вы регистрируетесь в службе, получаете идентифи-

катор клиента, устанавливаете модуль IOpipe из NPM командой `npm install @iopipe / iopipe --save`, а затем заключаете свой обработчик в функцию `iopipe`, как показано в листинге 14.1.

Листинг 14.1. Заключение обработчика в функцию iopipe

```
const iopipe = require('@iopipe/iopipe')

const iopipeWrapper = iopipe({
  clientId: process.env.CLIENT_TOKEN
})

exports.handler = iopipeWrapper(
  function(event, context, callback) {
    // Здесь должен находиться код обработчика
  }
)
```

← Импортировать модуль IOpipe.

Сгенерировать функцию-обертку с использованием идентификатора клиента.

Заключить обработчик в функцию-обертку IOpipe.

Интеграцию IOpipe с Claudia API Builder можно реализовать, как показано в листинге 14.2.

Листинг 14.2. Интеграция IOpipe с Claudia API Builder

```
const iopipe = require('@iopipe/iopipe')

const iopipeWrapper = iopipe({
  clientId: process.env.CLIENT_TOKEN
})

// Определения маршрутов

api.proxyRouter = iopipeWrapper(api.proxyRouter)
module.exports = api
```

Интеграция IOpipe с функцией AWS Lambda с помощью api.proxyRouter.

Другой вариант – потоковая передача журналов в функцию AWS Lambda или службу Amazon Elasticsearch. Это решение позволяет организовать потоковую передачу всех журналов и подключить журналы к другим обычным инструментам, таким как стек Elastic. Стек Elastic, также известный как стек ELK, – это комбинация из трех программных продуктов с открытым исходным кодом (Elasticsearch, Logstash и Kibana), которые помогают организовать анализ и визуализацию журналов. Узнать больше о стеке Elastic можно по адресу https://www.elastic.co/elk-stack.

СОВЕТ. Организовать потоковую передачу журналов в функцию AWS Lambda или службу Amazon Elasticsearch в единственный поток журналов или в группу мож-

но с помощью настроек журналирования CloudWatch в веб-консоли AWS. Узнать больше о потоковой передаче журналов в службу Amazon Elasticsearch можно по адресу https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_ES_Stream.html.

Какой вариант лучше?

Ответ на этот вопрос зависит от вашего приложения, используемых инструментов и ваших личных предпочтений. Сторонние библиотеки журналирования предлагают более широкие возможности и доступ к данным, отсутствующим в CloudWatch. Но, как показано на рис. 14.4, они также увеличивают время задержки вашей функции. В большинстве случаев время выполнения увеличивается ненамного, но так как стоимость услуги бессерверных вычислений определяется за 100-миллисекундные интервалы, это может увеличить ваш счет.

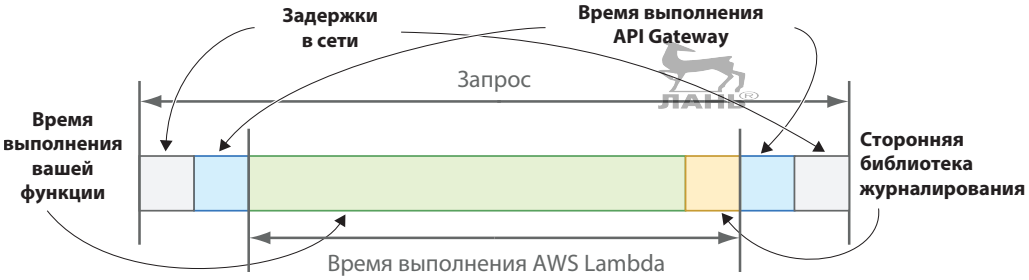


Рис. 14.4. Продолжительность обработки запроса с использованием сторонних библиотек журналирования

Возможно, хорошей идеей будет начать со сторонней библиотеки журналирования, понаблюдать за эффектом и оптимизировать использование библиотеки или заменить ее встроенными средствами журналирования либо, может быть, стеком Elastic.

14.3.4. Непрерывная интеграция

Одним из больших преимуществ бессерверной инфраструктуры является возможность заставить всех членов вашей команды выполнять развертывание одной командой. Правда, при этом возникает несколько потенциальных проблем, таких как тестирование или откат в случае сбоя.

Традиционно некоторые проблемы с частыми развертываниями решаются путем непрерывной интеграции. Непрерывная интеграция – это практика разработки, требующая от разработчиков интегрировать код в общий репозиторий несколько раз в день. После каждой отправки кода выполняется автоматическая сборка, что позволяет командам обнаруживать проблемы на ранних этапах. Иначе говоря, непрерывная интеграция позволяет быстро обнаруживать ошибки и исправлять их.

Вот несколько популярных инструментов для непрерывной интеграции:

- Jenkins (https://jenkins.io);
- Travis CI (http://travis-ci.org);
- Semaphore CI (http://semaphoreci.com).

Все эти инструменты прекрасно работают с бессерверными приложениями в AWS. Для интеграции с ними вам потребуется сохранить файл `claudia.json` в вашей системе управления версиями и выполнить команду `claudia update` после успешного выполнения всех комплектов тестов. Но не забудьте сохранить свои ключи доступа к AWS в переменных окружения.

Кроме перечисленных выше популярных инструментов, AWS предлагает ряд своих инструментов поддержки непрерывной интеграции, которые вы можете использовать в своих бессерверных приложениях:

- CodePipeline – используется для моделирования, визуализации и автоматизации этапов развертывания бессерверных приложений (http://docs.aws.amazon.com/codepipeline/latest/APIReference/);
- CodeBuild – используется для сборки, локального тестирования и упаковки бессерверных приложений (http://docs.aws.amazon.com/codebuild/latest/userguide/);
- AWS CloudFormation – используется для развертывания бессерверных приложений (http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/);
- CodeDeploy – используется для пошагового развертывания обновлений бессерверных приложений (https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html).

ПРИМЕЧАНИЕ. Некоторые из инструментов, доступных на платформе AWS, плохо работают с Claudia. Например, AWS CloudFormation – бесплатная служба, предлагающая инструменты для создания и управления инфраструктурой AWS. Это конкретное приложение должно быть запущено в Amazon Web Services. Оно развертывает функции AWS Lambda и другие части вашего серверного приложения.

Если ваше приложение состоит из большого количества разных компонентов, обратите внимание на CloudFormation как на потенциальное решение для управления приложением.

14.3.5. Управление окружениями: промышленное окружение и окружение для разработки

Каждый раз, когда публикуется функция Lambda, ей присваивается порядковый номер сборки. Вы можете вызвать определенную версию и настроить триггеры для запуска определенной версии, что упрощает откат развертывания и одновременное использование нескольких версий.

Кроме числовых номеров сборки, в AWS Lambda также поддерживаются *псевдонимы* – именованные указатели на конкретный числовой номер версии, – позволяющие использовать одну функцию Lambda в окружениях для эксплуатации, разработки и тестирования.

Например, во время разработки можно развернуть новую версию функции Lambda и отметить ее псевдонимом `development`, а затем присвоить ей псевдоним `testing` и тщательно протестировать. Наконец, убедившись, что функция работает должным образом, вы можете присвоить тому же числовому номеру версии псевдоним `production` и запустить эту версию в производство.

Поскольку триггеры позволяют настраивать запуск функций по псевдонимам, как только ваш флаг `production` начнет ссылаться на новую версию, триггеры в промышленном окружении начнут вызывать ее без дополнительных настроек.

> **ПРИМЕЧАНИЕ.** Некоторые источники событий, такие как триггер CloudFront для Lambda@Edge, не поддерживают псевдонимы и требуют явно указывать числовую версию функции Lambda. В большинстве этих случаев библиотека Claudia автоматически определит числовую версию для псевдонима и настроит триггер. Дополнительную информацию о Lambda@Edge ищите по адресу https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html. Инструкции по развертыванию Lambda@Edge с помощью Claudia ищите по адресу https://claudiajs.com/news/2018/01/04/claudia-3.html.

При создании бессерверных функций с поддержкой различных окружений важно помнить, что функция не зависит от окружения. Вы никогда не должны писать код, жестко определяющий используемые службы, такие как S3 или имя таблицы в DynamoDB. Вместо этого применяйте ту же корзину, откуда пришло событие, или получайте имя таблицы из переменной окружения.

14.3.6. Совместное использование конфиденциальных данных

Одной из ключевых составляющих успеха бессерверных приложений с поддержкой нескольких окружений является управление ключами доступа. В этой книге мы управляли ключами двумя способами: в переменных стадий API Gateway и переменных окружения AWS Lambda. Оба имеют свои сильные и слабые стороны, и какой из них использовать, будет зависеть от вашей ситуации и предпочтений.

Если вы используете псевдонимы для управления этапами тестирования/эксплуатации, переменные окружения Lambda будут привязаны к числовой версии функции Lambda, а не к псевдонимам, а значит, все псевдонимы, указывающие на одну и ту же версию сборки, будут использовать одни и те же переменные окружения. Например, если вы настроили псевдонимы `production` и `development` для версии 42 функции Lambda, они будут использовать одну и

ту же переменную окружения TABLE_NAME. На рис. 14.5 наглядно показано, как работают переменные окружения Lambda.

В отличие от переменных окружения Lambda, переменные стадий API Gateway привязаны к стадиям API Gateway, поэтому, как показано на рис. 14.6, две стадии API Gateway могут ссылаться на один и тот же номер сборки Lambda и иметь разные значения переменных.

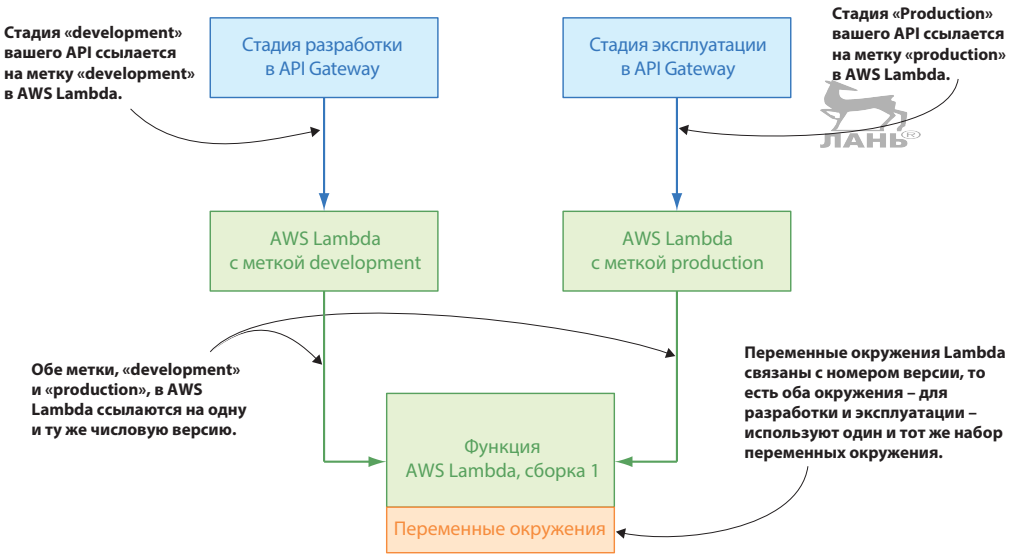


Рис. 14.5. Как работают переменные окружения Lambda

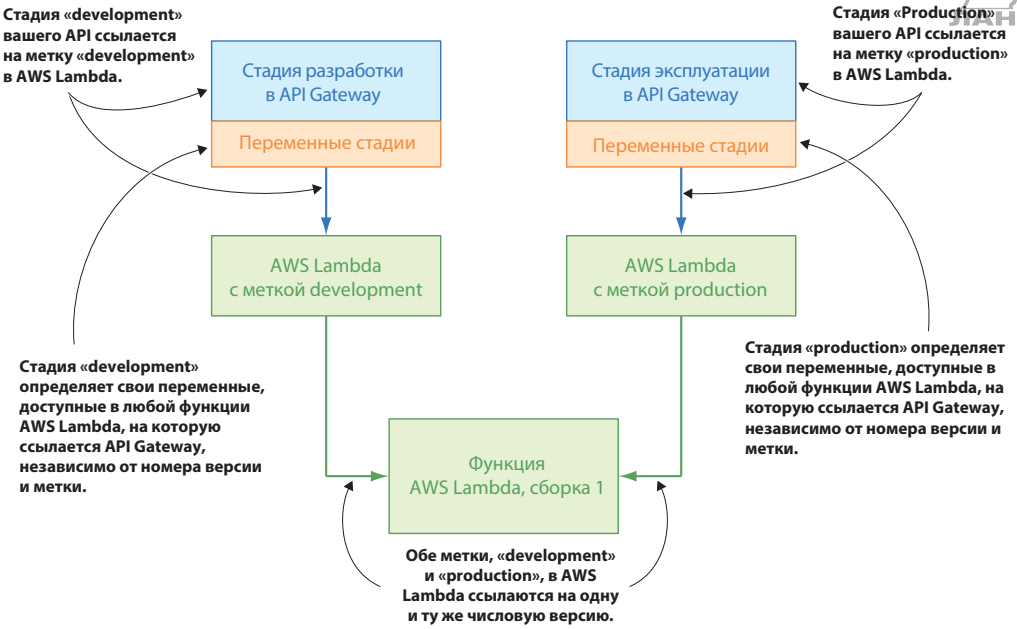


Рис. 14.6. Как работают переменные стадий в API Gateway

В новом развертывании повторно используются переменные окружения Lambda из предыдущих версий, если вы не определите новый набор переменных. Это означает, что переменные из вашей среды разработки будут переданы в эксплуатационную среду, если выполнить развертывание с помощью `claudia update --version production` без использования флагов `--set-env` или `--set-env-from-json`. Кроме того, чтобы изменить переменные окружения Lambda, необходимо снова определить все активные переменные, потому что каждое изменение переопределяет все существующие переменные. Например, если вы решите изменить `TABLE_NAME`, но оставить переменную `S3_BUCKET`, вам придется снова определить обе переменные, иначе переменная `S3_BUCKET` будет потеряна. С другой стороны, библиотека Claudia способна помочь в этой ситуации: она поддерживает дополнительную команду `--update-env`, которую можно использовать для изменения одной переменной окружения без необходимости повторно определять другие.

Переменные стадии API Gateway сохраняются для каждой стадии. Это означает, что если отправить новую версию в стадию разработки, она получит все переменные этой же стадии из предыдущей версии. При желании можно добавить одну переменную стадии, и она не повлияет на другие переменные.

API Gateway позволяет изменить единственную переменную стадии без повторного определения остальных.

Но переменные окружения Lambda имеют свои сильные стороны. Например, они хранятся в зашифрованном виде, что делает их более безопасными, чем переменные стадии API Gateway, которые не шифруются. Их также можно использовать независимо от события, вызвавшего функцию AWS Lambda.

Общим недостатком переменных стадий API Gateway и окружения Lambda является невозможность их совместного использования разными функциями Lambda. Если у вас много функций, которые используют общие конфиденциальные данные, например имя таблицы в DynamoDB, вы вынуждены будете передать одну и ту же переменную каждой из них. Например, таблица `pizza-orders` в DynamoDB используется в Pizza API, а также сценарием голосового помощника Alexa, поэтому мы были вынуждены передать имя таблицы обеим функциям Lambda.

Этот недостаток можно устранить с помощью хранилища параметров AWS Systems Manager Parameter Store, которое обеспечивает доступ к центральному, безопасному, надежному и высокодоступному хранилищу, предназначенному для хранения конфигураций приложений и конфиденциальных данных. Оно легко интегрируется с AWS Identity and Access Management (IAM), чтобы обеспечить точное управление доступом к отдельным параметрам или ветвям иерархического дерева. Одним из недостатков хранилища параметров является дополнительная задержка. Дополнительную информацию об AWS Systems Manager Parameter Store можно найти на странице https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html.

14.3.7. Виртуальное частное облако

Принимая решение о переходе на бессерверные вычисления, можно столкнуться с проблемой соблюдения определенных законов и правил – например, правил хранения и обработки персональных данных или даже специальных правил безопасности сети в самой компании. Эти ограничения могут помешать использовать AWS Lambda или другие бессерверные ресурсы.

К счастью, как раз для таких случаев поставщики услуг бессерверных вычислений выработали решение, которое называется *виртуальным частным облаком* (Virtual Private Cloud, VPC). VPC – это услуга, позволяющая создавать бессерверные ресурсы в виртуальной частной сети. Она дает возможность полностью контролировать свое сетевое окружение, например диапазоны IP-адресов, сетевые шлюзы и т. д. Нахождение ваших бессерверных ресурсов в виртуальной частной сети обеспечит повышенную безопасность и поможет вашей компании разместить свои ресурсы в определенных областях, регионах или странах. Например, если вы имеете дело с конфиденциальными данными клиентов (которые должны храниться в стране проживания клиента), VPC позволит вам разместить ваши бессерверные ресурсы в той же сети, что и ваш центр обработки данных в этой стране.

Проще говоря, VPC позволяет организовать виртуальную частную сеть с ресурсами вашего поставщика услуг бессерверных вычислений (например, AWS Lambda) и ограничивать доступ к этим ресурсам, чтобы они были доступны только экземплярам или ресурсам в вашем облаке VPC.

Однако VPC имеет свои недостатки. Закрытая сеть имеет определенные проблемы с холодным запуском, потому что AWS Lambda требует создания сетевых интерфейсов Elastic Network Interfaces (ENI) для VPC. Создание ENI по одному запросу легко может добавить до 10 секунд к холодному старту. Кроме того, функция Lambda в VPC по умолчанию не имеет доступа к интернету, который можно настроить с помощью шлюза преобразования сетевых адресов (Network Address Translation, NAT). Поэтому будьте осторожны при их использовании.

14.4. Оптимизация приложения

Переход на использование бессерверных вычислений действительно может снизить стоимость инфраструктуры для вашего приложения, но только если все сделано правильно. Самое важное, о чем следует помнить, – бессерверные вычисления все еще являются довольно новой технологией, и некоторые передовые методы проектирования и программирования теряют свою актуальность при ее использовании и могут даже привести к обратным результатам. Экономия затрат на бессерверное приложение видна не только в счетах от AWS; бессерверное приложение может также обеспечить значительную экономию за счет более короткого времени выхода на рынок, повышения эффективности и сокращения времени реакции на изменения на рынке.

Несмотря на появление новых удачных приемов и шаблонов проектирования, единственный способ создать хорошее бессерверное приложение –

это непрерывное наблюдение и оптимизация, которая поможет снизить затраты на поддержку вашего приложения и повысит удобство для пользователей.

14.4.1. Связанные и узкоспециализированные функции

Взгляните на диаграмму бессерверного приложения для тетушки Марии (рис. 14.1) и обратите внимание, что для получения прейскуранта и заказа пиццы используется одна и та же бессерверная функция. Эта функция решает несколько задач. Разве это не монолитное решение, завернутое в функцию? Многие могут сказать, что это пример неправильного использования бессерверных функций. Такие утверждения основаны на идее FaaS (Function as a Service – функция как услуга), согласно которой каждая функция должна иметь единственную цель и в бессерверных функциях не должно быть монолитов, потому что при монолитной организации теряются преимущества слабой связанности, возможности повторного использования и более простого обслуживания.

Другие, напротив, могут утверждать, что некоторые независимые службы (например, платежную службу) можно было бы объединить в один API. Поскольку эти службы используются не так часто, их работа может замедляться из-за холодных запусков; было бы лучше иметь «подогретую» бессерверную функцию и не заставлять пользователя ждать, пока запустится ваша платежная служба.

Обе точки зрения имеют право на жизнь. Но какой подход выбрать? Прежде всего помните известную поговорку «на вкус и цвет товарищей нет». Ваша цель – реализовать услугу, удобную для клиента. Выбирайте наиболее рациональный подход, который часто зависит от типа приложения.

Первоначально разделите функции по сферам ответственности, как это было сделано, например, в бессерверном приложении тетушки Марии. Выделите функции, связанные с пиццей и с оплатой, а затем выделите функции для каждой из дополнительных услуг, таких как чат-боты или обработка изображений. Позднее, когда ваша система начнет расти, попробуйте выделить узкоспециализированные функции, например попробуйте разделить службу вывода прейскуранта и оформления заказа на две функции. С ростом числа клиентов холодные запуски будут происходить все реже, и приложение будет откликаться быстрее.

14.4.2. Выбор правильного объема памяти для функции Lambda

Каждой бессерверной функции выделяется некоторый объем памяти с определенной организацией. Да, бессерверная архитектура не требует настройки сервера, но для решения некоторых задач может потребоваться больше памяти или вычислительной мощности. Поэтому поставщики услуг бессерверных вычислений дают возможность указать, сколько памяти выделить функции. Обратите внимание, что мы ничего не говорим о вычислительной

мощности, потому что она прямо связана с объемом памяти, а это значит, что если вам понадобится увеличить вычислительную мощность, вы должны увеличить объем выделяемой памяти. Например, если вы настроили выделение функции Lambda 2 Гб памяти вместо 1 Гб, она наверняка получит в свое распоряжение больше процессорного времени. За более точной информацией об этом и возможных изменениях в будущем мы предлагаем обратиться к документации AWS, описывающей настройку Lambda: https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html.

Но увеличение объема памяти увеличивает стоимость услуги. Этот бесплатный миллион запросов в месяц может легко превратиться в сотню тысяч, если увеличить объем памяти для функции Lambda до 1 Гб. Выбор правильного объема памяти для функции Lambda – сложная задача. Решая ее, легко попасть в одну из следующих ловушек:

- минимизация объема памяти для функции Lambda с целью уменьшить стоимость услуги, пытаясь угадать, сколько памяти или вычислительной мощности необходимо;
- максимизация объема памяти для функции Lambda, чтобы ускорить обработку запросов и сопутствующие вычисления, а также чтобы подготовиться к любым возможным всплескам в потреблении памяти.

И снова вспомните поговорку «на вкус и цвет товарищей нет». Попробуйте использовать инструменты журналирования и/или мониторинга для сбора информации об использовании памяти. Например, в журналах CloudWatch, кроме всего прочего, можно узнать, сколько памяти использовала функция и сколько времени потребовалось ей для выполнения. Основываясь на этих цифрах, вы сможете правильно оценить потребности, но всегда старайтесь учитывать показатели за разное время суток, а также учитывать конкретные события.

14.5. Преодоление проблем

Новая архитектура влечет за собой новый набор проблем, но и старые проблемы никуда не исчезают – они все еще могут встречаться, хотя и немного в другом обличье. Новые проблемы, характерные для бессерверных вычислений, – это тайм-ауты и холодные запуски. Но вам также придется столкнуться с некоторыми старыми проблемами, такими как привязка к производителю, безопасность и распределенные атаки типа «отказ в обслуживании» (Distributed Denial of Service, DDoS). В этом разделе рассматриваются некоторые из наиболее серьезных проблем, с которыми вы наверняка столкнетесь при переходе на бессерверную архитектуру.

14.5.1. Тайм-ауты

Одна из первых проблем, с которыми вы можете столкнуться при переходе, – ограничения бессерверных функций, к числу которых относится и тайм-аут.

Тайм-ауты позволяют функциям безопасно останавливать работу, чтобы не тратить деньги и время, если по какой-то причине они зависают или блокируются. Ограничение времени ожидания заставит вас задуматься о том, как завершить обработку запроса в указанные сроки. Это само по себе может быть проблемой, но настоящая проблема заключается в том, как выявлять и отлаживать проблемы превышения тайм-аута. Кроме того, функция может работать безошибочно, но требовать больше времени на выполнение, чем ожидалось.

Есть несколько способов преодолеть проблему превышения тайм-аута: первый и самый простой – просмотреть журналы CloudWatch, в которых регистрируются все такие случаи. Однако в этом мало проку, потому что этот подход не предполагает ничего для обработки таких ситуаций. Есть лучший способ, позволяющий обрабатывать случаи превышения тайм-аута и хотя бы зафиксировать, какой службе потребовалось слишком много времени на выполнение, или выявить причину. Решение заключается в создании *сторожевого таймера*, единственная цель которого состоит в том, чтобы дать возможность определить, когда истекает время тайм-аута для вашего приложения. Это простой таймер, который нужно добавить в свою функцию Lambda, чтобы с его помощью можно было узнать, сколько времени осталось до принудительного прерывания функции. Он рассчитывает оставшееся время, исходя из настройки времени тайм-аута, которая определяется в панели управления на веб-сайте AWS Console.

Сторожевой таймер определяет, когда функция приближается к границе тайм-аута. В этом случае таймер вызывает другую функцию Lambda, которая регистрирует это событие или обрабатывает его как-то иначе. Чтобы реализовать сторожевой таймер, вам нужно создать функцию таймера, которая постоянно проверяет наступление момента, когда до истечения тайм-аута останется менее одной секунды, вызывает другую функцию Lambda и пересылает ей текущий контекст функции. После этого вы сможете зафиксировать информацию о событии в журнале или предпринять что-то еще.

Вместо другой функции AWS Lambda можно также использовать сторонние службы обработки ошибок, такие как Bugsnag (https://www.bugsnag.com) или Sentry (https://sentry.io).

14.5.2. Холодный запуск

Еще одна проблема, с которой вы столкнетесь при использовании бессерверных приложений, – это задержки, вызванные *холодным запуском*. AWS автоматически управляет масштабированием и созданием новых контейнеров, поэтому первый вызов каждой функции происходит с небольшой задержкой. Это связано с необходимостью запустить контейнер и инициализировать вашу функцию. После первого вызова функция будет оставаться *горячей* в течение определенного времени (не более нескольких минут), благодаря чему сможет быстрее обрабатывать последующие запросы (рис. 14.7).

Однако холодный запуск происходит не только при первом вызове функции. Если потребуется параллельно (или почти параллельно) запустить не-

сколько экземпляров функции, для каждого будет выполнена процедура холодного запуска, потому что AWS может понадобиться запустить несколько виртуальных машин для обработки всех ваших запросов (рис. 14.8).

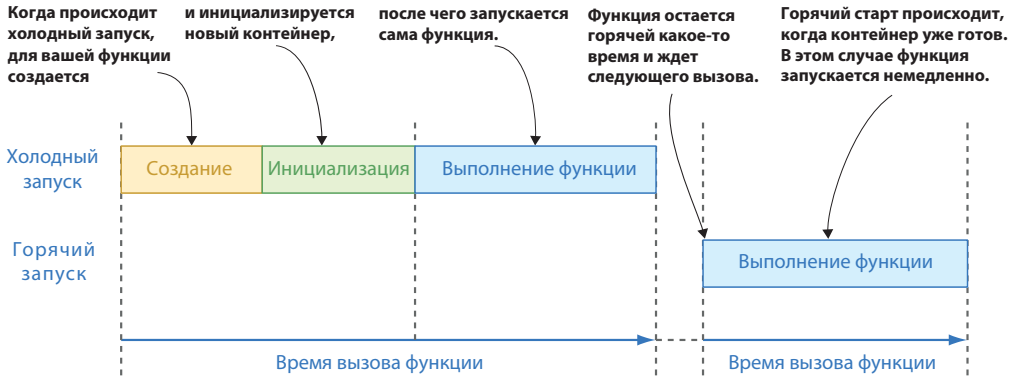


Рис. 14.7. Холодный и горячий запуски функции

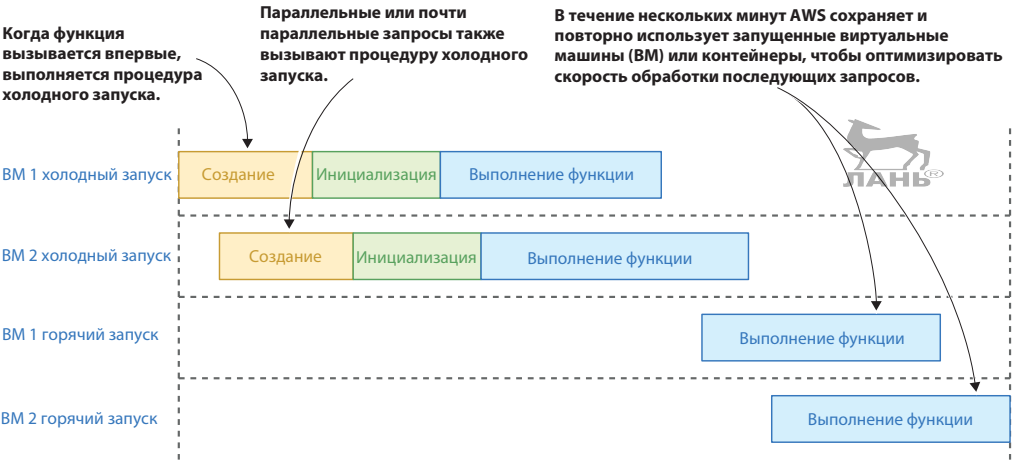


Рис. 14.8. Холодный запуск имеет место, когда возникает необходимость параллельной обработки запросов

Есть ли методы борьбы с холодными запусками? Увы, вы не сможете избежать их полностью. Можно попытаться предварительно подогреть определенное количество своих функций, но это увеличивает сложность реализации и заставляет попытаться предсказать количество запросов, которые функция получит при пиковой нагрузке. Другой и, вероятно, лучший вариант – сохранить функции как можно более короткими и быстрыми (не более нескольких мегабайт), потому что холодный запуск коротких функций происходит быстрее.

Кроме того, выбор языка программирования и используемых библиотек напрямую влияет на стоимость размещения бессерверного приложения, что делает этот выбор важным бизнес-решением. Функции на Node.js или Golang

обойдутся вам значительно дешевле, чем функции на Java, из-за более быстрой инициализации.

14.5.3. Атаки DDoS

Бессерверное окружение полностью меняет бизнес-модель, потому что плата взимается за использованное время, а не за зарезервированное. Это здорово! Но у вас может возникнуть вопрос о противодействии DDoS-атакам. Выполняя такие атаки, злоумышленник отправляет в приложение большой объем бессмысленных данных, из-за чего приложение оказывается не в состоянии достаточно быстро отвечать на действительные запросы клиентов и тем самым препятствует оказанию услуг вашим клиентам. Так как ваш поставщик услуг бессерверных вычислений обеспечивает автоматическое масштабирование и балансировку нагрузки, у вас может возникнуть опасение, что DDoS-атака способна обанкротить вас. Однако на самом деле шансы на это практически отсутствуют, потому что ваш провайдер бессерверных услуг (например, AWS) способен намного лучше противостоять таким атакам, чем типичные провайдеры, предлагающие хостинг на серверах.

Кроме того, у вас есть возможность определить следующие настройки:

- максимальный объем трафика на уровне API Gateway level (дополнительные подробности ищите по адресу https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html);
- максимальная степень параллелизма для ваших функций AWS Lambda (дополнительные подробности ищите по адресу https://docs.aws.amazon.com/lambda/latest/dg/concurrent-executions.html);
- настройка предупреждений CloudWatch для отправки уведомлений о необъяснимых пиках активности.

> **ПРИМЕЧАНИЕ.** В бессерверных вычислениях DDoS-атаки называют DDoW-атаками (Distributed Denial of Wallet – распределенный отказ кошелька). Поставщики услуг бессерверных вычислений взимают плату за каждый запрос, поэтому увеличение вашего счета можно интерпретировать как «отказ вашего кошелька».

14.5.4. Привязка к производителю

При использовании бессерверных вычислений одной из основных проблем является привязка к производителю. Легко стать зависимым от ресурсов вашего поставщика услуг бессерверных вычислений и его API. На первый взгляд в этом нет ничего плохого, но когда вы решите перейти к другому поставщику услуг, возникнет проблема. Ресурсы первого поставщика услуг станут для вас недоступными, что может потребовать полного рефакторинга вашего приложения или даже создания его заново.

Иногда это может стать серьезным препятствием. Многие компании стремятся избежать попадания в зависимость от какого-то одного поставщика

ресурсов. Главный аргумент в пользу обычных приложений, размещаемых на сервере, заключается в том, что независимо от поставщика услуг всегда можно установить на свои серверы одни и те же версии необходимых баз данных или инструментов.

Но многие часто путают разные проблемы. Анализируя зависимость от производителя, иногда путают два разных уровня:

- инфраструктура;
- услуга.

Зависимость *на уровне инфраструктуры* подразумевает привязку вашего приложения к конкретной инфраструктуре, тогда как зависимость *на уровне услуги* подразумевает привязку к определенной программной службе (базе данных, хранилищу файлов, службе поиска и т. д.).

В процессе размышлений о переходе на использование бессерверных вычислений у вас может возникнуть соблазн сравнить поставщиков серверных и бессерверных вычислений (таких как AWS Lambda). Но будьте осторожны, потому что в действительности вы будете сравнивать яблоки с апельсинами; бессерверные вычисления – это уровень услуги, тогда как серверные вычисления – это инфраструктурный уровень. Вы можете спросить себя, в чем разница или почему это так важно. Суть в том, что AWS Lambda не поддерживает традиционную роль сервера.

Даже понимая такое деление, некоторые могут подумать: «Но я все еще привязан к производителю. Это ничего не меняет, потому что я все еще должен использовать ресурсы AWS при применении AWS». Все верно, но понимание вскрывает два преимущества:

- переход от одного поставщика услуг бессерверных вычислений к другому можно сравнить с переходом с базы данных MySQL на PostgreSQL;
- использование гексагональной архитектуры вместо прямого взаимодействия с бессерверными ресурсами может практически полностью избавить от проблем, вызываемых привязкой к производителю.

При переходе от одного поставщика услуг бессерверных вычислений к другому используются те же правила, что и при переходе от использования одной службы к другой, и вы сможете безопасно выполнить такой переход, просто изменив взаимодействие с API поставщика услуг.

Существуют бессерверные платформы, которые абстрагируют специфику бессерверных провайдеров, но, используя их, вы окажетесь в зависимости уже от этого бессерверного фреймворка. Кроме того, вы все равно будете зависеть от своего бессерверного провайдера, потому что будете ожидать, что ответы и сообщения, приходящие от службы, будут иметь определенный формат. Например, используя AWS Kinesis, вы будете ожидать получения сообщений в формате Kinesis; вы не сможете просто перейти на использование услуг Google или другого провайдера.

По этой причине библиотека Claudia.js не абстрагирует деталей, характерных для бессерверного провайдера, и поэтому является специфичной для AWS.

Для смягчения этой «привязки к производителю» мы советуем использовать гексагональную архитектуру, в которой вы сами будете определять граничные объекты, единственной целью которых является взаимодействие со специфическим API бессерверного провайдера. Ваша бизнес-логика останется неизменной, а это означает, что для перехода к другому провайдеру потребуется просто изменить логику протокола граничных объектов.

14.6. Опробование!

В этой главе вам предлагается простое, но не самое легкое упражнение: выполните миграцию своего приложения Node.js в бессерверное окружение.

К сожалению, для этого упражнения не существует типового решения. Но мы уверены, что вам не будет скучно и, что еще более важно, это окажет большое влияние на ваш бизнес и ваши взгляды на подходы к разработке приложений в будущем. Удачи!

> **СОВЕТ.** Перед выполнением этого упражнения посетите репозиторий приложений AWS Serverless Application Repository – открытый интернет-магазин бессерверных приложений и компонентов. Возможно, вам удастся найти компонент, который вы сможете просто подключить к своему новому бессерверному приложению. Узнать больше о репозитории приложений можно по адресу https://aws.amazon.com/serverless/serverlessrepo/.

В заключение

- Перенос существующего приложения в AWS Lambda и API Gateway является хорошим началом для перехода на использование бессерверных вычислений, но может вызвать лишние расходы, если использовать не все возможности платформы.
- Миграцию можно выполнить, поместив приложение за API Gateway, а затем перенося маршруты один за другим.
- Чтобы получить все преимущества бессерверных вычислений, такие как низкая стоимость и высокая скорость разработки, необходимо использовать все бессерверные службы.
- При переходе на бессерверные вычисления выбор языка программирования, библиотек и времени проведения рефакторинга становится бизнес-решением с определенными рисками, поскольку этот выбор напрямую влияет на стоимость вашей инфраструктуры. Поэтому для успешного применения бессерверных вычислений важны непрерывное наблюдение и оптимизация.

- Бессерверные вычисления предлагают новую архитектуру, требующую применения новых шаблонов и методов программирования.
- Переход на бессерверные вычисления влечет за собой новые проблемы, такие как холодный запуск и время ожидания. При этом некоторые старые проблемы никуда не исчезают, а иные даже усиливаются, как, например, привязка к производителю.

Глава 15

Примеры из практики

Эта глава охватывает следующие темы:

- использование бессерверных вычислений в препроцессорах CodePen;
- организация клиентских API и инструментов преобразований файлов в MindMup.

Вот и подошло концу наше путешествие по миру бессерверных вычислений. Прочитав эту книгу, вы узнали, что такое бессерверные вычисления, как их использовать для создания новых приложений и как перенести существующие приложения в бессерверное окружение. Но остался еще один важный вопрос, который, как мы понимаем, заинтересует вас: кто использует бессерверные вычисления на практике?

Всегда полезно знать о компаниях, использующих новый подход на практике, а также о проблемах, с которыми они сталкивались, и о том, как они их преодолевали. Другой интересный вопрос: почему эти компании решили, что использование бессерверных вычислений вкупе с Claudia.js является верным выбором?

Чтобы ответить на эти вопросы, мы выбрали две компании, на примере которых познакомим вас с проблемами, с которыми они столкнулись, почему они решили перейти на использование бессерверных вычислений и как теперь оценивают свое решение.

Многие компании используют бессерверные вычисления в своей практике, и нам было довольно трудно выбрать двух наиболее ярких представителей. Но поскольку в блоге AWS вы можете прочитать множество историй успеха компаний, использующих бессерверные системы, мы выбрали две из них, создавшие успешные продукты с небольшими командами. Мы считаем, что бессерверные вычисления позволят вам быстро двигаться вперед с привлечением небольшой команды разработчиков при относительно невысоких затратах на инфраструктуру, поэтому решили показать вам, как CodePen и MindMup делают это.

15.1. CodePen

CodePen (https://codepen.io) – популярное веб-приложение для создания, демонстрации и тестирования фрагментов HTML, CSS и JavaScript. Это онлайн-редактор кода и среда обучения с открытым исходным кодом, в которой разработчики могут создавать фрагменты кода (которые в терминологии CodePen называются «сочинениями» или «набросками»), тестировать их и делиться ими с другими разработчиками.

Разработчики CodePen уже выпустили видеоролик в своем видеоблоге, посвященном их бессерверным приложениям с Node и Claudia.js. Алекс Васкес (Alex Vazquez), один из основателей CodePen, сказал, что они широко используют AWS Lambda с Claudia для своих препроцессоров, поэтому мы связались с Алексом и попросили дать нам интервью. Далее приводится то, что мы узнали.

15.1.1. До перехода на бессерверные вычисления

Приложение CodePen позволяет разработчикам писать и компилировать HTML, CSS и JavaScript прямо в редакторе. Для этого оно должно иметь возможность отображать код разработчика; если разработчик использует препроцессоры (такие как SCSS, Sass, LESS для CSS или даже Babel для JavaScript), оно должно предварительно обработать его.

Поэтому исходная архитектура CodePen была основана на двух монолитных приложениях Ruby on Rails – главном веб-сайте и другом приложении, предназначенном для препроцессоров, – и одной, относительно небольшой службе баз данных.

Архитектура CodePen до перехода на использование бессерверных вычислений показана на рис. 15.1.

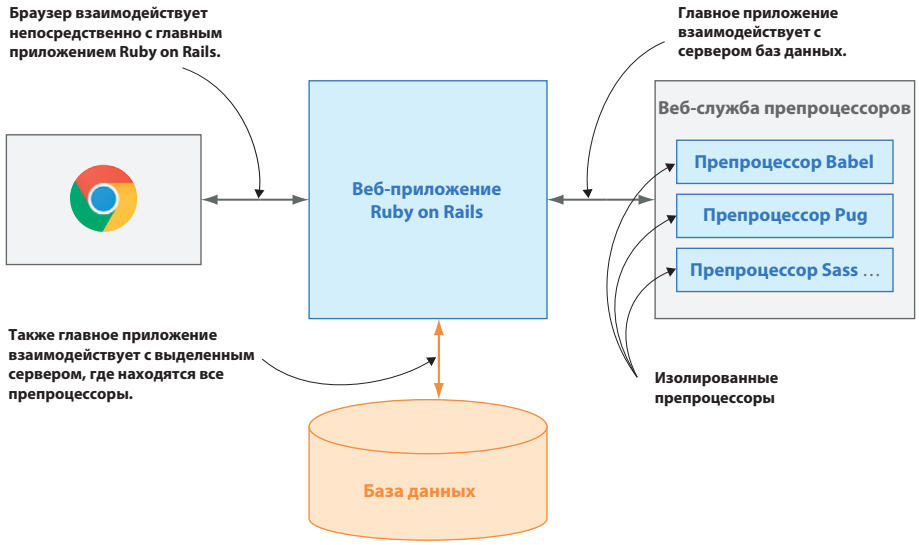


Рис. 15.1. Архитектура CodePen до перехода на использование бессерверных вычислений

Главная цель CodePen – дать пользователям возможность запускать свой код так часто, как они пожелают. Еще одна цель CodePen – позволить людям находить примеры кода на сайте, которые могли бы привести их в восхищение. Естественно, некоторые «сочинения» пользуются огромным интересом, который, как вы понимаете, невозможно предсказать. В таких ситуациях приложение CodePen должно быстро масштабироваться. Кроме того, большинство пользователей CodePen использует приложение бесплатно, поэтому ему нужен быстрый и недорогой способ гарантировать обслуживание всех пользователей.

Команда CodePen небольшая, ее члены разбросаны по всему миру, и у них есть всего один инженер, отвечающий за сопровождение и эксплуатацию. Поскольку CodePen выполняет код других людей, приложение должно обеспечить максимальную безопасность. По этой причине разработчики начали искать возможности отделить выполнение кода пользователя от приложения. Именно тогда они впервые услышали о AWS Lambda: их инженер сопровождения и эксплуатации – Тим Сабат (Tim Sabat) – предложил воспользоваться этой услугой как возможным решением. Первоначально разработчики CodePen отвергли эту идею, так как не видели в ней большого смысла, думая, что переход к ее использованию окажется хлопотным делом. У них уже были настроенные серверы, и им не особенно хотелось заниматься дополнительными проблемами отдельных служб Lambda.

Но однажды им потребовалось организовать форматирование кода по требованию с использованием нового инструмента, и тогда «идея использования Lambda» показалась им идеальным решением для этой задачи, особенно в совокупности с Claudia.js в качестве инструмента развертывания. Алекс сказал, что они многое узнали о возможностях, предлагаемых бессерверными функциями, таких как использование API Gateway для настройки полного HTTP API, подключение к S3 и настройка заданий `cron`. Внезапно до команды дошло, насколько мощными могут быть бессерверные приложения.

15.1.2. Миграция на бессерверные вычисления

Приложение CodePen использует много препроцессоров, каждый из которых предназначен для обработки определенного типа кода: HTML, CSS или JavaScript. Они действуют по-разному, имеют разные требования к вычислительной мощности и памяти и должны работать асинхронно. На рис. 15.2 вы можете увидеть, как работают препроцессоры.

Как видите, каждый препроцессор решает свою задачу, и работа одного может повлиять на работу другого. Поэтому, проанализировав текущую архитектуру приложения, разработчики поняли, что разделение препроцессоров позволит поднять не только производительность, но и безопасность. Когда все препроцессоры выполняются на одном же сервере, их очень сложно оптимизировать и выявлять ошибки. Они решили, что имеет смысл разделить их и запускать в отдельных функциях AWS Lambda, по аналогии с упомянутой выше услугой «форматирования кода по требованию».

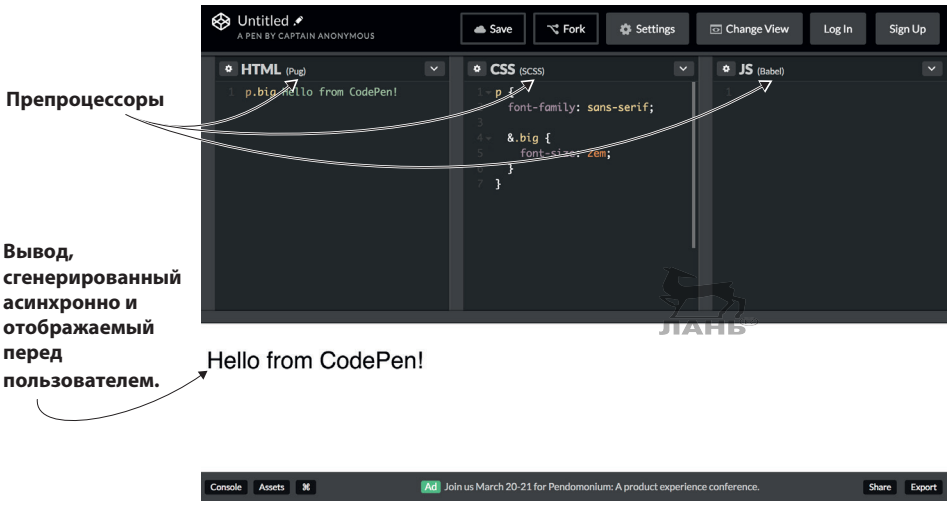


Рис. 15.2. Препроцессоры CodePen

Члены команды решили полностью реорганизовать свое монолитное приложение с препроцессорами, выделив его в отдельную бессерверную функцию – *маршрутизатор* – и создав множество отдельных бессерверных функций для препроцессоров. Задача маршрутизатора – вызывать необходимые функции препроцессоров, каждая из которых имеет целью предварительную обработку кода определенного типа.

Используя библиотеку Claudia, члены команды поняли, что с миграцией могут помочь даже их инженеры, занимающиеся разработкой пользовательского интерфейса, и тем самым снять нагрузку с единственного инженера сопровождения. Рэйчел Смит (Rachel Smith), ведущий разработчик CodePen, смогла в одиночку создать маршрутизатор в AWS Lambda, что позволило CodePen обслуживать более 200 000 запросов одновременно в пиковые периоды. В то же время Алекс и другие разработчики выполнили перенос отдельных препроцессоров. Получившаяся у них архитектура показана на рис. 15.3.

После перехода на бессерверные вычисления в CodePen по-прежнему оставалось основное монолитное приложение на Ruby on Rails, но теперь вместо монолитного приложения с препроцессорами имелась бессерверная функция маршрутизатора и около десятка бессерверных функций препроцессоров. Доступ к маршрутизатору осуществляется через API Gateway, и в нем используется AWS SDK для вызовов нужных бессерверных функций препроцессоров. В теле запроса маршрутизатор получает массив заданий. Например, одно из заданий требует использовать препроцессор Babel; в этом случае маршрутизатору посылается версия Babel по умолчанию, которую следует использовать. Маршрутизатор знает, какие версии Babel доступны, потому что каждая функция Lambda препроцессора имеет соответствующий номер версии. Дизайн соответствует шаблону проектирования «Команда», согласно которому передается команда для запуска, и маршрутизатор знает, какие функции Lambda он должен вызывать.

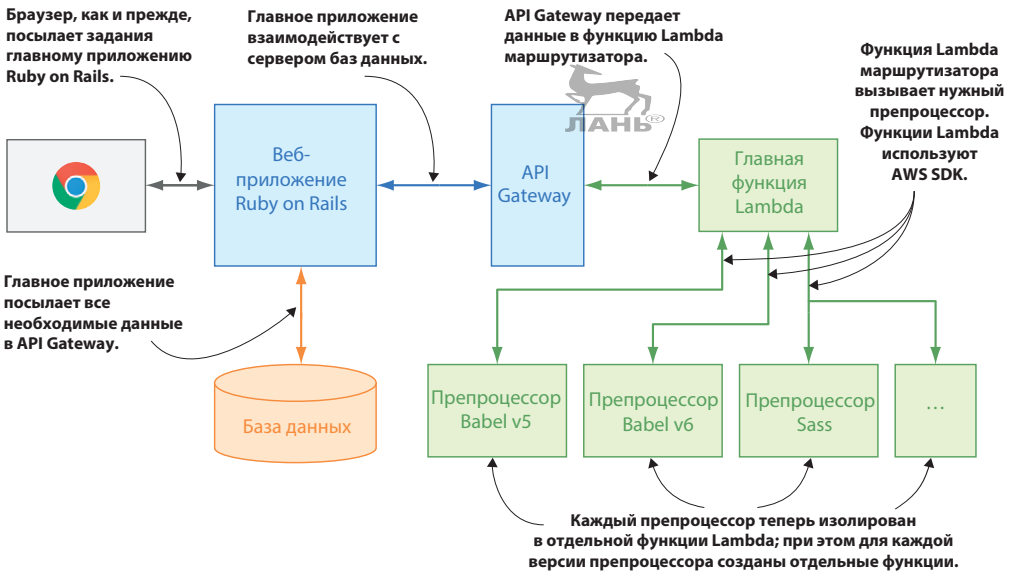


Рис. 15.3. Архитектура бессерверной версии CodePen

ПРИМЕЧАНИЕ. Шаблон проектирования «Команда» – это шаблон, предполагающий использование объекта для инкапсуляции всей информации, необходимой для выполнения некоторого действия. Это один из шаблонов проектирования, описанных в популярной книге «Design Patterns: Elements of Reusable Object-Oriented Software»[1]. Больше информации о шаблоне «Команда» можно найти здесь: https://ru.wikipedia.org/wiki/Команда_(шаблон_проектирования).

Тандем AWS Lambda и Claudia

Алекс отметил, что тандем AWS Lambda и Claudia оказался идеальным решением для CodePen. Большинство проблем разработки было устранено, и комбинация AWS Lambda/Claudia позволила им сделать намного больше с их существующими навыками. Раньше разработчикам пользовательского интерфейса приходилось ждать, пока инженер сопровождения напишет сценарии, настроит серверы и инструменты и автоматизирует все задачи, необходимые для развертывания. Благодаря AWS Lambda и Claudia разработчики пользовательского интерфейса теперь могут делать все сами. Основатель CodePen заявил, что уровень интеграции варьируется от «Мы любим Claudia и AWS Lambda» до «Мы жить не можем без Claudia и AWS Lambda». Это поднимает разработку на совершенно другой уровень.

[1] *Джонсон Ральф, Хелм Ричард, Влиссидес Джон, Гамма Эрих.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2019. ISBN: 978-5-4461-1213-5, 978-5-4590-1720-5, 978-5-469-01136-1, 978-5-496-00389-6. – *Прим. перев.*

Кроме высокой производительности и безопасности, разделение монолитного приложения с препроцессорами на бессерверные функции обеспечило простую возможность добавления в CodePen новых препроцессоров в виде бессерверных функций. Для разработчиков CodePen это самое большое преимущество: раньше все упиралось в «мини-монолит», как они называли свое прежнее приложение с препроцессорами на Ruby on Rails, и изменение или добавление новых возможностей было непростым делом.

15.1.3. Затраты на инфраструктуру

Затраты команды CodePen значительно уменьшились после перехода на бессерверные вычисления, поскольку отпала необходимость резервировать экземпляры заранее; теперь они платят только за фактически выполненную работу. Нагрузка меняется, поэтому в периоды невысокого трафика команде CodePen не приходится платить за простаивающие экземпляры. Кроме того, благодаря кешированию API Gateway удалось сэкономить еще больше денег. По сравнению с оплатой за зарезервированные серверы, AWS Lambda обходится очень дешево.

Еще одно важное соображение, которое большинство не учитывает, – количество времени, затрачиваемое на управление серверами, и внутренняя сложность. Чтобы увидеть разницу, в CodePen даже попробовали выделить всем своим функциям Lambda максимальный объем памяти (3 Гб). Они стали работать еще быстрее, потому что выполнялись на превосходном оборудовании (даже лучшем, чем требовалось). Ежемесячная плата выросла на 1000 долларов, что очень много, но даже на этом уровне она все еще была доступной и, по словам Алекса Васкеса, даже с дополнительными расходами на максимальную конфигурацию AWS Lambda, стоимость обслуживания не выдерживала сравнения с затратами на настройку и развертывание серверов.

Выбор правильного размера функции Lambda

В CodePen отсутствует стандарт для определения правильного размера бессерверной функции. Монолиты и функции (при увеличении размеров) делятся с использованием понятия *логической единицы*. Например, если потребуется реализовать операцию с изображением, будет создана бессерверная функция, включающая в себя все, что для этого необходимо.

Аналогично, для выполнения операций с аудиозаписями будет создана бессерверная функция, предназначенная для обработки аудиозаписей.

В CodePen логические единицы – это препроцессоры: Babel, Autoprefixer, Sass, LESS и др. Для каждого создана отдельная бессерверная функция Node.js. Это позволяет развернуть несколько версий одного и того же препроцессора. Разбивать функциональность еще дальше не имеет смысла, потому что более дробным делением сложнее управлять.

Кроме того, каждая бессерверная функция в CodePen имеет определенное распределение памяти. В общем случае для бессерверной функции выделяет-

ся 512 Мб памяти. Babel – единственный препроцессор, который запускается очень долго, поэтому для него выделено 1024 Мбайт.

15.1.4. Тестирование и проблемы

Основное тестирование CodePen осуществляется с применением модульных тестов, выполняющихся под управлением фреймворка Jest. Так как бессерверные функции в составе CodePen написаны на Node.js с Claudia, тесты для них писали те же инженеры, разрабатывавшие эти функции. Интеграционных тестов немного, но существует много тестов, используемых для отладки или тестирования всей системы вручную.

Увеличение степени масштабирования

По умолчанию AWS Lambda ограничивает степень масштабирования 1000 одновременно выполняемых экземпляров, но из-за непредсказуемости трафика приложение CodePen быстро достигло этого предела.

После отправки простого запроса-просьбы в AWS степень масштабирования для CodePen почти сразу была увеличена до 5000 одновременных запросов.

Холодные запуски

Асинхронный характер препроцессоров CodePen смягчает влияние холодных запусков на бессерверные функции. Это не влияет на их работу. CodePen также сильно зависит от кеша в API Gateway. Они создали уникальный URL для каждого определенного набора данных, потому что API Gateway может кешировать только уникальные URL.

Мониторинг

Для мониторинга своей бессерверной системы команда CodePen использует лишь CloudWatch. Для получения отчетов об ошибках они используют HoneyBadger и его Node.js SDK. Вся система находится под постоянным наблюдением, и всякий раз, когда возникает тайм-аут или ошибка, сообщение об этом посылается в HoneyBadger.

Безопасность

Для обеспечения безопасности в CodePen используется JSON Web Token (JWT). Монолит генерирует ключ и передает его клиенту, чтобы клиент мог легко аутентифицировать запросы, отправляемые им бессерверной функции маршрутизатора.

15.2. MindMup

MindMup (https://www.mindmup.com) – популярное веб-приложение для составления интеллект-карт (mind map), написанное преимущественно на JavaScript. По словам Гойко Адзича (Gojko Adzic), одного из основателей компании,

MindMup обслуживает почти полмиллиона активных пользователей, при этом сопровождение и развитие приложения осуществляет команда всего из двух человек. Такого успеха они смогли добиться благодаря широкому использованию бессерверных служб в AWS, что позволило им значительно сократить расходы. Что еще более важно, применение бессерверных вычислений подтолкнуло их к совершенствованию архитектуры и теперь позволяет им быстрее двигаться дальше и больше экспериментировать.

Интеллект-карты

Интеллект-карта (mind map) – это диаграмма с визуальным представлением информации. Она имеет иерархическую организацию и показывает отношения между частями целого. Интеллект-карта часто создается вокруг единого понятия, нарисованного в центре пустой страницы, к которому потом добавляются представления связанных идей, таких как изображения, слова и части слов. Основные идеи напрямую связаны с центральным понятием, а другие идеи вытекают из них. Узнать больше можно по адресу https://ru.wikipedia.org/wiki/Диаграма_связей.

15.2.1. До перехода на бессерверные вычисления

С момента основания в 2013 году проект MindMup был оптимизирован для разработки небольшой командой и использования относительно недорогой инфраструктуры. Вначале разработчики решили объединить Heroku с AWS, чтобы получить масштабируемую инфраструктуру, требующую минимального обслуживания.

Heroku

Heroku – это облачная платформа как услуга (Platform as a Service, PaaS), поддерживающая несколько языков программирования, на которой развертываются веб-приложения. Она появилась в июне 2007 года и стала одной из первых облачных платформ. Тогда Heroku поддерживала только язык программирования Ruby, но теперь она поддерживает Java, Node.js, Scala, Clojure, Python, PHP и Golang. Как многоязычная платформа, Heroku позволяет разработчикам одинаковым образом создавать, запускать и масштабировать приложения на всех языках.

Узнать больше о Heroku можно на сайте проекта https://www.heroku.com.

В MindMup имелось одно приложение на Heroku, которое было ядром системы; оно обслуживало одностраничное веб-приложение и API. API отвечал за аутентификацию, авторизацию, предоставление данных и подписку. Проект

MindMup начинался как бесплатная служба, но спустя почти два года создатели добавили платную версию с поддержкой совместной работы. Для поддержки платных пользователей им потребовалась простая и масштабируемая база данных, поэтому они добавили AWS DynamoDB.

Помимо одностраничного приложения для создания интеллект-карт, одной из наиболее важных частей MindMup являются *экспортеры* – инструменты, позволяющие экспортировать интеллект-карты в различные форматы, такие как PDF, Word и даже презентации PowerPoint. Для этой цели применялось большое число конвертеров, и каждый отличался своими особенностями использования и потребления памяти. Например, экспортер в формат PDF использовался все время и требовал много вычислительной мощности и памяти. Экспортер в простой текстовый формат был намного менее требовательным к ресурсам, а экспортер в формат Markdown использовался очень редко. Кроме того, разные экспортеры требовали наличия в системе разных приложений:

- для преобразования в формат PDF требовалось наличие Ghostscript – программного пакета для работы с форматом PDF;
- для преобразования в формат Word использовалась Apache POI – библиотека на Java для чтения и записи документов в формате Microsoft Office;
- для преобразования в другие форматы требовались языки программирования Ruby и Python.

Для преобразования интеллект-карт в разные форматы использовалось хранилище файлов AWS S3. Но вместо выгрузки файлов из API была реализована выгрузка из браузера непосредственно в корзину AWS S3, с использованием подписанных URL. Уведомления о вновь выгруженных файлах помещались в очередь AWS Simple Queue Service (SQS). Для каждого типа файла была организована своя очередь SQS и запускалось несколько приложений, читающих задания из очереди и выполняющих преобразование в указанный формат.

Приложения-конвертеры выполнялись на платформе Heroku, но для поддержки большого количества независимых экспортеров требовалось почти 30 динов[2] (легких контейнеров на основе Linux), что было слишком дорого. Чтобы снизить стоимость, они объединили экспортеров в несколько приложений Heroku, после чего каждое приложение стало представлять собой группу экспортеров, которым требовался один и тот же язык программирования, потому что дины в Heroku требуют выбора языка программирования при настройке. На тот момент приложение MindMup имело архитектуру, изображенную на рис. 15.4.

Объединение экспортеров уменьшило затраты на инфраструктуру, но породило некоторые другие проблемы:

- ухудшилась изоляция и выросло число конфликтов между библиотеками;

[2] Название dino переводится на русский язык как «динозавр» или «динозаврик». – *Прим. перев.*

- добавление нового экспортера требовало значительных усилий по координации, потому что необходимо было создать новое приложение, организовать новую очередь SQS. Сами очереди SQS были довольно легковесными, но они добавляли дополнительный уровень сложности;
- так как экспортеры объединялись в большие пакеты, проводить эксперименты было трудно;
- было сложно обновлять пакеты для разных приложений Heroku с разными языками программирования.

Кроме того, случился скандал с механизмом маршрутизации запросов в Heroku – запросы передавались случайному дино, а не свободному, даже если случайно выбранный дино был уже занят. Узнать больше об этом инциденте можно по адресу https://genius.com/James-somers-herokus-ugly-secret-annotated.

На тот момент экспортеры были главной проблемой архитектуры MindMup.

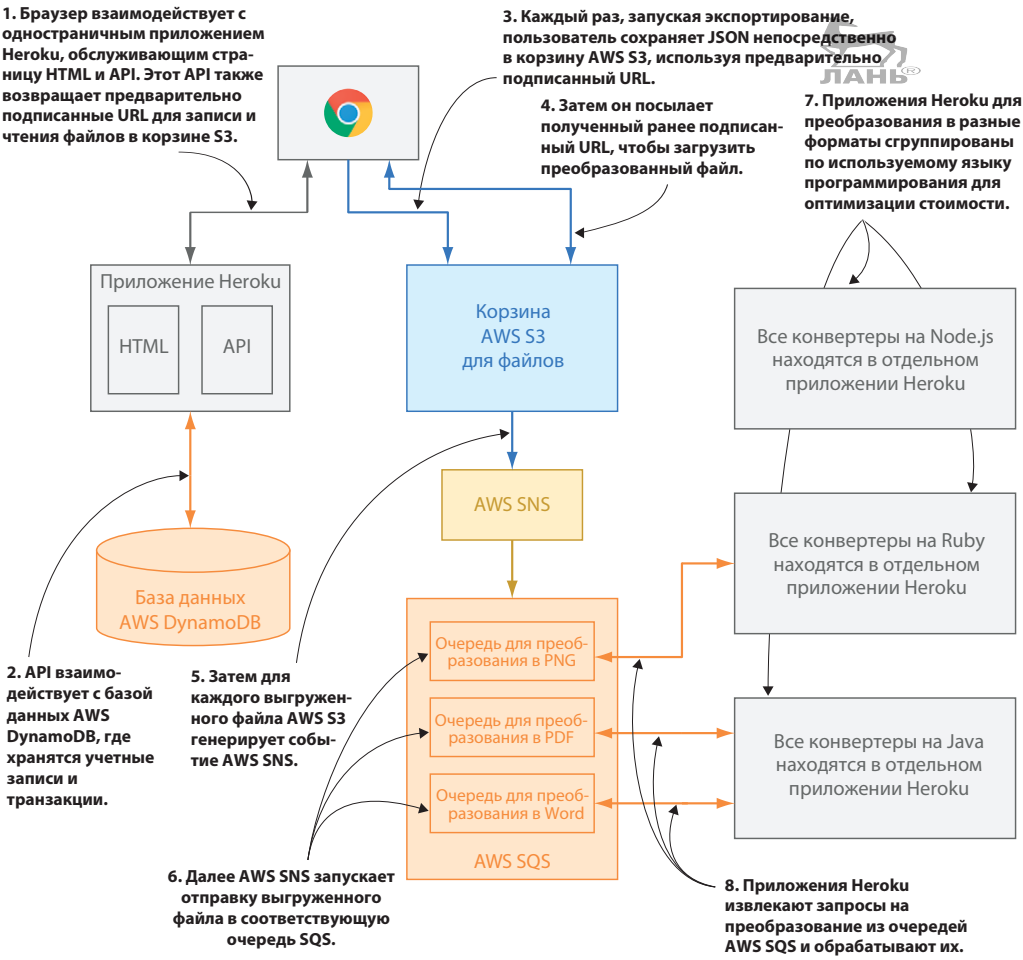


Рис. 15.4. Архитектура MindMup перед миграцией на бессерверные вычисления

15.2.2. Миграция на бессерверные вычисления

В январе 2016 года разработчики планировали добавить нового экспортера и, изучая возможности AWS Lambda, заметили, что сохранение файлов в S3 может вызывать функции Lambda. Для эксперимента Гойко написал экспортера в Node.js и сценарий на bash для автоматизации развертывания новой функции Lambda. Им понравилось, что из этого получилось: в отличие от Heroku, которая требовала резервирования вычислительных ресурсов заранее, AWS Lambda предлагала платить только за время использования. Это означало, что для каждого экспортера можно написать отдельную функцию. Воодушевленные результатами эксперимента, они решили начать постепенный перенос всех экспортеров в AWS Lambda.

Гойко заметил, что для реализации первого экспортера ему потребовалось написать 30 строк кода в Node.js и более 200 строк в сценарии bash. Он осознал, что риск сместился с кода на развертывание и что ему нужен проверенный инструмент для миграции в бессерверное окружение. Однако экосистема бессерверных инструментов была недостаточно развита, поэтому он решил реализовать свое решение, которое позже было открыто и опубликовано как Claudia.js.

Поскольку веб-сайт MindMup был монолитом, который обслуживал API и выполнял отображение на стороне сервера, разработчики решили начать миграцию в AWS Lambda с перевода службы API. Миграцию API можно было осуществить через API Gateway, но они воспользовались удобной возможностью переписать и реорганизовать часть старого кода и привести в порядок трехлетнюю базу кода, которая, по словам Гойко, развивалась «порой самыми неожиданными путями».

В процессе миграции они создали в Heroku одно приложение, отвечавшее за отображение файла `index.html`. Заменили прежнее приложение статическим сайтом в корзине AWS S3 с AWS CloudFront в качестве слоя CDN и кеширования. Единственным экспортером, который остался на Heroku, был экспортер в формат PowerPoint, написанный на Java. Некоторое время он работал нормально, но в конце концов его тоже перенесли в AWS Lambda.

Вся миграция заняла около года неторопливой работы, и в феврале 2017 года MindMup превратился в полностью бессерверное приложение. Весь код распределился по небольшим функциям Lambda, организованным в логические единицы. В текущей архитектуре веб-сайт MindMup загружается из AWS S3 и CloudFront. Браузер напрямую взаимодействует с так называемым *Gold API*, который является функцией Lambda за API Gateway. Gold API подключен к базе данных DynamoDB и использует платежными системами Stripe и PayPal.

Gold API также генерирует предварительно подписанные URL, которые позволяют браузеру выгружать файлы непосредственно в корзину S3 и проверять ее в поисках результатов преобразования. Когда файл выгружается в корзину S3, она посылает уведомление SNS, которое, в свою очередь, запускает конвертеры.

Самое большое отличие новых экспортеров состоит в том, что теперь каждый из них – независимо от количества обрабатываемых им запросов – находится в отдельной функции Lambda.

> **СОВЕТ.** Реализация каждого экспортера в виде отдельной функции Lambda также позволила MindMup оптимизировать затраты: некоторые экспортеры находятся в функциях Lambda с минимальным объемом памяти (128 Мб), а некоторые – в функциях с большим объемом памяти.

Закончив преобразование, экспортер выгружает результат непосредственно в корзину S3, а браузер загружает его к себе.

Еще больше функций Lambda используется для аналитических целей и вызывается различными компонентами системы (например, API или одним из экспортеров) посредством уведомлений SNS. Эти функции Lambda обрабатывают данные и сохраняют их в нескольких разных службах, таких как корзины S3 и таблицы DynamoDB.

Кроме API, имеется компонент авторизации, который взаимодействует со статическим веб-сайтом и функцией Lambda в Gold API. Для поддержки учетных записей Google в MindMup используется Cognito, но в приложении есть также собственный компонент авторизации, используемый остальной частью приложения.

В настоящий момент приложение MindMup имеет архитектуру, изображенную на рис. 15.5.

Несмотря на то что теперь MindMup является полностью бессерверным приложением, команда постоянно работает над улучшениями. Например, в ближайшее время они предполагают добавить масштабируемую поддержку совместной работы в реальном времени с использованием потоков Kinesis и функций AWS Lambda.

15.2.3. Затраты на инфраструктуру

Одним из самых удивительных результатов, полученных проектом MindMup от миграции на бессерверные вычисления, стала оптимизация расходов на инфраструктуру. Сравнение затрат и количества пользователей за декабрь 2016 года и декабрь 2015 года показало, что число пользователей выросло примерно на 50 %, тогда как затраты снизились примерно на 50 %. Воодушевившись такими показателями, разработчики решили полностью отказаться от Heroku и SQS; они перенесли все компоненты в AWS, сократили объем передаваемых данных и время ожидания. Пользуясь Heroku, они платили за зарезервированные мощности. Ситуация изменилась, когда они перешли на бессерверные вычисления. Gold API был единственной точкой доступа к остальным компонентам приложения, и это было их главным узким местом. Превратившись в бессерверную функцию, Gold API стал масштабироваться автоматически и перестал быть узким местом.

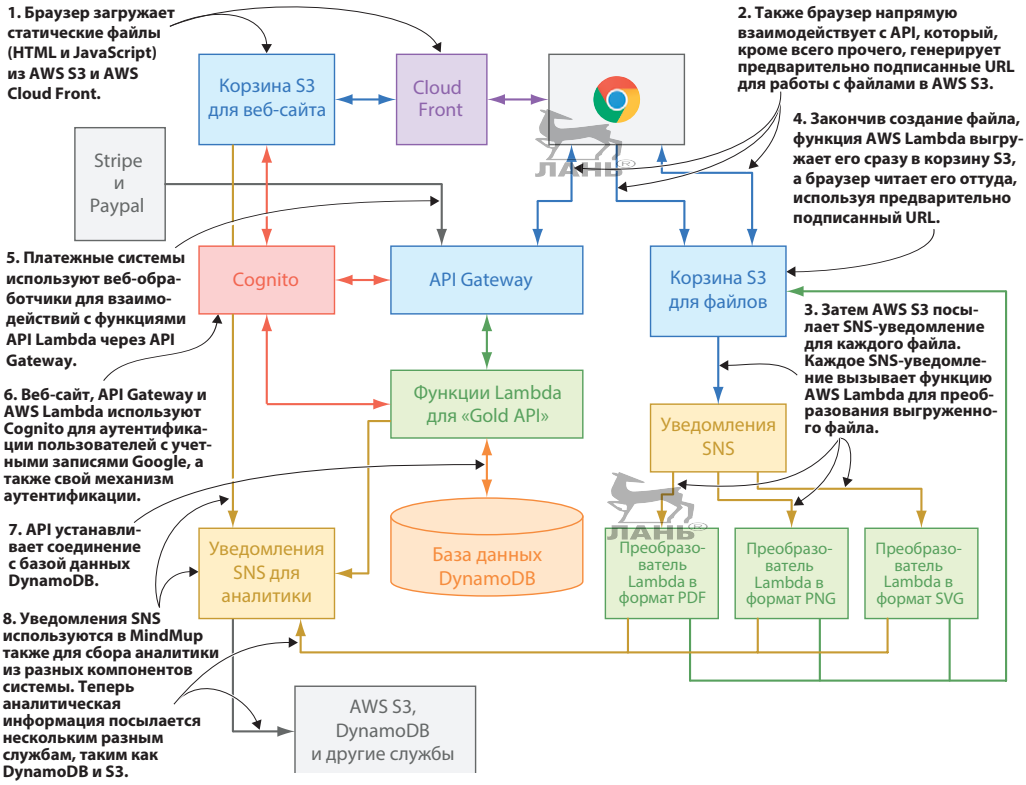


Рис. 15.5. Архитектура бессерверной версии MindMup

В сентябре 2017 года проект MindMup имел около 400 000 активных пользователей. Их ежемесячный счет за услуги AWS составил 102.92 долл. США. В табл. 15.1 показано, как эта сумма складывается из платежей за разные услуги.

Таблица 15.1. Приложения

Ресурс AWS	Ежемесячный платеж
Lambda	$0.53
API Gateway	$16.41
DynamoDB	$0
CloudFront	$65.20
S3	$5.86
Передача данных	$4.27

Им удалось снизить стоимость, потому что они позволили пользовательскому интерфейсу напрямую взаимодействовать с некоторыми службами, такими как S3, минуя API между браузером и S3. Это влияет на стоимость, поскольку разные службы AWS имеют разные модели ценообразования. Например:

- AWS взимает плату за количество запросов, продолжительность выполнения и потребление памяти;
- API Gateway взимает плату за количество запросов и передачу данных;
- Amazon S3 взимает плату только за передачу данных.

В традиционном приложении выгрузка изображений происходит через API, вследствие чего вы платите за передачу данных в S3 и API Gateway, а также на количество запросов, продолжительность их обработки и потребление памяти в AWS Lambda. В бессерверной версии можно получить предварительно подписанный URL для S3 из API, что занимает меньше 100 мс и почти не требует памяти. Затем вы можете выгрузить файл напрямую в корзину S3 из пользовательского интерфейса и оплатить только эту передачу данных.

15.2.4. Тестирование, журналирование и проблемы

Зная, что библиотека Claudia хорошо охвачена автоматизированными тестами, было интересно услышать от Гойко, как они тестируют MindMup. Как и ожидалось, MindMup имеет множество модульных и интеграционных тестов. Все приложение разделено на библиотеки, и в нем широко используется гексагональная архитектура.

Тесты написаны с использованием фреймворка Jasmine. Для каждой функции Lambda имеется множество модульных и несколько интеграционных тестов, если необходимо. Для большинства функций Lambda имеется файл `lambda.js`, который почти ничего не делает и отвечает только за подключение других компонентов. Эти файлы `lambda.js` почти не охвачены тестами, потому что содержат всего по три-четыре строки кода. Также для каждой функции имеется файл `main.js`, являющийся основным файлом данной функции Lambda; он получает событие из `lambda.js` и обрабатывает его. Файл `main.js` подключает различные библиотеки (например, `FileRepository`), перечисленные в файле `lambda.js`.

Наибольший объем имеют модульные тесты для файла `main.js`. Имеются также интеграционные тесты, которые подключают `main.js` к `MemoryRepository`. Библиотека `FileRepository` тестируется модульными и интеграционными тестами, но отдельно, потому она стоит особняком.

Ход тестирования типичной функции AWS Lambda изображен на рис. 15.6.

Для некоторых экспортеров также выполняются визуальные тесты с использованием Appraise, инструмента для визуального тестирования, о котором мы упоминали в главе 11. Узнать больше об Appraise можно на странице проекта http://appraise.qa/.

Для журналов в MindMup используется CloudWatch. С его помощью также отслеживаются ошибки, информация об оплате и доступе; например, исключения в пользовательском интерфейсе отслеживаются через Google Analytics. Некоторые события сохраняются в S3, чтобы потом их можно было отыскать. Приложение не нуждается в журналировании в реальном времени с широкими возможностями поиска.

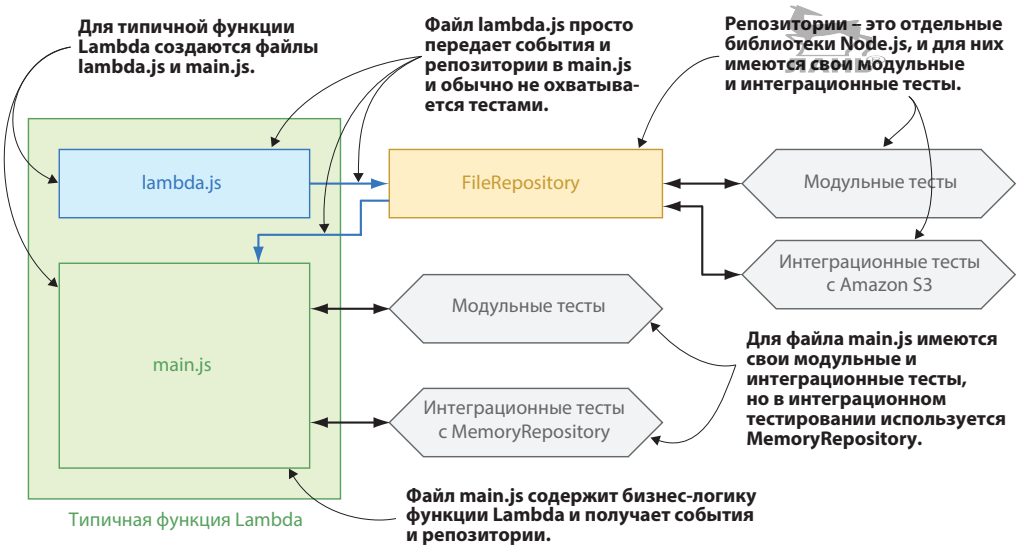


Рис. 15.6. Процесс тестирования типичной функции AWS Lambda

Важно отметить, что MindMup использует метки для разных окружений, что немного противоречит рекомендациям AWS. Метки используются в MindMup для маркировки функций в стадии разработки; эти функции взаимодействуют с корзиной S3 для разработки, таблицей в DynamoDB и т. д. Чтобы перенести функцию из окружения разработки в окружение эксплуатации, достаточно лишь изменить метку. Безопаснее было бы иметь разные учетные записи для разных окружений, но небольшой размер команды делает описанный подход вполне приемлемым, а его использование позволяет развивать MindMup быстрее.

В заключение

- Бессерверная архитектура позволяет быстро создать масштабируемый продукт с помощью небольшой команды и недорогой инфраструктуры.
- Тандем Claudia и AWS Lambda дает возможность разрабатывать и развертывать готовые к работе службы без обширных знаний внутренних особенностей.
- Переход на бессерверные вычисления может дать значительное снижение затрат, если правильно оптимизировать приложение с учетом всех особенностей платформы.
- В бессерверном окружении риски смещаются в область развертывания и интеграции, поэтому важно покрыть эти операции тестами.
- Использование гексагональной архитектуры уменьшает сложность кода и упрощает тестирование бессерверного приложения.

- Переход на бессерверные вычисления можно использовать для полного рефакторинга имеющегося приложения.

Помимо всего того, что вы узнали о преимуществах бессерверных вычислений в этой книге, мы надеемся, что вам понравилась компания тетушки Марии и вы не слишком проголодались, читая о ней. Приятного вам аппетита!

Приложение A

Установка и настройка

В этом приложении подробно описывается, как установить и настроить Claudia и другие библиотеки экосистемы: Claudia API Builder и Claudia Bot Builder.

В данном приложении, как и во всей книге, предполагается, что вы знакомы с основами платформы AWS и имеете учетную запись. Если это не так, настоятельно рекомендуем создать учетную запись и познакомиться с AWS в целом и системой управления учетными записями и разрешениями в частности, прежде чем читать дальше. Создать учетную запись можно на веб-сайте AWS: https://aws.amazon.com. Желающим лучше понять суть учетных записей и ролей рекомендуем обратиться к официальной документации: http://docs.aws.amazon.com/IAM/latest/UserGuide/id.html.

A.1. Установка Claudia

Claudia – это обычный модуль для Node.js, доступный в NPM.

Чтобы установить Claudia и получить доступ к команде `claudia` в терминале, выполните команду

```
npm install claudia -g
```

Другая возможность: установить Claudia в проект Node.js как зависимость времени разработки с помощью следующей команды:

```
npm install claudia --save-dev
```

В этом случае Claudia будет установлена локально, то есть вы не сможете использовать ее в терминале. Вместо этого вы вынуждены будете запускать сценарий NPM. В листинге A.1 показана минимальная версия файла `package.json`, который добавляется в проект при установке библиотеки Claudia в виде зависимости времени разработки, с необходимыми сценариями NPM.

Листинг A.1. Пример файла `package.json` с локальной версией Claudia

```
{
  "name": "pizza-api",
  "version": "1.0.0",
  "description": "",
```

```
  "main": "api.js",
  "scripts": {
    "create": "claudia create --region eu-central-1 --api-module api",
    "update": "claudia update"
  },
  "keywords": [],
  "license": "MIT",
  "devDependencies": {
    "claudia": "^4.0.0"
  },
}
```

Сценарий NPM для создания API в регионе eu-central-1.

Сценарий NPM для обновления API.

Claudia сохраняется как зависимость времени разработки.

После добавления файла `package.json` с содержимым из листинга А.1 создать функцию Lambda и определение API Gateway можно командой `npm run create`, которая должна выполняться в терминале, в папке проекта, а обновить – командой `npm run update`.

Полное описание процедуры установки Claudia можно также найти на веб-сайте Claudia: https://claudiajs.com/tutorials/installing.html.

А.1.1. Настройка зависимостей Claudia

При всей простоте установки библиотека Claudia имеет одну важную зависимость: ключи доступа к профилю AWS.

Если вы еще не создали профиль AWS, обращайтесь к следующему разделу.

Для работы с AWS библиотека Claudia использует пакет AWS SDK для Node.js, а этот пакет SDK, в свою очередь, требует передать ему ключи для доступа к профилю AWS. Передать ключи можно несколькими способами. Самый простой – создать папку `.aws` в домашнем каталоге пользователя в операционной системе и внутри этой папки создать файл `credentials` без расширения и со следующим содержимым:

```
[default]
aws_access_key_id=YOUR_ACCESS_KEY
aws_secret_access_key=YOUR_ACCESS_SECRET
```

Имя профиля в AWS.

Открытый ключ доступа к профилю AWS.

Закрытый ключ доступа к профилю AWS.

ПРИМЕЧАНИЕ. Не забудьте заменить `YOUR_ACCESS_KEY` и `YOUR_ACCESS_SECRET` фактическими значениями ключей.

Если вы присвоили своему профилю имя, отличное от имени по умолчанию, то должны передать это имя в Claudia. Сделать это можно, передав флаг `--profile` с именем профиля (например, `claudia update --profile yourProfileName`) или определив переменную среды `AWS_PROFILE` (например, `AWS_PROFILE = yourProfileName; update claudia`).

Исчерпывающее руководство по настройке AWS SDK для Node.js находится по адресу http://docs.aws .amazon.com/sdk-for-javascript/v2/developer-guide/configuring-the-jssdk.html.

A.1.2. Создание профиля AWS и получение ключей

Чтобы создать новый профиль AWS для Claudia, откройте веб-консоль AWS (https://console.aws.amazon.com) и зарегистрируйтесь.

Затем перейдите на вкладку **Users** (Пользователи) в разделе **IAM** (https://console.aws.amazon.com/iam/home#/users). Щелкните на кнопке **Add User** (Добавить пользователя), как показано на рис. A.1.

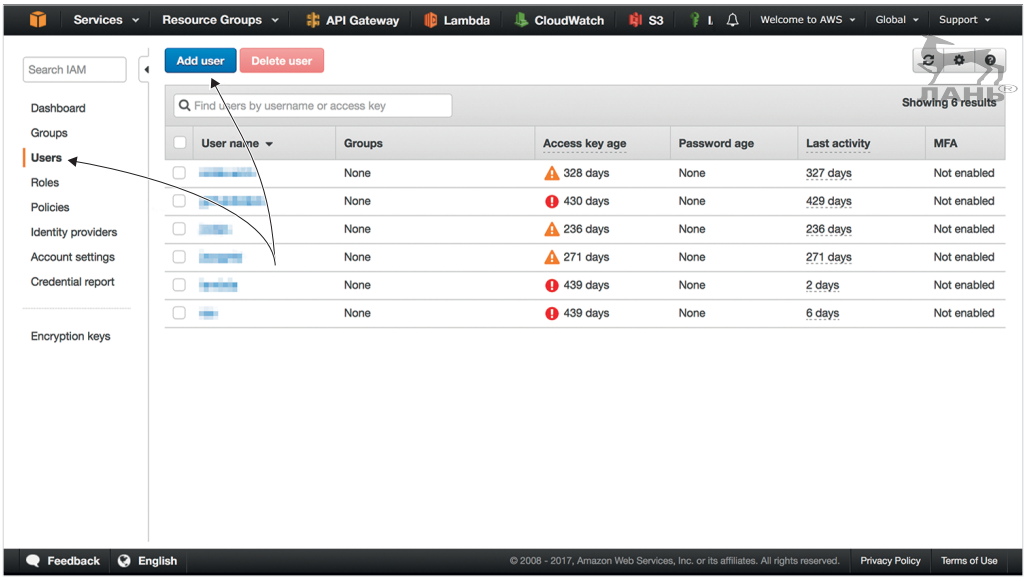


Рис. A.1. Вкладка **Users** (Пользователи) в разделе IAM в веб-консоли AWS

Чтобы добавить нового пользователя, нужно выполнить процедуру, состоящую из четырех шагов. Прежде всего выберите имя пользователя (для первого вашего пользователя прекрасно подойдет имя *claudia*) и определите для него тип доступа. Так как пользователь будет применяться только в AWS CLI и AWS SDK для Node.js, выберите вариант **Programmatic Access** (Программный доступ). Щелкните на кнопке **Next: Permissions** (Далее: Разрешения), см. рис. A.2.

На втором шаге необходимо определить разрешения для пользователя. Выберите вкладку **Attach Existing Policies Directly** (Добавить существующие политики непосредственно), как показано на рис. A.3. Затем используйте поле ввода для поиска политик для добавления.

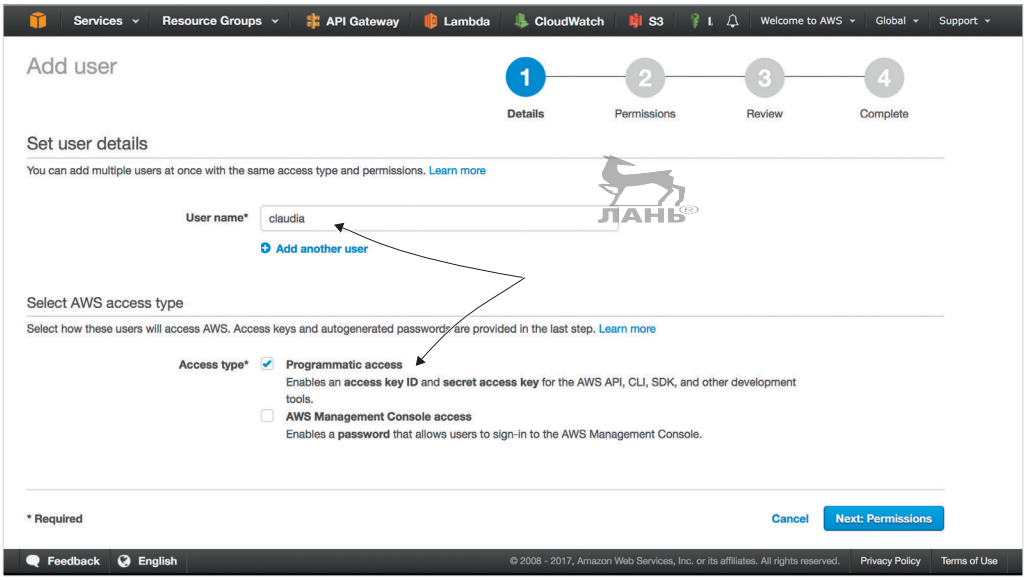


Рис. А.2. Первый шаг в процедуре добавления пользователя AWS: настройка свойств пользователя

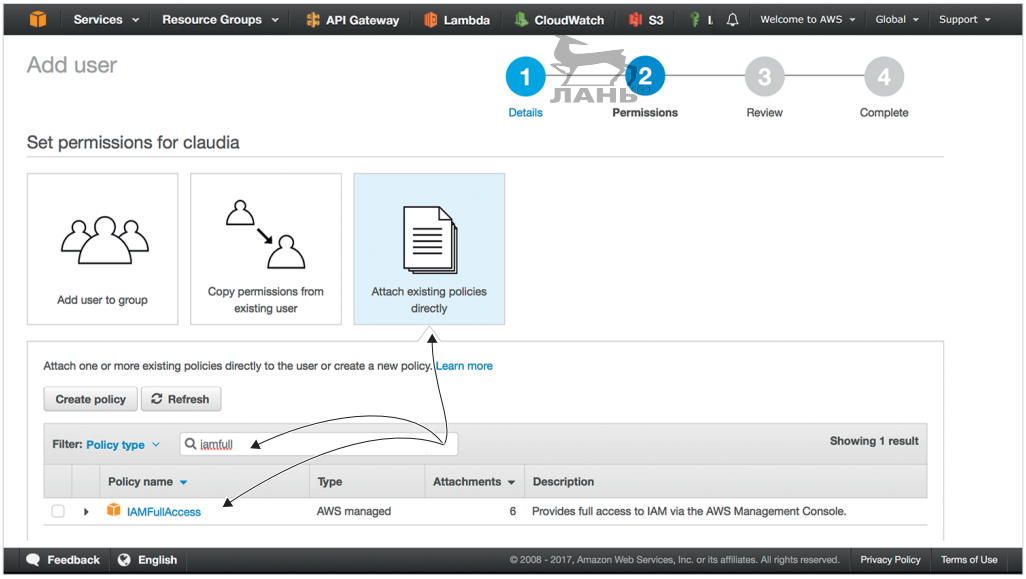


Рис. А.3. Второй шаг в процедуре добавления пользователя AWS: настройка разрешений

Для опробования примеров в этой книге нужно добавить следующие политики:

- *IAMFullAccess* – необходима, чтобы Claudia могла автоматически создавать роли выполнения для функций Lambda (рекомендуется для на-

чинающих). Альтернативный вариант заключается в том, чтобы передавать имя существующей роли команде `claudia create` в параметре `--role`;

- *AWSLambdaFullAccess* – нужна для развертывания Claudia;
- *AmazonAPIGatewayAdministrator* – необходима для работы Claudia API Builder и Claudia Bot Builder;
- *AmazonDynamoDBFullAccess* – нужна для управления базой данных DynamoDB;
- *AmazonAPIGatewayPushToCloudWatchLogs* – необязательна; используется для журналирования запросов и ответов в API Gateway.

В промышленном окружении к выбору ролей нужно подходить особенно внимательно; обсуждение этого вопроса выходит далеко за рамки книги, но мы советуем тщательно изучить все, что связано с ролями и политиками AWS, прежде чем приступать к развертыванию приложений в промышленном окружении.

Третий шаг – обзор выполненных настроек, как показано на рис. А.4. Если все настройки содержат правильные значения, щелкните по кнопке **Create User** (Создать пользователя) внизу страницы.

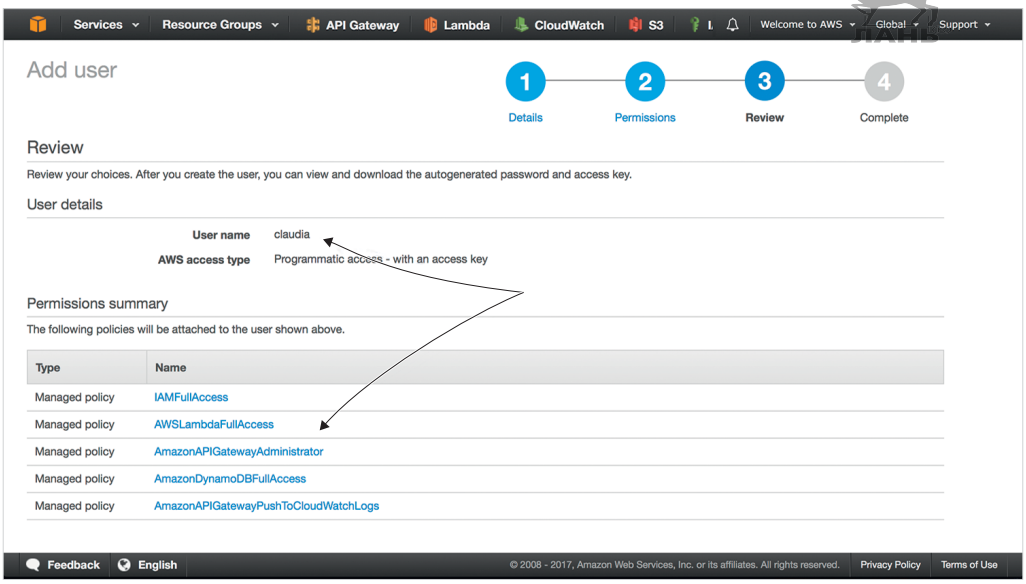


Рис. А.4. Третий шаг в процедуре добавления пользователя AWS: обзор

В ответ вам будет предложено подтвердить свое решение, как показано на рис. А.5. Этот последний шаг очень важен, потому что именно здесь дается доступ к открытому и закрытому ключам доступа.

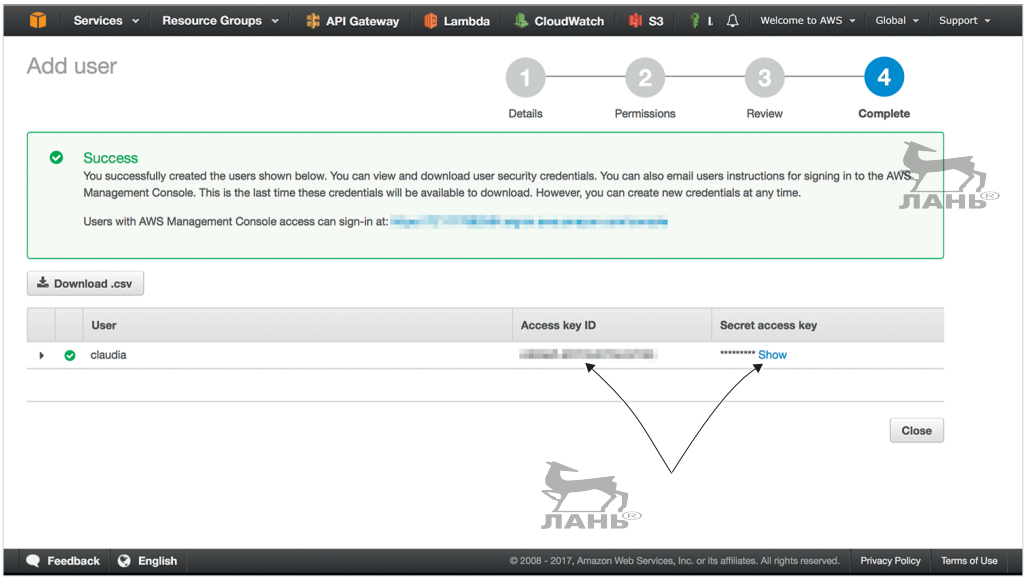


Рис. А.5. Получение ключей для пользователя

Теперь, получив ключи, можете вернуться в предыдущий раздел и подставить их.

А.1.3. Установка Claudia API Builder

Библиотека Claudia API Builder доступна как пакет NPM. Она не требует настройки, поэтому для установки достаточно сохранить ее как зависимость в проекте Node.js, выполнив команду

```
npm install claudia-api-builder --save
```

В примерах в этой книге использовалась версия 4.0.0.

А.1.4. Установка Claudia Bot Builder

Так же как и API Builder, библиотека Claudia Bot Builder доступна в виде обычного пакета NPM и не требует специальной настройки. Для установки достаточно сохранить ее как зависимость в проекте Node.js, выполнив команду

```
npm install claudia-bot-builder --save
```

В примерах в этой книге использовалась версия 4.0.0.

А.2. Установка AWS CLI

Интерфейс командной строки AWS (AWS Command Line Interface, CLI) – это универсальный инструмент для управления службами AWS. В этой книге

инструмент AWS CLI использовался для решения самых разных задач, включая создание ролей и разрешений, а также для доступа к таблицам DynamoDB.

Чтобы установить AWS CLI в Windows, перейдите на страницу https://aws.amazon.com/cli/ и загрузите установочный пакет для Windows.

Если вы пользуетесь Mac или Linux, вам понадобится Python версии 2.6.5 или выше с диспетчером пакетов `pip`. Установка AWS CLI с помощью этого диспетчера выполняется командой


```
pip install awscli
```

Дополнительную информацию о AWS CLI можно найти на странице https://aws.amazon.com/cli/.

Чтобы убедиться, что клиент командной строки установился без ошибок, выполните команду `aws --version`.

В примерах в этой книге использовалась версия

```
aws-cli/1.11.138 Python/2.7.10 Darwin/16.7.0 botocore/1.6.5
```

Приложение B

Настройка Facebook Messenger, Twilio и Alexa

В этом приложении описывается порядок настройки следующих компонентов, используемых в главах 8, 9 и 10:

- страница и приложение Facebook Messenger;
- учетная запись Twilio;
- учетная запись Amazon Alexa.

ПРИМЕЧАНИЕ. Все перечисленные службы продолжают активно развиваться, и в какой-то момент пользовательский интерфейс или даже некоторые этапы могут измениться. Если пользовательский интерфейс, который вы видите, отличается от скриншотов в этом приложении, загляните в официальную документацию для данной службы. Ссылки приведены в тексте.

B.1. Настройка Facebook Messenger

Для настройки чат-бота для Facebook Messenger в главах 8 и 9 требуется выполнить следующие шаги:

1) создать страницу Facebook;
2) создать приложение Facebook;
3) создать чат-бота для Facebook Messenger с помощью Claudia Bot Builder;
4) включить обработку естественного языка (Natural language processing, NLP).

B.1.1. Создание страницы Facebook

Чтобы создать страницу Facebook, перейдите на страницу https://www.facebook.com/pages/create/. Здесь вы увидите список категорий, как показано на рис. B.1. Выберите тип нужной вам страницы.

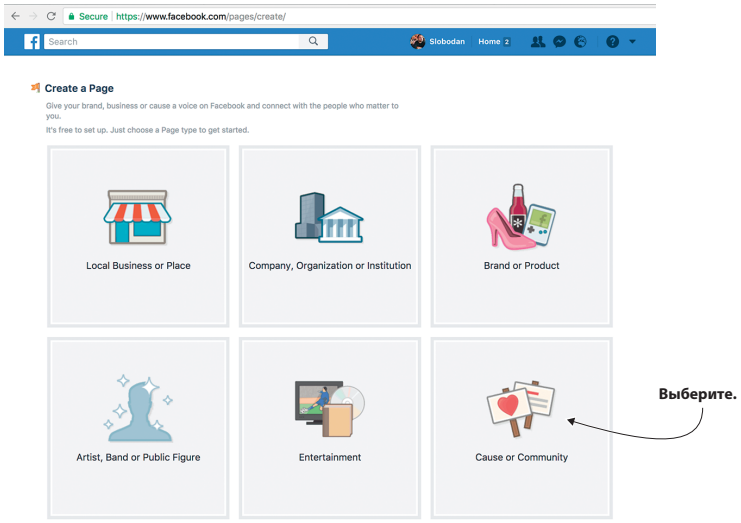


Рис. B.1. Создание страницы Facebook

ПРИМЕЧАНИЕ. Если скриншоты, показанные здесь, не совпадают с тем, что вы видите, обращайтесь к справочной статье, где описывается создание страниц, доступной по адресу https://www.facebook.com/business/help/104002523024878.

Вы можете выбрать любую категорию; мы выбрали **Cause or Community** (Общая идея или сообщество), потому что эта категория требует меньше всего настроек. После выбора **Cause or Community** (Общая идея или сообщество) вам будет предложено ввести имя страницы. Назовите страницу **Aunt Maria's pizzeria**, как это сделали мы (см. рис. B.2), и щелкните по кнопке **Get Started** (Приступить).

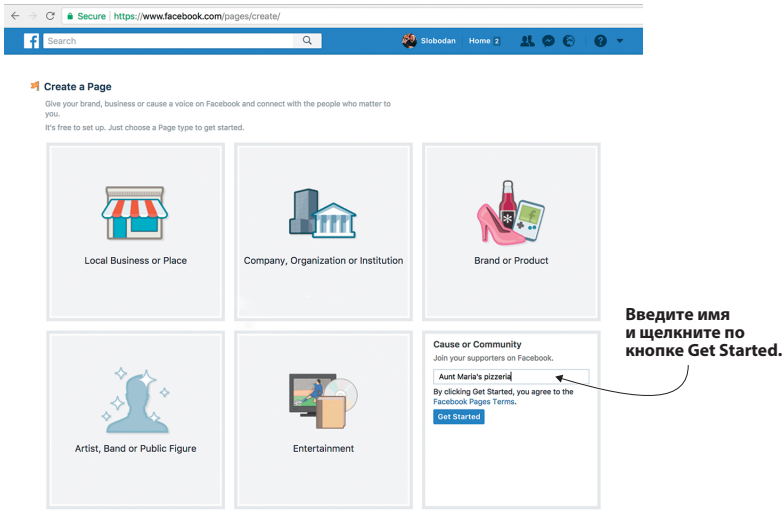


Рис. B.2. Выберите категорию и имя страницы

После ввода имени страницы Facebook предложит выгрузить профиль и титульное изображение, а также заполнить несколько дополнительных полей. Выполнив эти шаги (или пропустив их), вы получите страницу, как показано на рис. B.3.

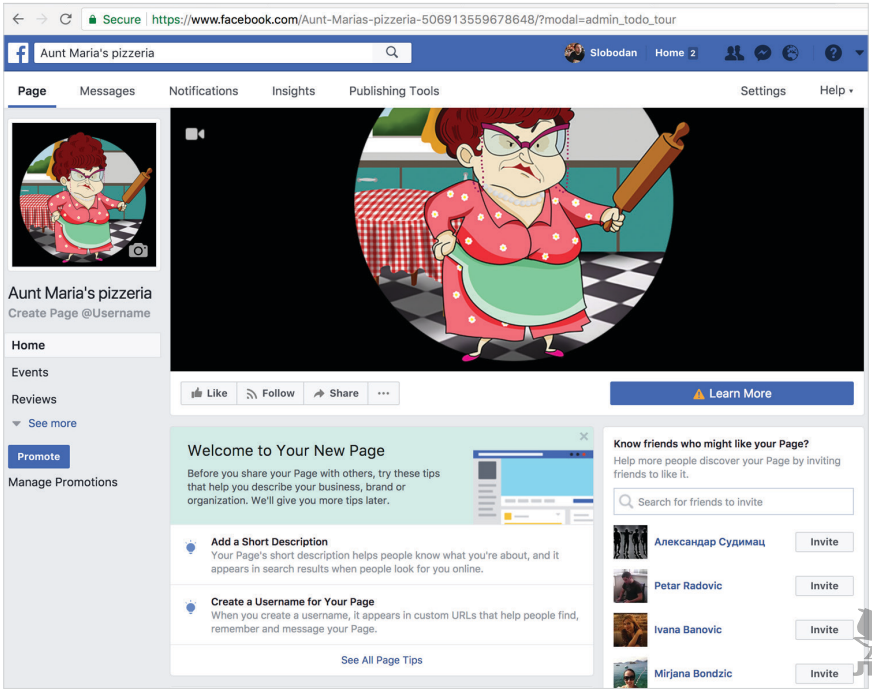


Рис. B.3. Страница Facebook для пиццерии тетушки Марии

B.1.2. Создание приложения Facebook

Следующий шаг – создание приложения Facebook. Для этого перейдите на страницу https://developers.facebook.com и в меню **My Apps** (Мои приложения) выберите пункт **Add a New App** (Добавить новое приложение), как показано на рис. B.4.

> **ПРИМЕЧАНИЕ.** Если скриншоты, показанные здесь, не совпадают с тем, что вы видите, обращайтесь к справочной статье, где описывается создание приложений, доступной по адресу https://developers.facebook.com/docs/apps/register.

Появится диалог **Create a New App ID** (Создать идентификатор нового приложения), как показано на рис. B.5, где вы должны ввести имя приложения и ваш адрес электронной почты. Заполните форму (используйте имя приложения «Aunt Maria's pizzeria») и щелкните на кнопке **Create App ID** (Создать идентификатор приложения), чтобы создать новое приложение Facebook.

На экране появится список некоторых рекомендуемых продуктов. При наведении указателя мыши на любой из них появятся две кнопки: **Read Docs**

(Прочитать описание) и **Set Up** (Установить). Найдите продукт **Messenger**, наведите на него указатель мыши, как показано на рис. B.6. Щелкните на кнопке **Set Up** (Установить).

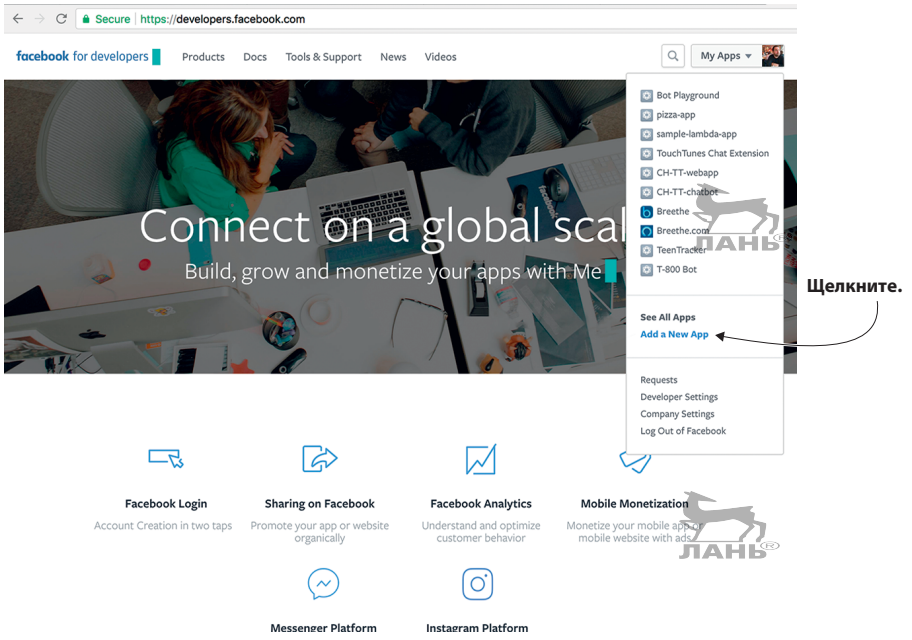


Рис. B.4. Портал разработчиков Facebook

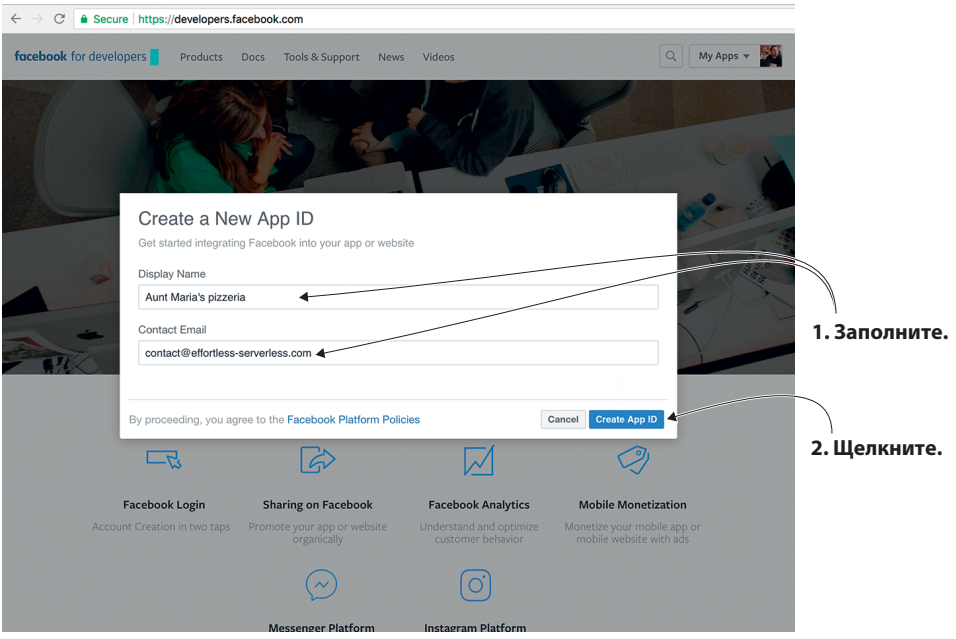


Рис. B.5. Создание нового приложения Facebook

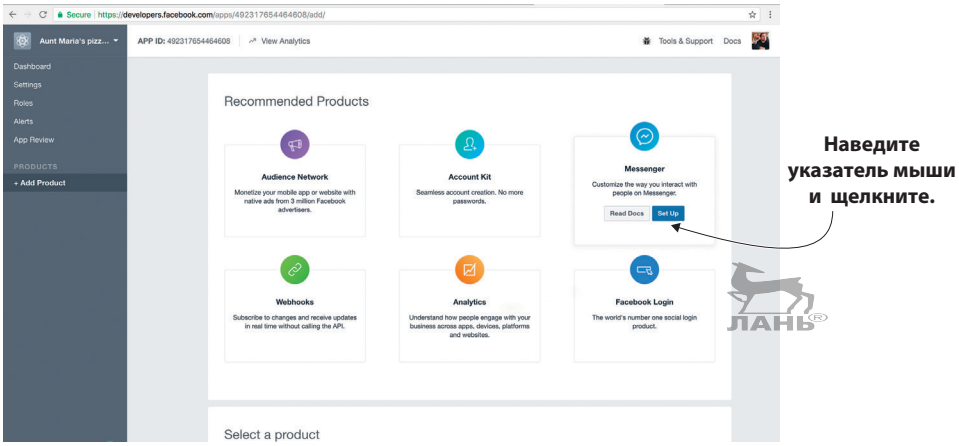


Рис. B.6. Список рекомендуемых продуктов

После щелчка откроется форма с настройками **Messenger Platform**, как показано на рис. B.7.

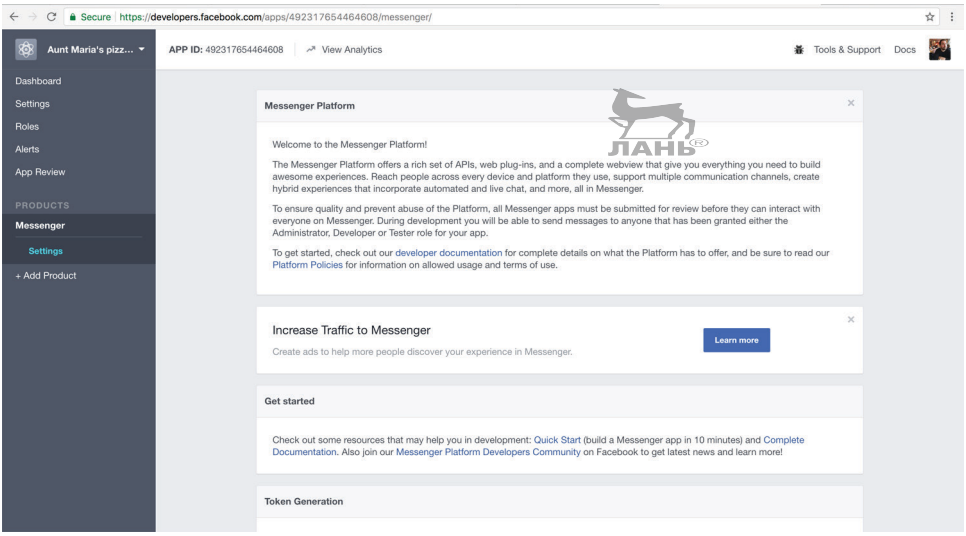


Рис. B.7. Форма с настройками Messenger Platform

Не закрывайте эту страницу в браузере, она вам вскоре понадобится.

B.1.3. Создание чат-бота Facebook Messenger с использованием Claudia Bot Builder

После создания страницы и приложения можно приступать к созданию чат-бота.

ПРИМЕЧАНИЕ. Приступая к выполнению следующего шага, имейте в виду, что библиотека Claudia должна быть установлена глобально, как описано в прило-

жении А. Также у вас должен быть инициализирован проект NPM и установлена библиотека Claudia Bot Builder как зависимость (как описывается в том же приложении А).
Наконец, вам понадобится код с реализацией чат-бота из листинга главы 8.

Откройте терминал и перейдите в папку проекта. Затем выполните команду из листинга А.1, чтобы создать функцию AWS Lambda и настроить чат-бот.

```
claudia create \
  --region eu-central-1 \
  --api-module bot \
  --configure-fb-bot
```

Выберите свой регион.

Выберите главный файл (здесь предполагается, что файл имеет имя bot.js, как описывалось в главе 8).

Сообщить библиотеке Claudia, что требуется настроить чат-бот Facebook Messenger.

ПРИМЕЧАНИЕ. Многострочные команды, такие как в следующем листинге, могут не поддерживаться в некоторых системах. Если у вас они не поддерживаются, вводите команду в одну строку, удаляя обратные слеши (\), которые сообщают командной строке, что команда продолжается на следующей строке.

В отличие от обычного развертывания Claudia, команда с параметром `--configure-fb-bot` имеет интерактивный характер. После развертывания кода в функции AWS Lambda команда выведет URL веб-обработчика и ключ проверки, необходимые для настройки чат-бота, как показано на рис. В.8. Эти значения понадобятся вам на следующем шаге.

Рис. В.8. Настройка чат-бота Facebook Messenger с использованием Claudia Bot Builder

Оставьте окно терминала открытым, потому что процесс еще не завершен. Вернитесь в форму настройки Messenger Platform в браузере (рис. В.7) и

щелкните на кнопке **Setup Webhook** (Настроить веб-обработчик) в разделе **Webhooks** (Веб-обработчики). После этого откроется диалог, как показано на рис. B.9. Заполните поля с URL веб-обработчика и ключом проверки, используя значения из окна терминала, полученные на предыдущем шаге. В разделе **Subscription Fields** (Поля подписки) установите флажки **messages** (сообщения) и **messaging_postbacks** (отправка ответов). Затем щелкните на кнопке **Verify and Save** (Проверить и сохранить).

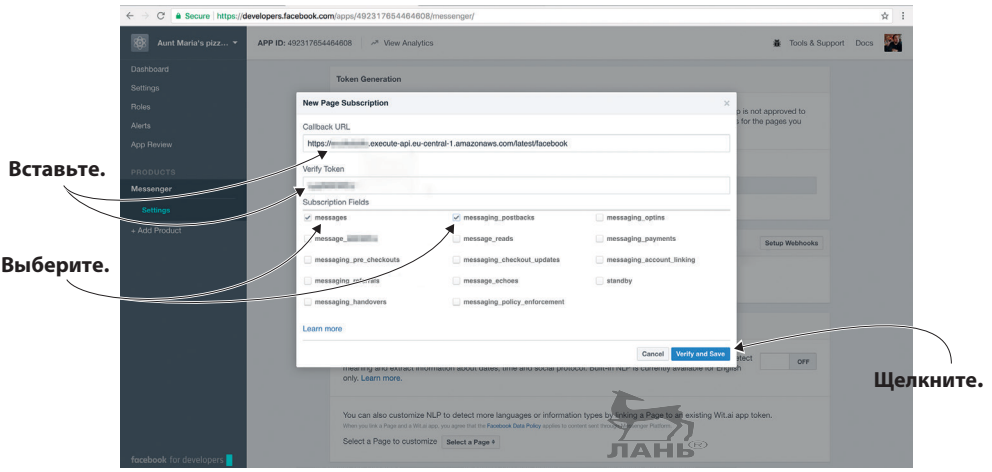


Рис. B.9. Настройка веб-обработчика и ключа проверки

Через короткое время диалог закроется, и вы увидите настроенный веб-обработчик, как показано на рис. B.10.

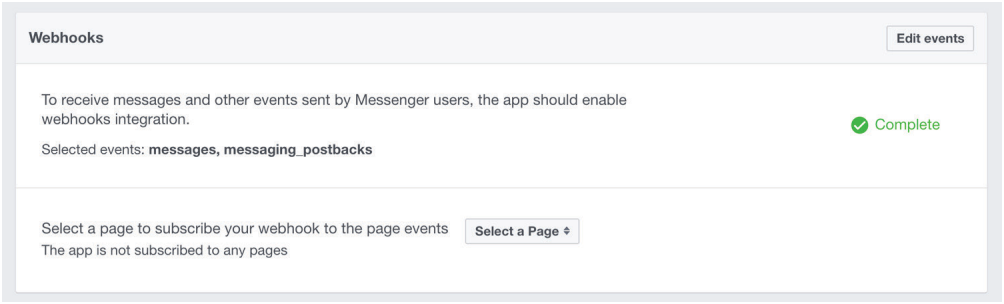


Рис. B.10. Подтверждение активации веб-обработчика

Следующий шаг – получение ключа доступа к странице Facebook. Чтобы получить ключ, перейдите в раздел **Token Generation** (Создание ключа), в раскрывающемся меню выберите страницу и скопируйте ключ, как показано на рис. B.11.

Вернитесь в терминал и вставьте ключ доступа, скопированный на предыдущем шаге, нажмите клавишу **Enter**, как показано на рис. B.12.

После этого команда предложит вам ввести закрытый ключ приложения Facebook. Этот ключ необходим, потому что с его помощью Facebook прове-

ряет, действительно ли сообщение получено от вашего чат-бота, а не из какого-то другого источника. Секретный ключ будет сохранен в переменной стадии в API Gateway.

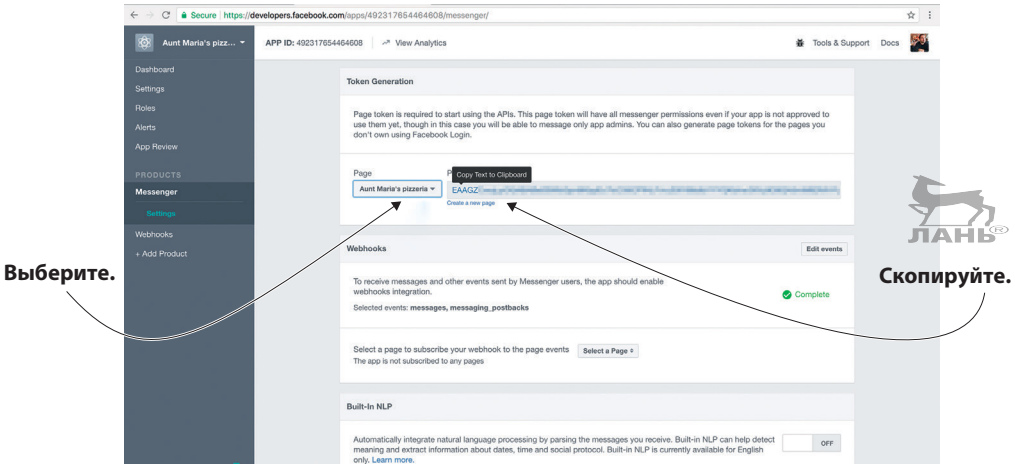


Рис. B.11. Создание ключа доступа к странице

Рис. B.12. Настройка ключа страницы

Чтобы увидеть закрытый ключ приложения Facebook, вернитесь в браузер и выберите вкладку **Dashboard** (Дашборд) в меню слева. Щелкните на кнопке **Show** (Показать) рядом с полем **App Secret** (Закрытый ключ приложения), как показано на рис. B.13. Скопируйте это значение и вернитесь в окно терминала.

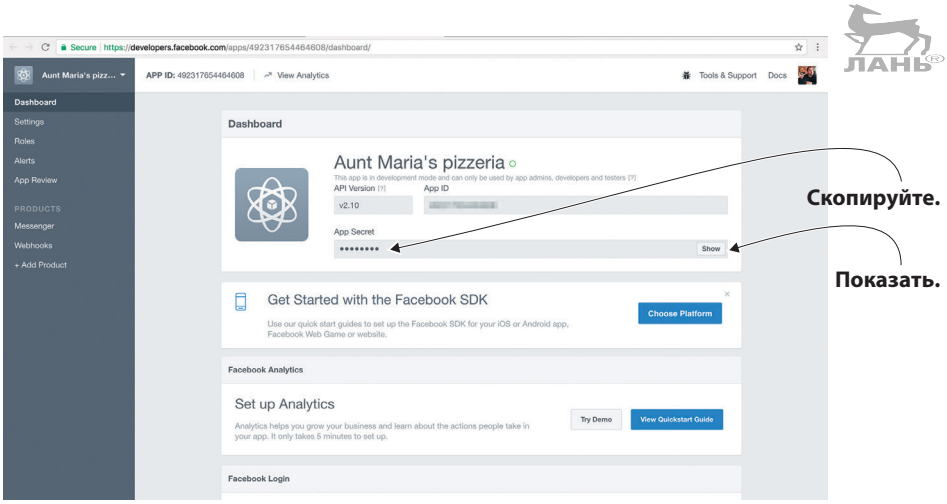


Рис. B.13. Копирование закрытого ключа приложения

Вставьте ключ и нажмите клавишу **Enter**, как показано на рис. B.14.

Рис. B.14. Настройка закрытого ключа приложения

Когда команда завершится, вы должны увидеть ответ, как показано в листинге B.1.

Листинг B.1. Ответ команды после завершения создания чат-бота

```
{
  "lambda": {                    ← Информация о AWS Lambda.
    "role": "pizza-fb-bot-executor",
    "name": "pizza-fb-bot",
    "region": "eu-central-1"
```

```
  },
  "api": {          ← Информация о API Gateway.
    "id": "wvztkdiz8c",
    "module": "bot",
    "url": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/latest",
    "deploy": {
      "facebook": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/
     latest/facebook",
      "slackSlashCommand": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.
     com/latest/slack/slash-command",
      "telegram": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/
     latest/telegram",
      "skype": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/latest/
     skype",
      "twilio": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/latest/
     twilio",
      "kik": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/latest/
     kik",
      "groupme": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/
     latest/groupme",
      "line": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/latest/
     line",
      "viber": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/latest/
     viber",
      "alexa": "https://wvztkdiz8c.execute-api.eu-central-1.amazonaws.com/latest/
     alexa"
    }
  }
}
```

Веб-обработчики для всех поддерживаемых платформ, включая Facebook Messenger.

В своем ответе команда перечислит все веб-обработчики, но они вам не понадобятся, потому что Claudia позаботится обо всем автоматически. Кроме того, Claudia автоматически подпишет чат-бот на получение событий от страницы, как показано на рис. B.15.

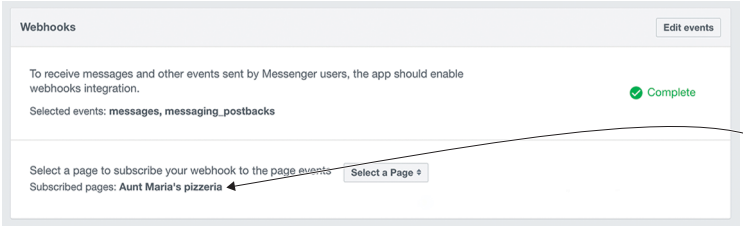


Рис. B.15. Чат-бот автоматически подписывается на получение событий от страницы

Теперь попробуйте найти свою страницу в Facebook Messenger. Вы должны получить результат, изображенный на рис. В.16.

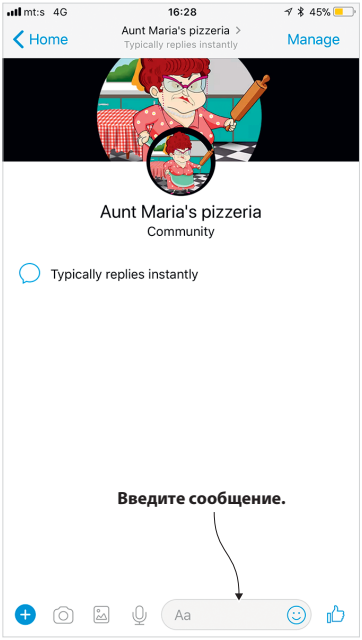


Рис. В.16. Начальная страница чат-бота в Facebook Messenger

И если вы пошлете сообщение боту, он должен ответить, как показано на рис. В.17.

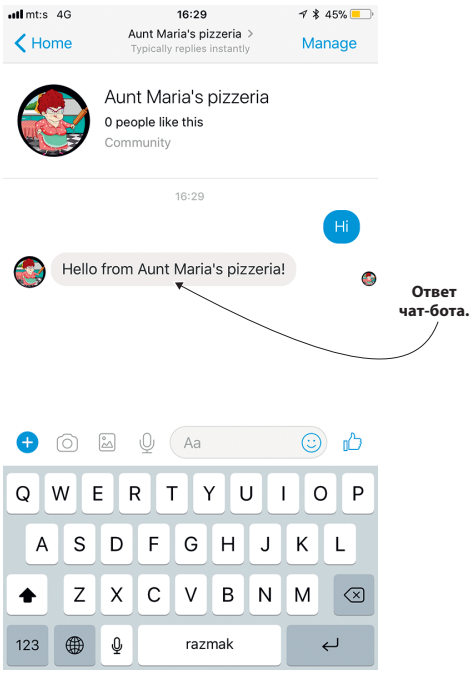


Рис. В.17. Ответ чат-бота Facebook Messenger

B.1.4. Подключение встроенного механизма NLP

Чтобы подключить встроенный механизм NLP, вернитесь в форму с настройками Messenger Platform на портале разработчиков Facebook и прокрутите ее вниз, до раздела **Built-In NLP** (Встроенный механизм NLP). Затем выберите свою страницу Facebook в списке **Select a Page to Customize Built-In NLP** (Выбор страницы для настройки встроенного механизма NLP), как показано на рис. B.18.

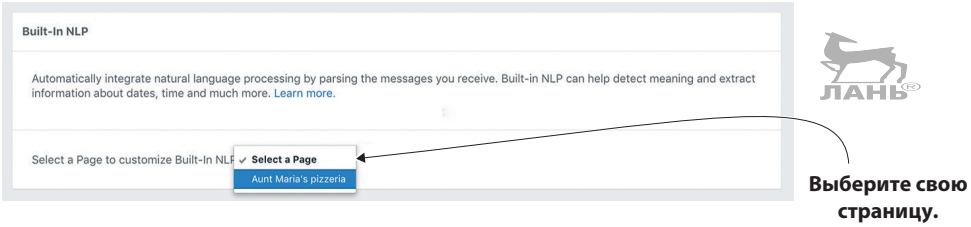


Рис. B.18. Выбор страницы для подключения встроенного механизма NLP

Теперь вы можете подключить встроенный механизм NLP: выберите язык по умолчанию и выполните дополнительные появившиеся настройки. Для приложения пиццерии в этой книге используется английский язык, поэтому настройки встроенного механизма NLP выполнены, как показано на рис. B.19.

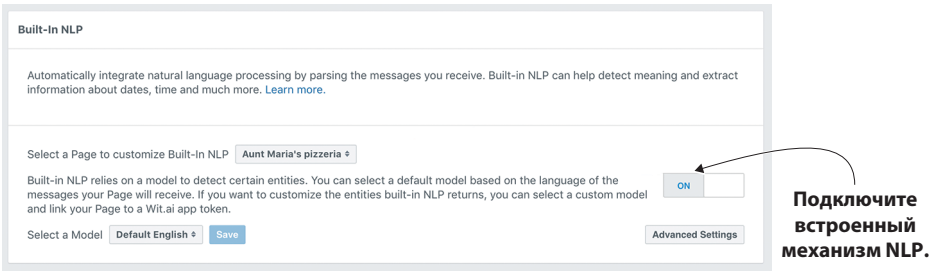


Рис. B.19. Подключение встроенного механизма NLP

B.2. Настройка Twilio

Чтобы настроить чат-бот Twilio SMS для опробования примеров в главе 10, нужно выполнить следующие шаги:

1) зарегистрировать учетную запись Twilio;
2) получить номер Twilio;
3) настроить свою службу Twilio Programmable SMS;
4) создать чат-бота Twilio SMS с использованием Claudia Bot Builder.

ПРИМЕЧАНИЕ. Twilio предоставляет пробный бесплатный период, поэтому в течение некоторого времени вам не придется платить за услугу, но спустя установленный период вам будет предложено заплатить за обслуживание.

B.2.1. Создание учетной записи Twilio

Если у вас уже есть учетная запись Twilio, переходите к следующему разделу «Получение номера Twilio».

Чтобы зарегистрировать новую учетную запись Twilio, откройте страницу https://www.twilio.com/try-twilio. Введите информацию о себе в форме регистрации. Там же вы увидите четыре раскрывающихся списка, как показано на рис. B.20.

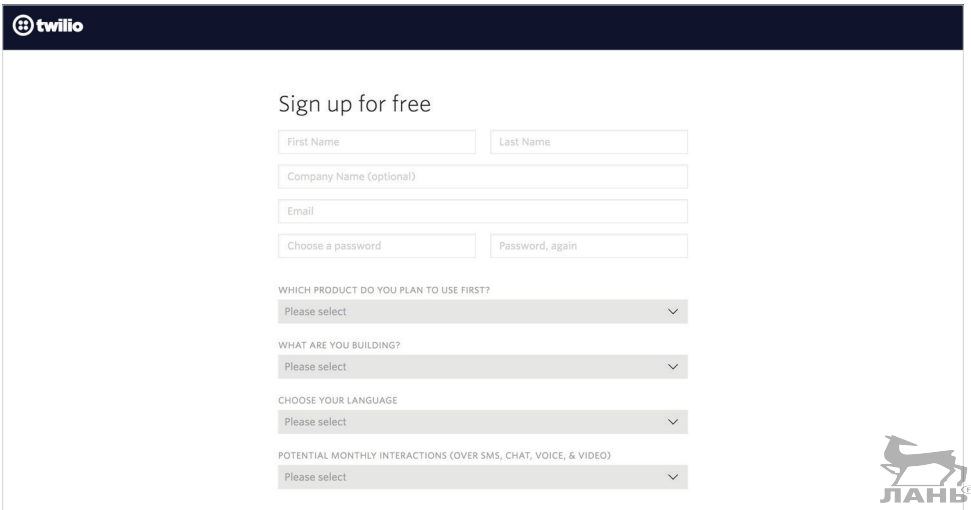


Рис. B.20. Регистрация новой учетной записи Twilio

В этих раскрывающихся списках:

1) в списке выбора продукта выберите **SMS**;
2) в списке выбора цели выберите **SMS Support** (SMS-поддержка);
3) в списке выбора языка выберите **Node.js**;
4) в списке выбора количества планируемых операций выберите **Less Than 100,000** (Меньше 100 000) или, если вы планируете большее число операций, можете выбрать любое другое значение из предложенных.

После заполнения всех полей служба Twilio пожелает убедиться, что общается с человеком, и пошлет вам проверочное сообщение SMS. Вы должны ввести свой номер мобильного телефона, на который придет SMS с кодом аутентификации. Введите этот код на следующем появившемся экране.

В случае успешной проверки вашего номера Twilio предложит создать новый проект, как показано на рис. B.21.

Укажите название проекта и щелкните на кнопке **Create Project** (Создать проект). Далее вы увидите страницу проекта Programmable SMS, как показано на рис. B.22.

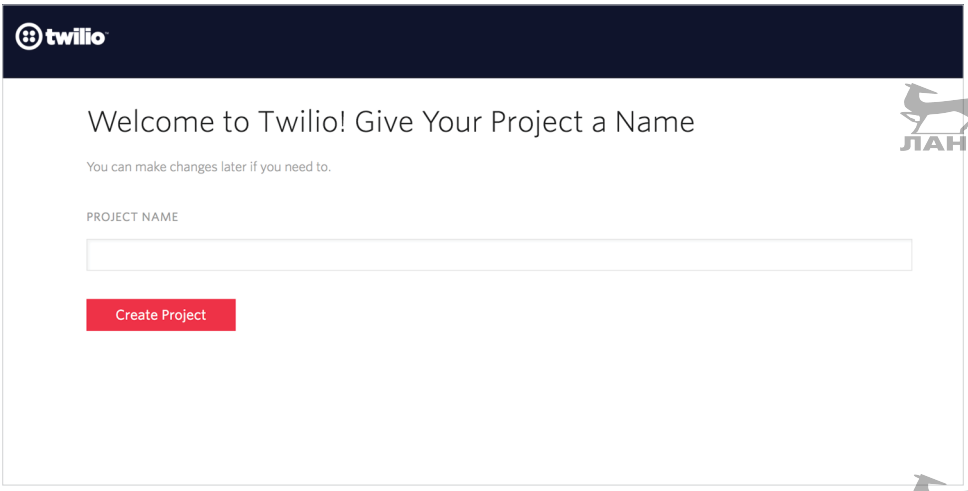


Рис. B.21. Создание нового проекта Twilio

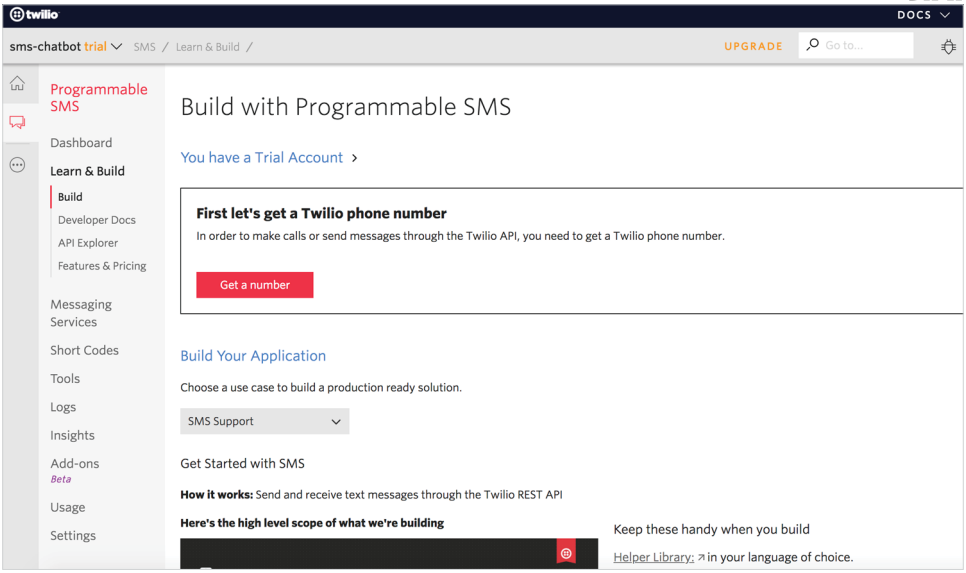


Рис. B.22. Страница проекта Programmable SMS

B.2.2. Получение номера Twilio

Если у вас уже есть номер Twilio, переходите к следующему разделу «Настройка службы Twilio Programmable SMS».

Иначе на странице проекта щелкните на кнопке **Get a Number** (Получить номер). В ответ появится диалог с предложенным номером. Если он вам не понравится или вы просто захотите какой-нибудь другой номер, просто щелкните на ссылке **Search for a Different Number** (Найти другой номер) в диалоге. Если номер вас удовлетворяет, щелкните на ссылке **Choose This Number** (Выбрать этот номер).

Завершив обработку запроса на получение нового номера, Twilio выведет диалог **Congratulations** (Поздравляем) с выбранным вами телефонным номером. Щелкните на кнопке **Done** (Завершить), и Twilio откроет страницу вашего проекта Programmable SMS с выбранной вкладкой **Learn & Build** (Обучение и настройка).

B.2.3. Настройка службы Twilio Programmable SMS

Чат-бот для Twilio SMS должен автоматически посылать и получать сообщения. Для этого нужно настроить проект Programmable SMS как **Messaging Service** (Служба обмена сообщениями). Пункт **Messaging Service** (Служба обмена сообщениями) находится в навигационном меню слева на странице проекта (см. рис. B.23).

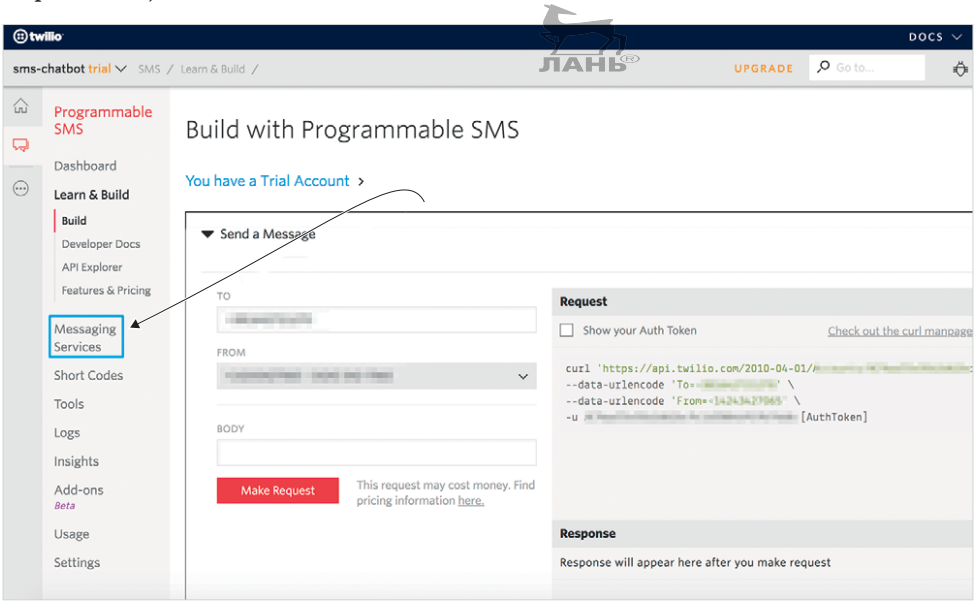


Рис. B.23. Пункт **Messaging Service** (Служба обмена сообщениями) в навигационном меню

Откройте эту вкладку и щелкните на кнопке **Create New Service** (Создать новую службу). Появится диалог, в котором вы должны ввести название вашей службы и указать вариант использования. Введите название «Aunt Maria's Pizzeria chatbot» и выберите вариант использования «Mixed» (Разное).

Далее откроется страница с настройками вновь созданной службы обмена сообщениями, как показано на рис. B.24.

На этой странице установите флажок **Process Inbound Messages** (Обрабатывать входящие сообщения). После этого появятся два поля ввода:

- **Request URL** (URL для отправки запросов);
- **Fallback URL** (URL для обработки ошибок).

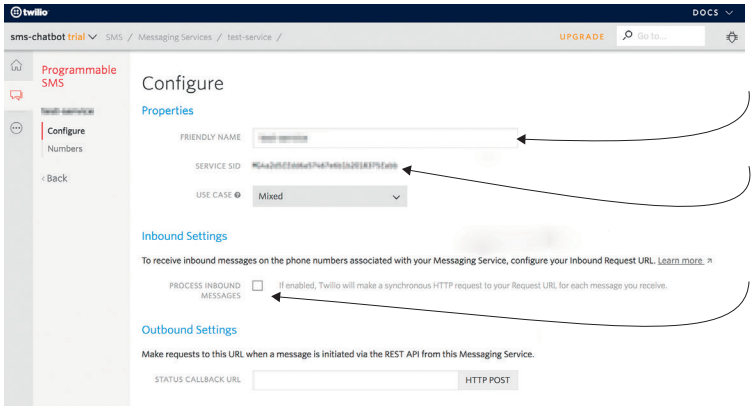


Рис. B.24. Настройка службы обмена сообщениями

В поле **Request URL** (URL для отправки запросов) введите URL своей бессерверной функции, реализующей чат-бот для Twilio SMS, созданный с использованием Claudia Bot Builder. Затем щелкните на кнопке **Save** (Сохранить).

Потом добавьте в эту службу номер Twilio, полученный в предыдущем разделе. Для этого щелкните на ссылке **Numbers** (Номера) в навигационном меню слева, на вкладке **Messaging Service** (Служба обмена сообщениями).

На странице **Numbers** (Номера) щелкните на кнопке **Add an Existing Number** (Добавить существующий номер). В ответ откроется диалог, изображенный на рис. B.25.

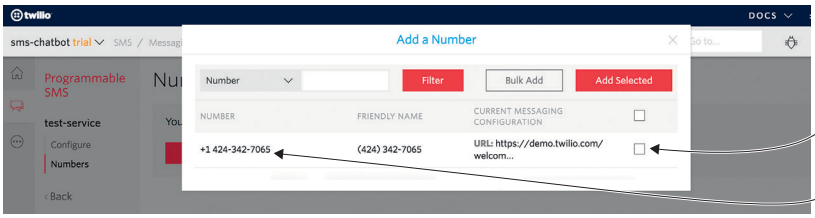


Рис. B.25. Добавление номера Twilio в службу обмена сообщениями

В этом диалоге вы увидите список своих номеров Twilio. Если список пуст, вернитесь к предыдущему разделу «Получение номера Twilio».

Чтобы добавить один или несколько номеров, установите флажки напротив нужных, а затем щелкните на кнопке **Add Selected** (Добавить выбранные). Номера появятся в списке на странице **Numbers** (Номера).

Если выше вы уже ввели URL бессерверного чат-бота для Twilio SMS в поле **Request URL** (URL для отправки запросов), тогда можете считать настройку проекта Programmable SMS и учетной записи Twilio завершенной. Поздравляем! Теперь вы можете опробовать свой чат-бот для SMS, отправив короткое сообщение на свой номер Twilio.

B.3. Настройка Alexa

Чтобы настроить поддержку Alexa, перейдите на страницу https://developer.amazon.com/alexa и выполните вход, используя свои учетные данные в Amazon. Затем щелкните на ссылке **Add Capabilities to Alexa** (Добавить возможности в Alexa), как показано на рис. B.26.

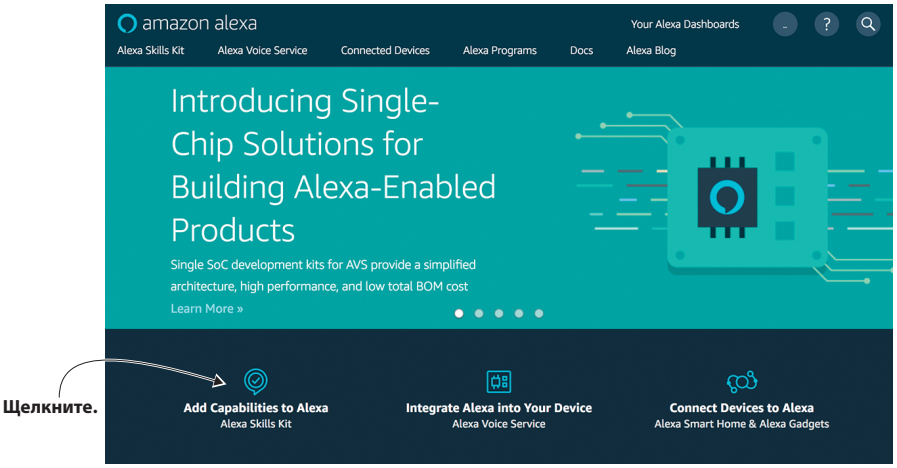


Рис. B.26. Дашборд Amazon Alexa

Откроется страница Alexa Skills Kit, где вы сможете найти документацию и руководства по проектированию, конструированию и запуску сценариев для Alexa. Здесь же вы сможете создавать новые сценарии. Для этого щелкните на кнопке **Start a Skill** (Запустить сценарий), как показано на рис. B.27, после чего откроется экран **Create a New Alexa Skill** (Создание нового сценария для Alexa).

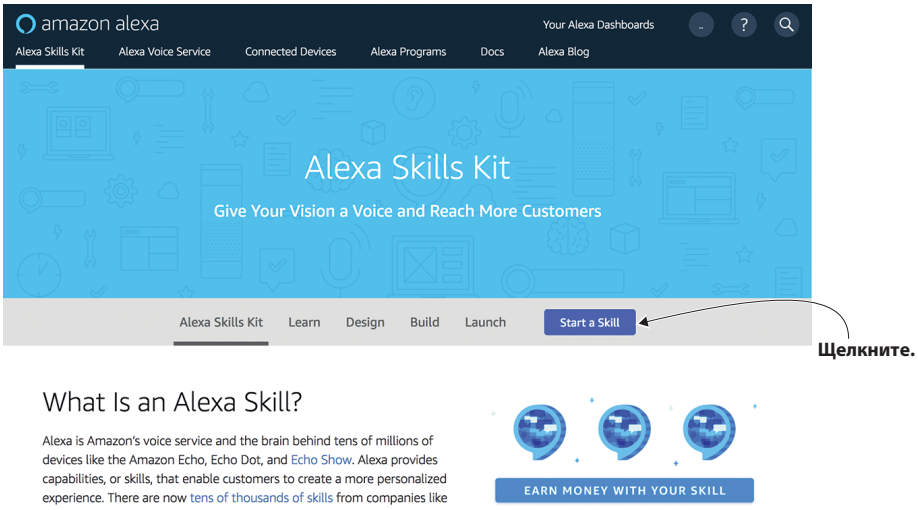


Рис. B.27. Страница Alexa Skills Kit

Процесс создания начинается в разделе **Skill Information** (Информация о сценарии). В этом разделе вы можете выбрать тип сценария, определить название и имя вызова для сценария, а также настроить глобальные параметры, такие как **Audio Player** (Аудиоплеер), **Video App** (Видеоприложение) и **Render Template** (Шаблон отображения).

В качестве типа сценария по умолчанию выбрано **Custom Interaction Model** (Своя модель взаимодействий); оставьте этот выбор как есть, потому что это позволит вам создать свой новый сценарий. Кроме этого типа, на выбор также доступны: **Smart Home Skill** (Умный дом), **Flash Briefing Skill** (Короткая встреча) и **Video Skill** (Видеосценарий), последний из которых предназначен для Amazon Echo Show и других видеоустройств с поддержкой Alexa.

В оба поля, с названием и именем вызова, введите «Aunt Maria's Pizzeria», убедитесь, что все остальные поля выключены, и щелкните на кнопке **Save** (Сохранить), как показано на рис. B.28. Щелкните на кнопке **Next** (Далее), чтобы перейти к следующему разделу.

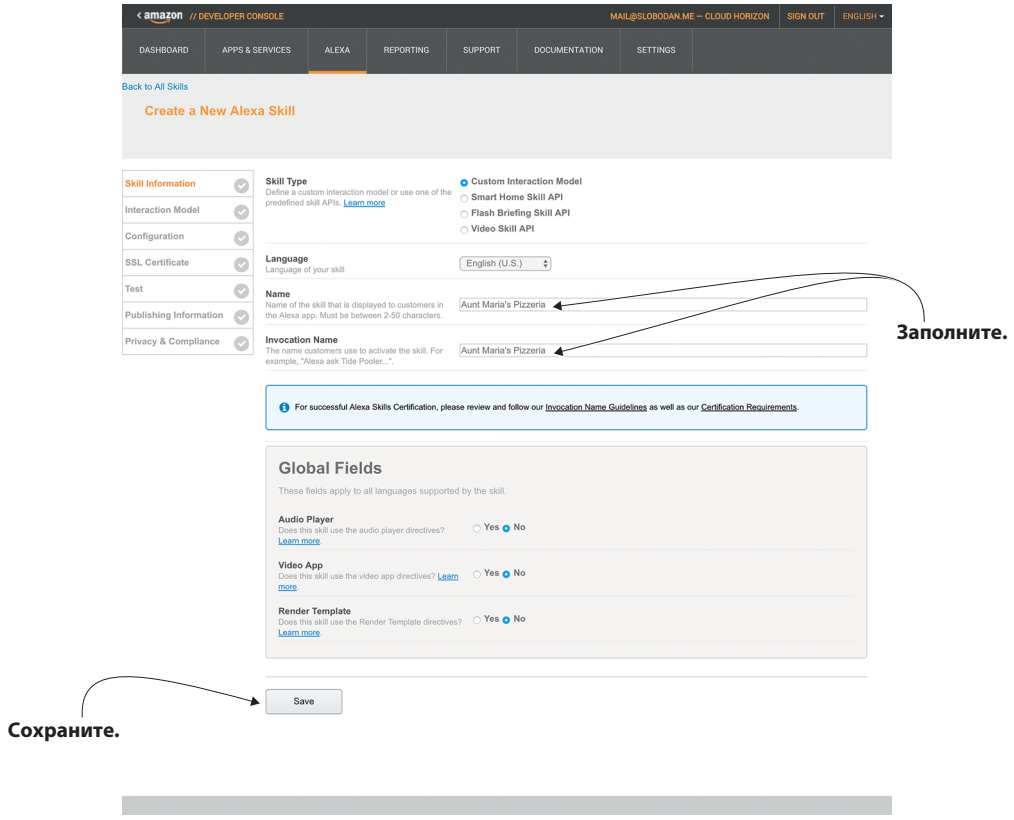


Рис. B.28. Настройка сценария

В следующем разделе **Interaction Model** (Модель взаимодействий) нужно определить схему намерений, слот и образцы выражений, которые мы предопределили в главе 10.

Сначала вставьте схему намерений (листинг 10.9) в поле **Intent Schema** (Схема намерений). Затем заполните форму **Custom Slot Types** (Свои типы слотов), добавив имя своего слота (`LIST_OF_PIZZAS`) и значения из главы 10 (листинг 10.10). Далее щелкните на кнопке **Add** (Добавить), как показано на рис. B.29.

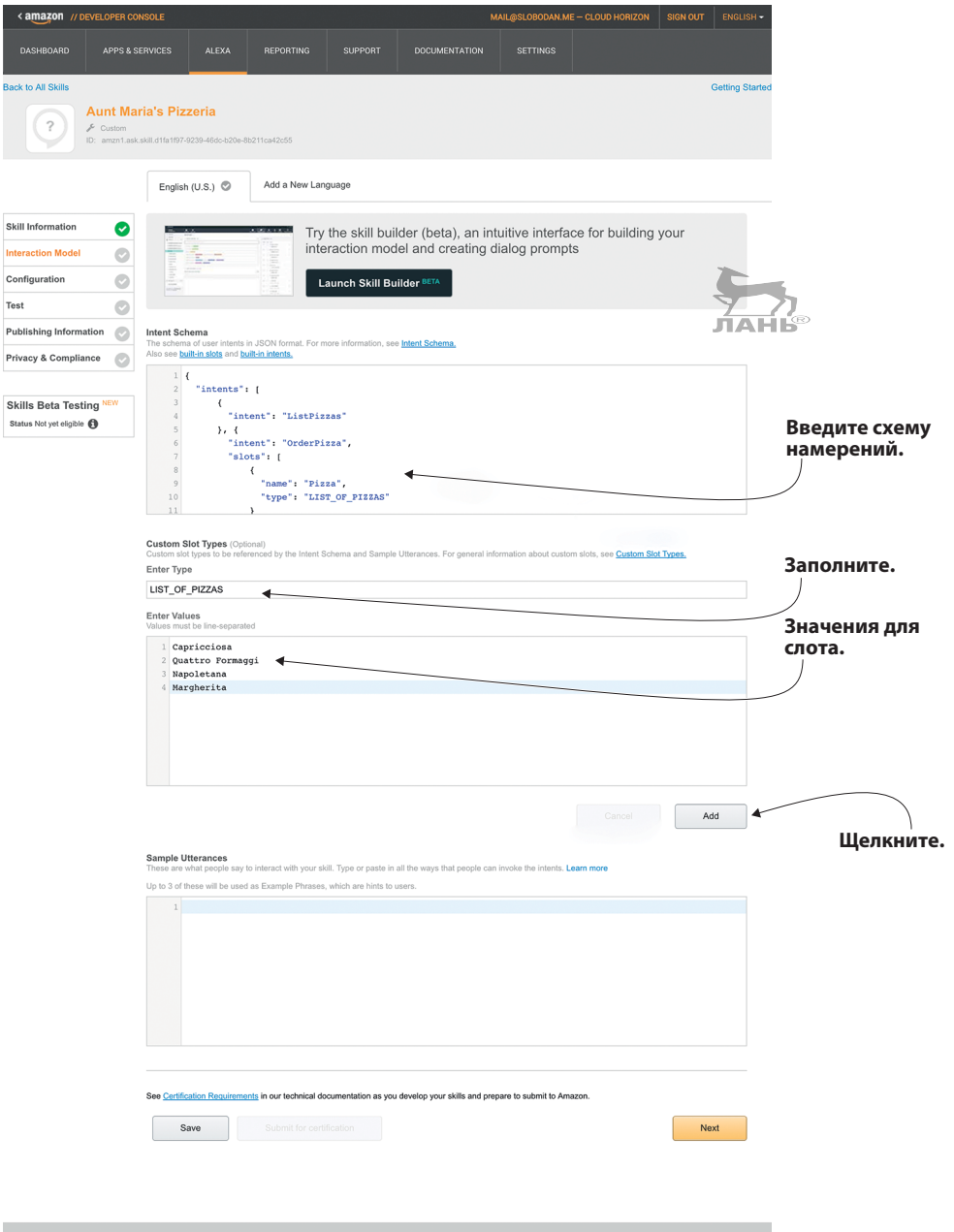


Рис. B.29. Настройка модели взаимодействий

После добавления своего типа слота введите образцы выражений из главы 10, затем щелкните на кнопке **Next** (Далее), как показано на рис. B.30.

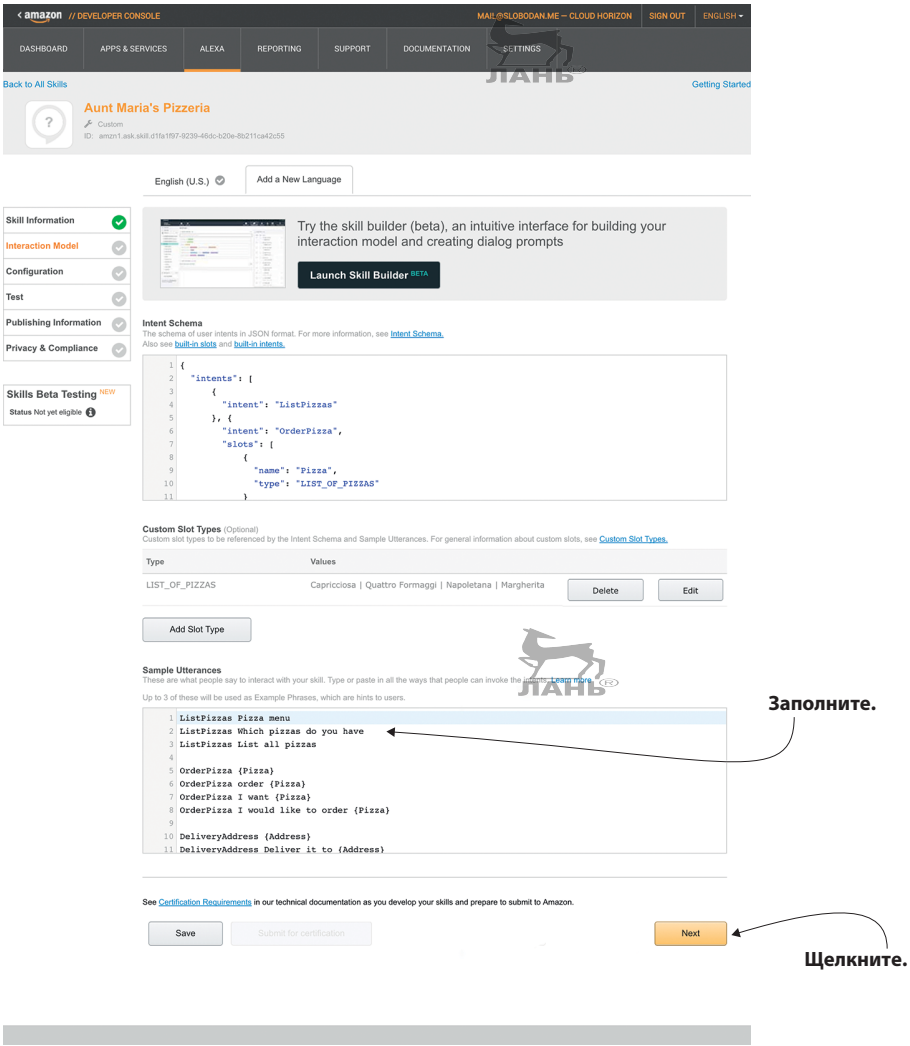


Рис. B.30. Добавление образцов выражений

Откроется раздел **Configuration** (Настройки), где нужно настроить веб-обработчик для поддержки сценария или указать URL функции AWS Lambda. Перед выполнением этого шага разверните функцию Lambda, если вы этого еще не сделали, а пока будете заниматься развертыванием, оставьте окно браузера открытым, потому что вам еще придется вернуться в раздел **Configuration** (Настройки).

Чтобы развернуть функцию Lambda, откройте терминал, перейдите в папку с кодом поддержки сценария для Alexa и запустите следующую команду:

```
claudia create --region eu-west-1 --handler skill.handler --version skill
```

Эта команда развернет функцию AWS Lambda в регионе eu-west-1 (только два региона, us-east-1 и eu-west-1, поддерживаются голосовым помощником Alexa) и настроит версию сценария.

Спустя несколько мгновений вы увидите стандартный ответ команды `claudia create`:

```
{
  "lambda": {
    "role": "pizza-alexa-skill-executor",
    "name": "pizza-alexa-skill",
    "region": "eu-west-1"
  }
}
```

Прежде чем появится возможность использовать функцию Lambda, нужно разрешить голосовому помощнику Alexa вызывать ее. Для этого выполните следующую команду:

```
claudia allow-alexa-skill-trigger --version skill
```

Эта команда позволит голосовому помощнику Alexa вызывать версию `skill` вашей функции Lambda. Спустя несколько мгновений вы увидите ответ:

```
{
  "Sid": "Alexa-1518380119842",
  "Effect": "Allow",
  "Principal": {
    "Service": "alexa-appkit.amazon.com"
  },
  "Action": "lambda:InvokeFunction",
  "Resource": "arn:aws:lambda:eu-west-1:721177882564:function:pizza-alexa-skill:skill"
}
```

Скопируйте Lambda ARN (значение поля `Resource` в JSON-ответе), вернитесь на страницу с настройками сценария в браузере. Выберите радиокнопку **AWS Lambda ARN** в поле **Service Endpoint Type** (Тип конечной точки службы) и вставьте ARN своей функции Lambda в поле ниже, как показано на рис. B.31.

Здесь мы не собираемся создавать несколько конечных точек для разных географических регионов (например, для США и Соединенного Королевства), поэтому выберите **No** (Нет) в ответ на вопрос «Provide geographical region endpoints?» (Определить конечные точки для географических регионов?) и щелкните на кнопке **Next** (Далее).

После настройки сценария появится экран **Test** (Тестирование). Здесь вы можете проверить работу своего сценария, например введите выражение в поле **Service Simulator** (Имитатор службы) и прослушайте ответ, щелкнув

на кнопке **Listen** (Слушать), как показано на рис. B.32. Ваш сценарий доступен также на вашем устройстве Alexa, поэтому вы можете сказать: «Alexa, start Aunt Maria's Pizzeria» («Алекса, начать работу с пиццерией тетушки Марии»)[1].

Сценарий теперь доступен на вашем устройстве Alexa, но если вы решите сделать его доступным для всех, то должны отправить его на проверку, как описывается на странице https://developer.amazon.com/docs/custom-skills/submit-an-alexa-skill-for-certification.html.

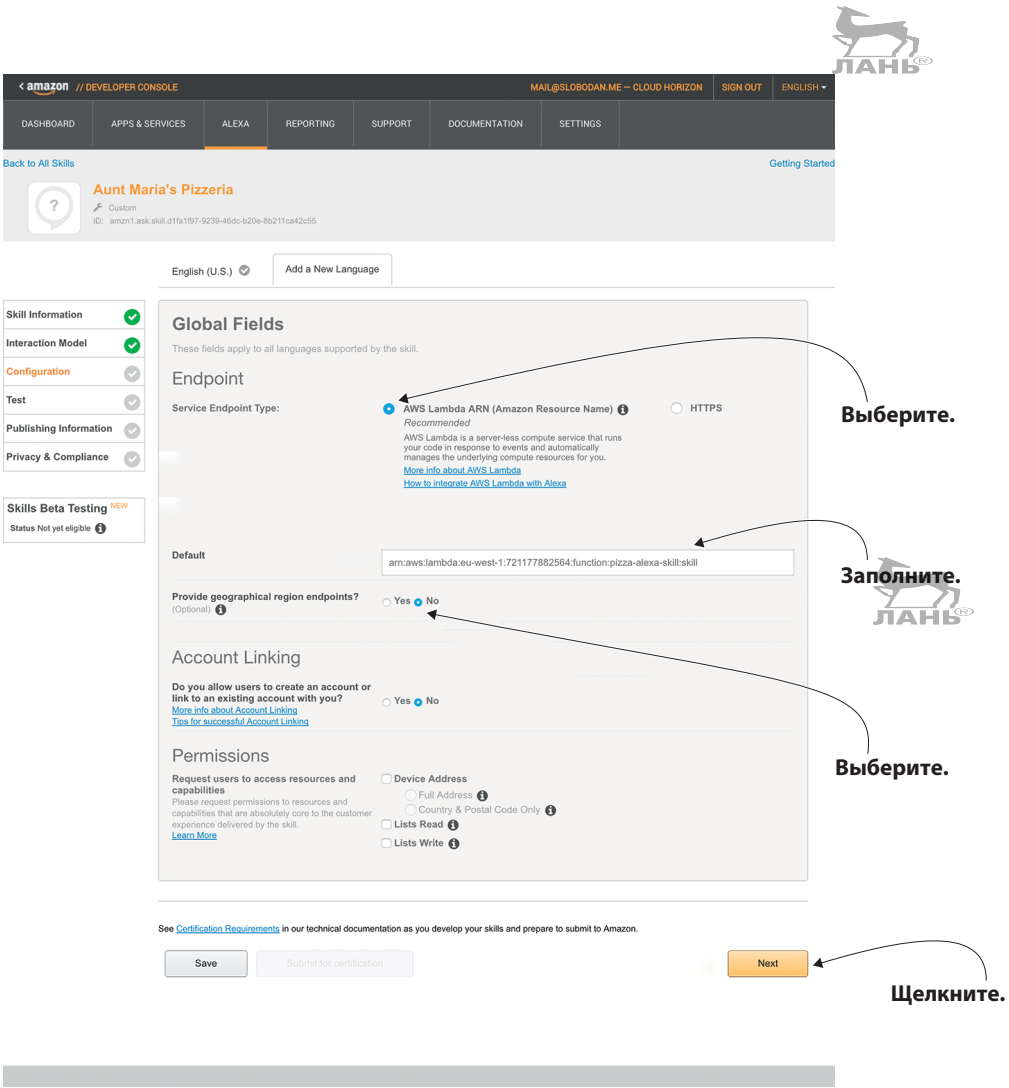


Рис. B.31. Настройка сценария

[1] На момент публикации книги (2019 год) голосовой помощник Алекса поддерживал только английский и немецкий языки. Имейте это в виду при опробовании примеров. – *Прим. перев.*

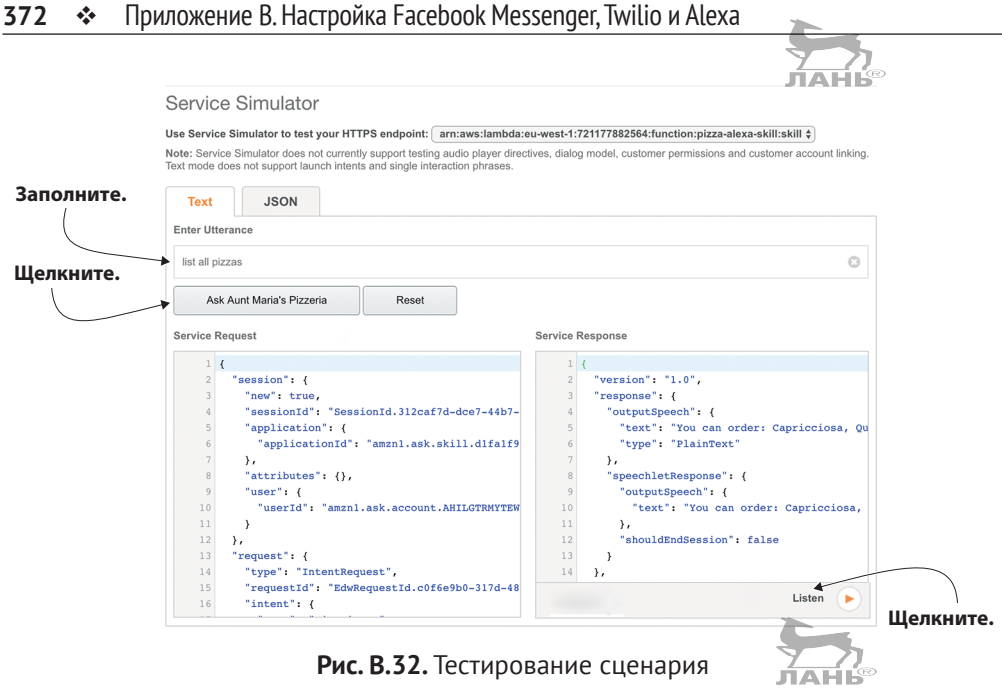


Рис. В.32. Тестирование сценария

Приложение C

Настройка Stripe и MongoDB

В этом приложении описывается порядок:

- настройки учетной записи Stripe и получения ключей Stripe API;
- установки и настройки MongoDB.

C.1. Настройка учетной записи Stripe и получение ключей Stripe API

Создание и настройка учетной записи Stripe и получение ключей API необходимы для опробования примеров в главе 12, где рассказывается о создании своей бессерверной платежной службы. Процесс состоит из следующих шагов:

1) регистрация учетной записи Stripe;
2) получение ключей Stripe API;
3) создание бессерверной платежной службы Stripe с использованием Claudia API Builder.

Если у вас уже есть учетная запись Stripe, но пока нет ключей, переходите к разделу «Получение ключей Stripe API».

C.1.1. Создание учетной записи Stripe

Учетная запись Stripe создается легко и быстро. Откройте браузер и перейдите по адресу https://stripe.com. Щелкните на кнопке **Create Account** (Создать учетную запись), после чего откроется форма регистрации в Stripe.

Введите свой адрес электронной почты, полное имя и пароль. После отправки формы Stripe предложит вам добавить номер телефона для восстановления пароля. Мы советуем сделать это на всякий случай.

После этого будет создана ваша учетная запись, но не забудьте подтвердить адрес электронной почты – Stripe не будет принимать платежи на ваш счет, если этого не сделать. Теперь у вас есть своя учетная запись Stripe!

C.1.2. Получение ключей Stripe API

Если вы предполагаете применять Stripe для приема платежей в своих приложениях, вам необходимо использовать Stripe API. Платежная система

Stripe должна иметь возможность идентифицировать вас при использовании ее API. Для целей идентификации Stripe предоставит вам пару хешированных ключей, которые предназначены для применения во всех взаимодействиях с ее API. Эти ключи автоматически генерируются при создании учетной записи Stripe.

После создания учетной записи, как описывалось в предыдущем разделе, вы должны получить ключи API. Для этого откройте страницу https://dashboard.stripe.com.

В навигационном меню выберите пункт **API** (см. рис. C.1).

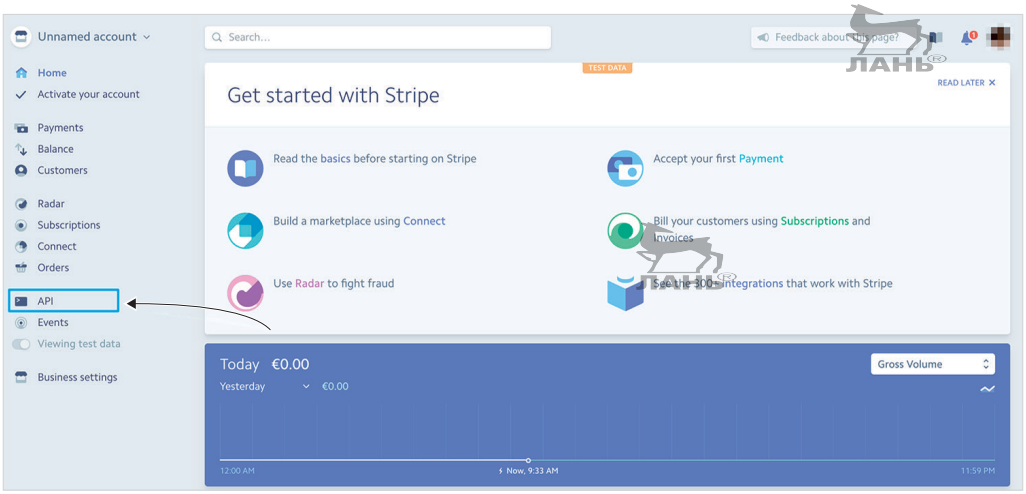


Рис. C.1. Дашборд Stripe

На странице **API** вы увидите две таблицы: со стандартными и ограниченными ключами API (рис. C.2). Вам нужно сохранить стандартные ключи API; они чаще всего используются в бессерверных платежных службах.

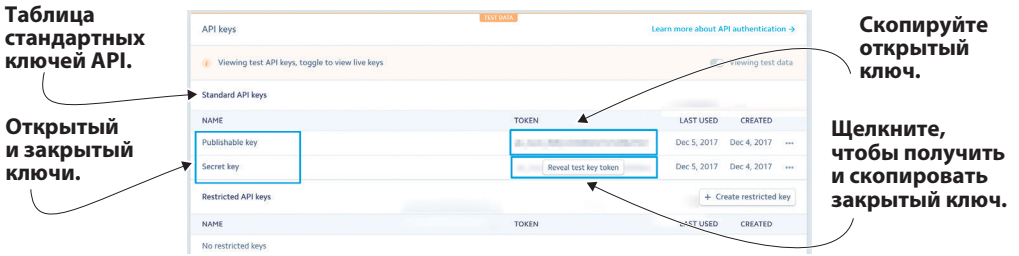


Рис. C.2. Таблица стандартных ключей API

Как отмечалось выше, вместе с учетной записью автоматически создаются два стандартных ключа API: открытый и закрытый. Открытый ключ может использоваться в ваших интерфейсных мобильных или веб-приложениях. Его можно безопасно публиковать, так как он является чем-то, напоминающим ваш адрес электронной почты.

Закрытый ключ обеспечивает доступ ваших приложений или API к ресурсам Stripe. Благодаря ему Stripe будет знать, что именно вы используете ресурсы платежной системы. Он играет роль пароля, и вы должны хранить его в тайне от всех, но не волнуйтесь: если у вас появится подозрение, что кто-то мог его украсть, вы сможете изменить открытый и закрытый ключи.

Скопируйте оба ключа в пустой документ на своем компьютере, чтобы иметь возможность быстрого доступа к ним, но не забудьте удалить этот документ, закончив читать главу 12.

ВНИМАНИЕ. Храните свой закрытый ключ в надежном месте и в скрытом виде. Будьте очень осторожны, обращаясь со своим закрытым ключом, потому что он может использоваться для доступа или даже манипулирования вашей учетной записью Stripe.

С.2. Установка и настройка MongoDB

MongoDB Atlas – это облачная служба базы данных MongoDB, разработанная и поддерживаемая той же командой, которая создала саму базу данных. В этом разделе вы создадите и настроите бесплатный экземпляр MongoDB, которого достаточно для опробования примеров кода из главы 13 и для работы с небольшим действующим приложением.

С.2.1. Создание учетной записи

Чтобы узнать больше о продукте и создать учетную запись MongoDB Atlas, откройте страницу https://www.mongodb.com/cloud/atlas в браузере (см. рис. С.3).

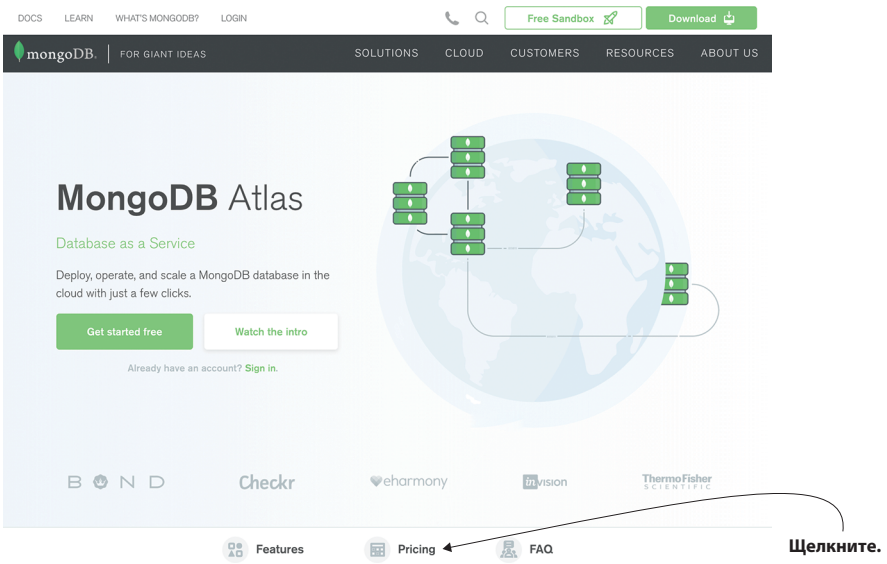


Рис. С.3. Начальная страница проекта MongoDB Atlas

Щелкните на вкладке **Pricing** (Тарифы), чтобы открыть ее, где вы сможете выбрать провайдера облачных услуг, регион и размер экземпляра. Выберите **AWS**, как показано на рис. С.4, и затем прокрутите страницу вниз.

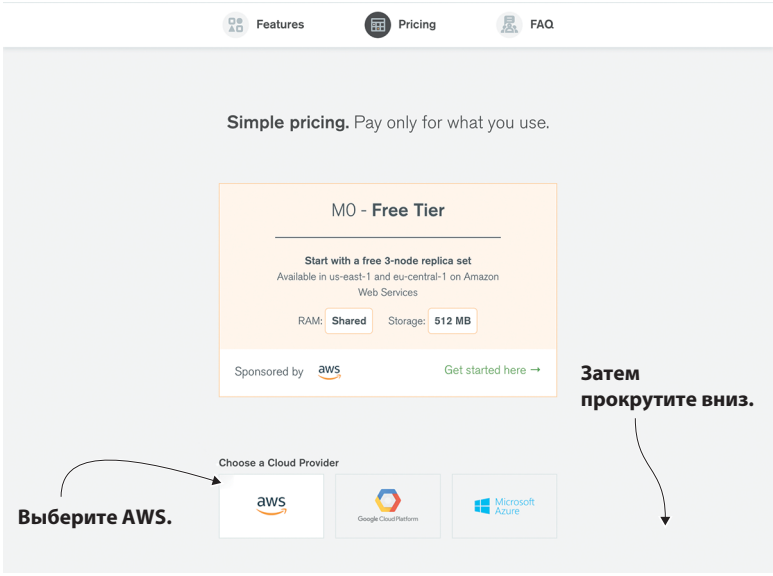


Рис. С.4. Выбор облачного провайдера на вкладке **Pricing** (Тарифы)

Под разделом для выбора провайдера облачных услуг выберите регион вашей функции Lambda (мы используем eu-central-1). Затем выберите экземпляр **M0** с суммой платежа $0 в месяц и щелкните на кнопке **Get Started Free** (Начать бесплатное обслуживание), как показано на рис. С.5.

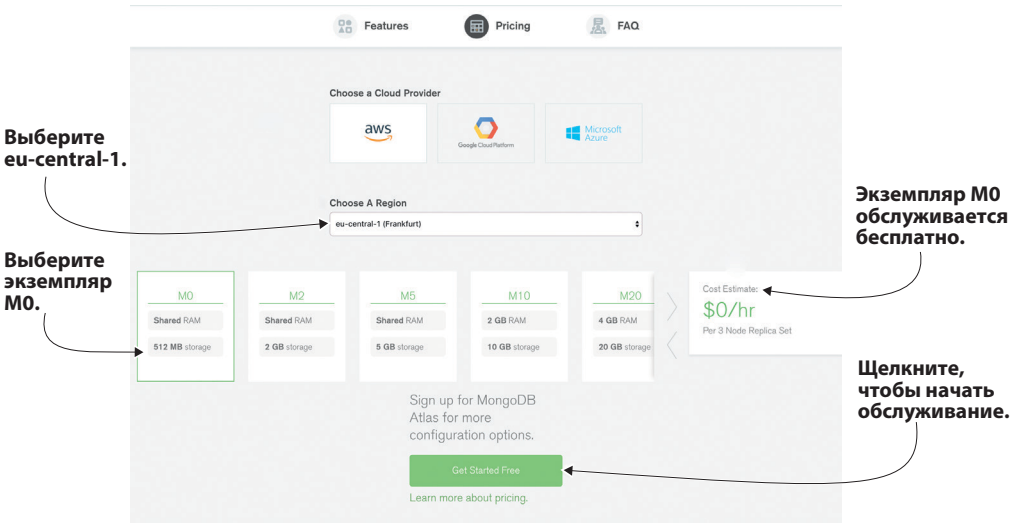


Рис. С.5. Выбор региона и размера экземпляра

В появившейся форме регистрации заполните обязательные поля и щелкните на кнопке **Continue** (Продолжить), как показано на рис. C.6. Когда выбирается бесплатный тариф, MongoDB Atlas не требует данных вашей кредитной карты, поэтому, щелкнув на кнопке **Continue** (Продолжить), вы создадите учетную запись и перейдете на страницу настройки.

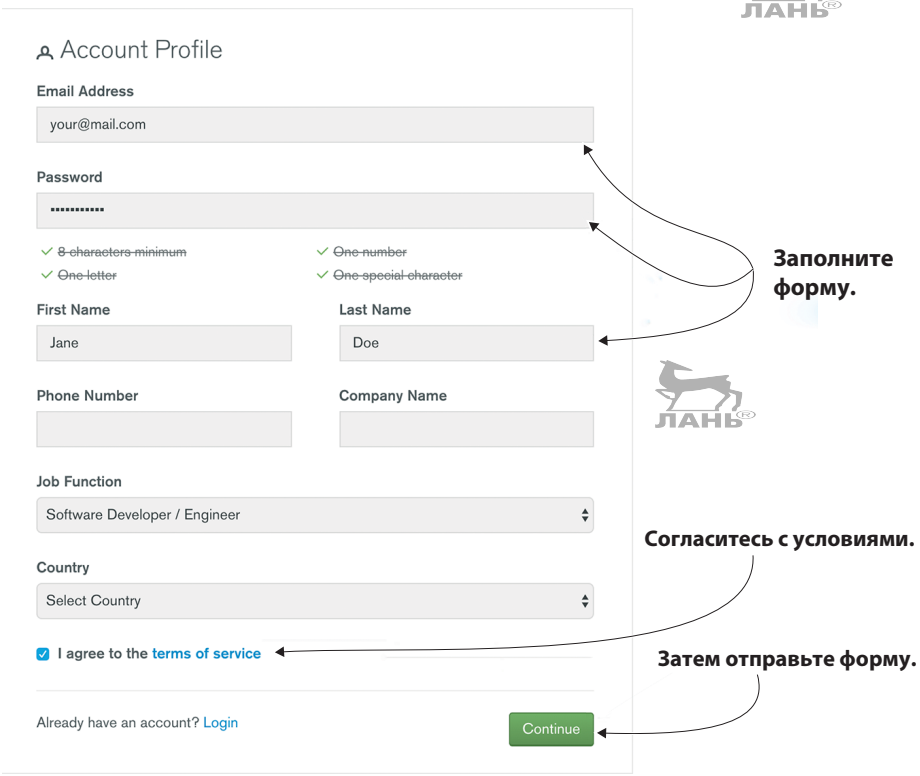


Рис. C.6. Создание учетной записи MongoDB Atlas

C.2.2. Настройка кластера

После создания учетной записи вам нужно создать свой первый кластер. Кластер баз данных – это набор баз данных, который управляется единственным экземпляром сервера баз данных. Как показано на рис. C.7, нужно добавить имя кластера (например, «RobertosTaxiCompany»). Убедитесь, что цена по-прежнему составляет $0, а затем щелкните на кнопке **Confirm & Deploy** (Подтвердить и развернуть).

После создания кластера появится дашборд MongoDB Atlas. Теперь вам нужно создать нового пользователя для вашей базы данных MongoDB. Для этого выберите вкладку **Security** (Безопасность) и щелкните на кнопке **Add New User** (Добавить нового пользователя), как показано на рис. C.8.

В диалоге **Add New User** (Добавить нового пользователя) введите имя пользователя (например, «roberto») и пароль. Затем перейдите в раздел **User Privileges** (Привилегии пользователя). Здесь вы сможете более точно настроить

разрешения для вашего нового пользователя. Добавляя пользователя для единственной базы данных, выберите **readWrite** в раскрывающемся списке слева, а затем введите имя базы данных в поле ввода, как показано на рис. C.9. Поскольку ваша база данных фактически еще не создана, вы можете ввести «taxi», и база данных будет создана автоматически. Закончив, щелкните на кнопке **Add User** (Добавить пользователя).

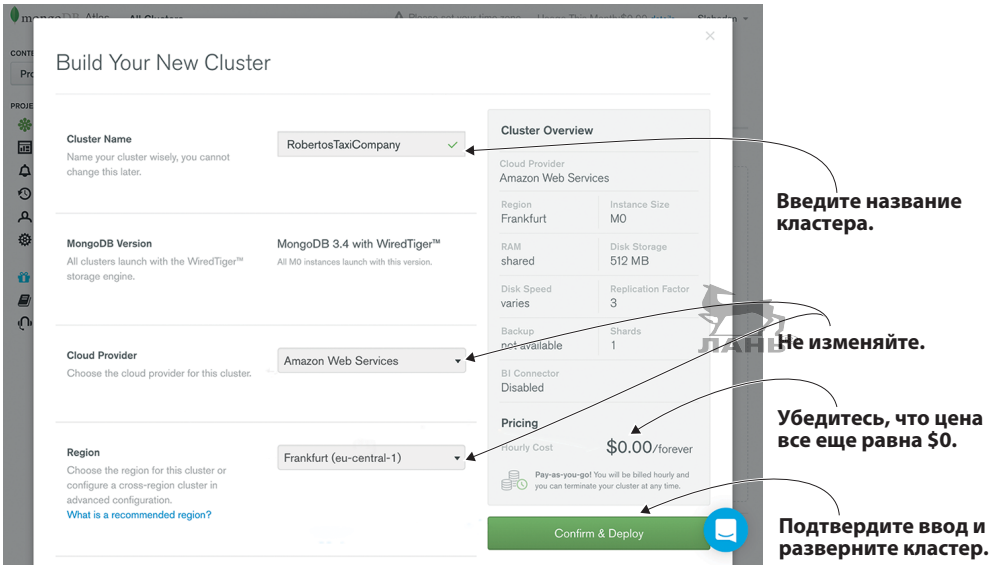


Рис. C.7. Создание кластера

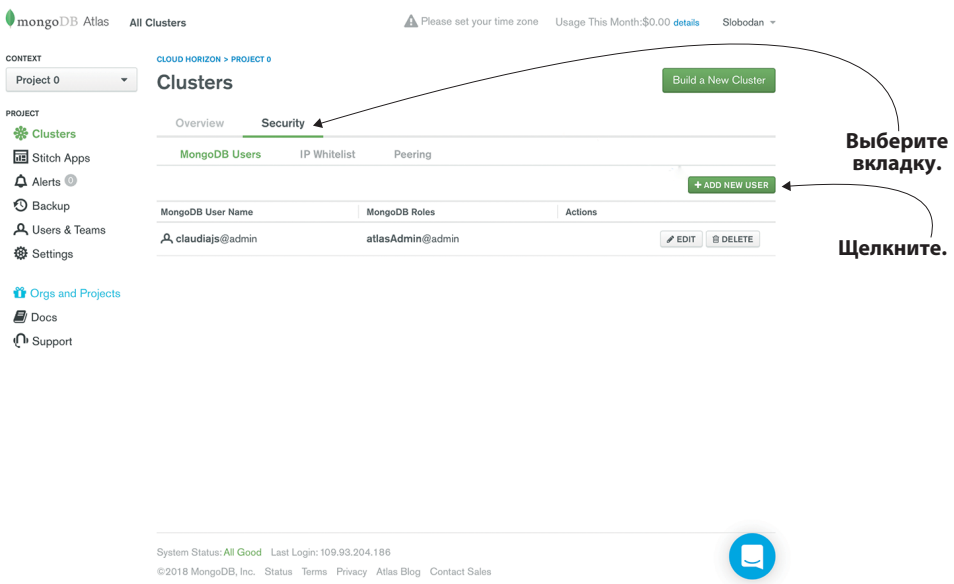


Рис. C.8. Вкладка **Security** (Безопасность) в дашборде MongoDB Atlas

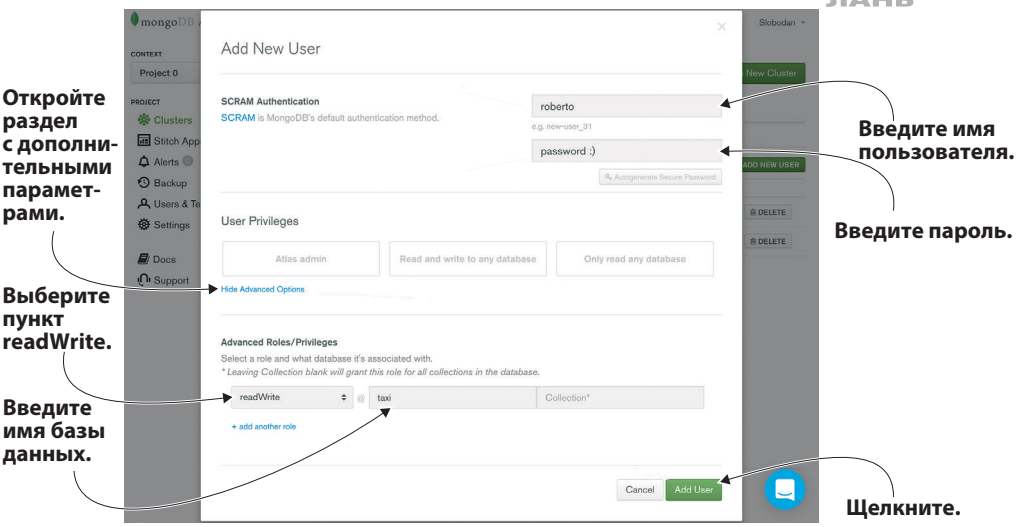


Рис. С.9. Создание нового пользователя

После создания нового пользователя вновь откроется вкладка **Security** (Безопасность) в дашборде MongoDB Atlas. Последний шаг в настройке базы данных – получение строки подключения. Для этого выберите вкладку **Overview** (Обзор) и щелкните на ссылке с названием кластера **RobertosTaxiCompany**, как показано на рис. С.10.

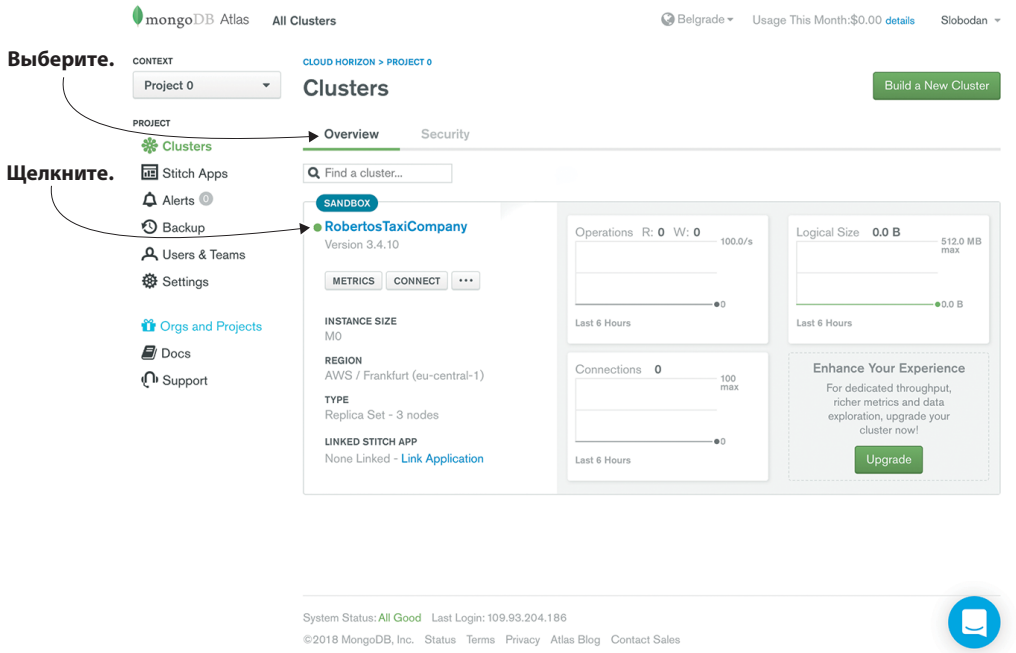


Рис. С.10. Вкладка **Overview** (Обзор)

Как показано на рис. C.11, щелкните на кнопке **Connect** (Подключиться), чтобы открыть диалог подключения.

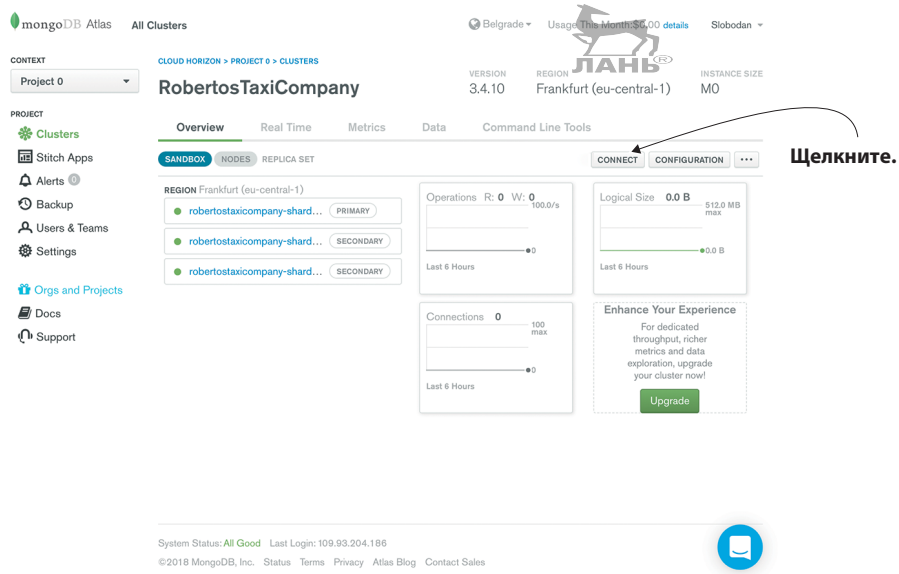


Рис. C.11. Обзор кластера RobertosTaxiCompany

Чтобы создать строку подключения, необходимо внести в белый список хотя бы один IP-адрес, который будет подключаться к вашему кластеру MongoDB. Для этого щелкните на кнопке **Add Entry** (Добавить запись), как показано на рис. C.12.

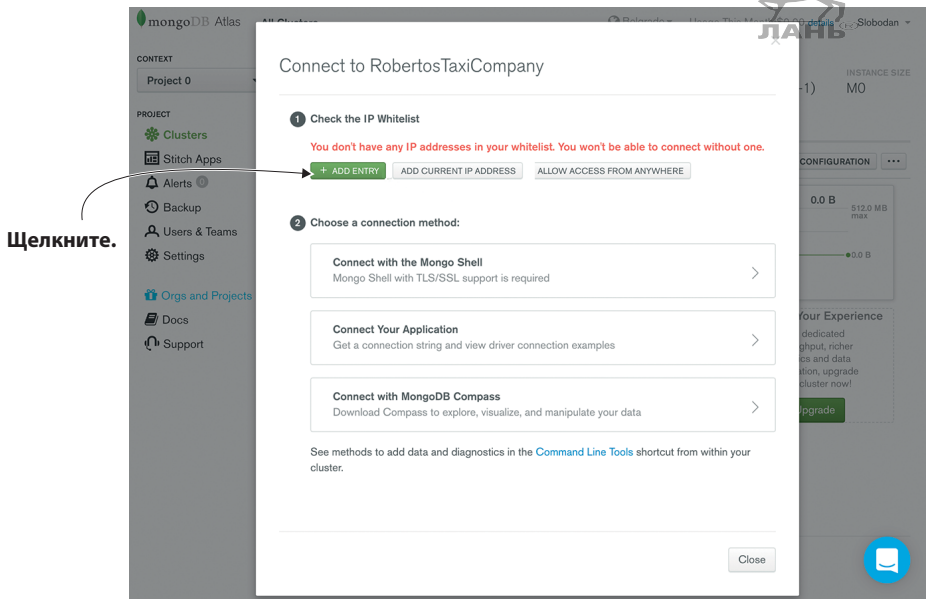


Рис. C.12. Подключение к кластеру

Но так как IP-адреса функций AWS Lambda заранее неизвестны, откройте кластер MongoDB для всех IP-адресов, указав значение `0.0.0.0/0` в поле IP-адреса. Затем добавьте описание и щелкните на кнопке **Save** (Сохранить), как показано на рис. С.13.

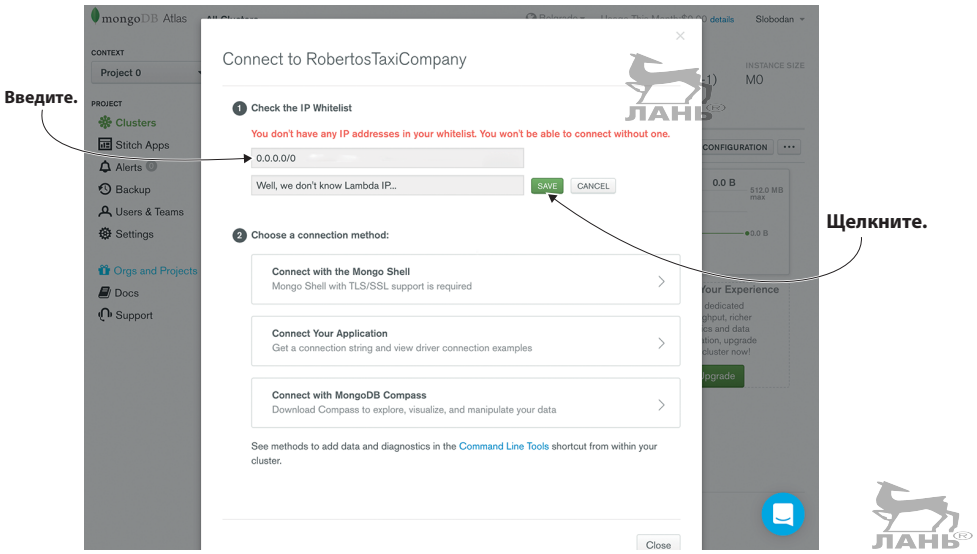


Рис. С.13. Добавьте в белый список все IP-адреса, потому что адреса функций AWS Lambda заранее неизвестны

Наконец, нужно выбрать метод подключения. Выберите **Connect Your Application** (Подключить ваше приложение), как показано на рис. С.14, потому что сейчас вам нужно получить строку подключения к MongoDB.

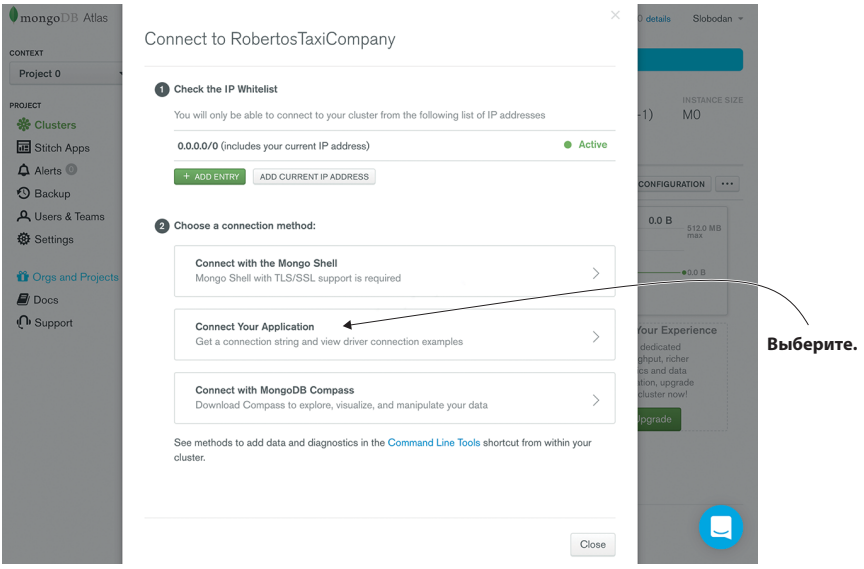


Рис. С.14. Получение строки подключения (шаг 1)

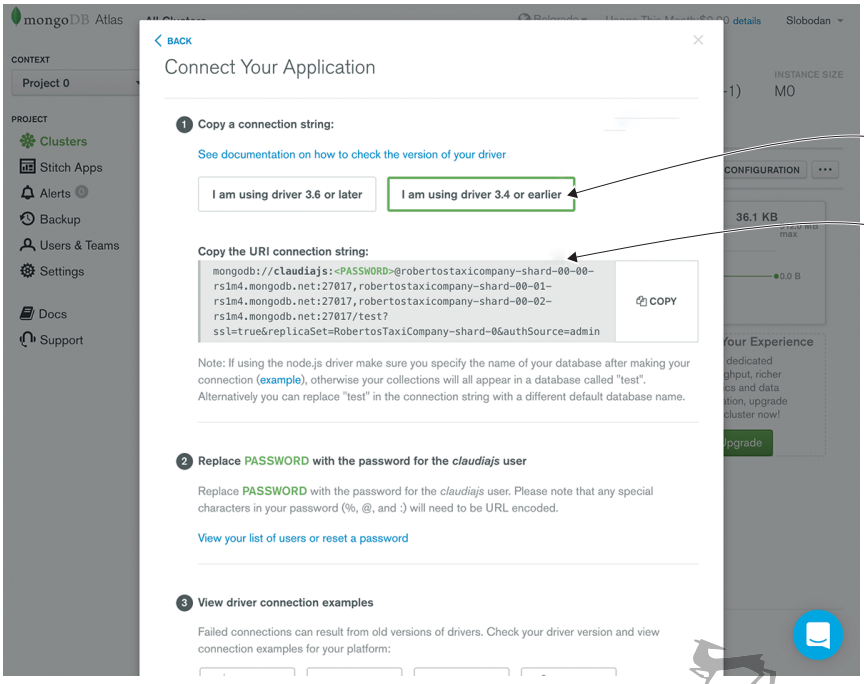


Рис. C.15. Получение строки подключения (шаг 2)

Просле этого откроется диалог со строкой подключения (см. рис. C.15). Щелкните по кнопке **I Am Using Driver 3.4 or Earlier** (Я использую драйвер версии 3.4 или ниже), потому что именно эта версия драйвера используется в главе 13, а затем скопируйте строку подключения. Не забудьте изменить имя пользователя в строке подключения на «roberto» (или другое, которое вы использовали), а значение пароля на пароль, который вы вводили при создании нового пользователя.

Теперь, получив строку подключения к MongoDB, вы сможете развернуть и протестировать приложение Express.js из главы 13.

Приложение D

Рецепт пиццы

Обычно при приготовлении пиццы используется несколько ингредиентов. Каждый из них добавляется в определенный момент в процессе приготовления. Ингредиенты для пиццы можно разделить на три категории:

- ингредиенты для теста;
- начинка;
- специи/соусы.

На первый взгляд тесто приготовить проще простого, достаточно смешать воду и муку, разве не так? Нет, не так! Как оказывается, тесто для пиццы – самая важная и сложная часть. Оно может иметь самые разные:

- форму;
- толщину;
- цвет;
- мягкость
- и другие характеристики (например, бортик с начинкой из плавленого сыра, что, впрочем, у итальянцев считается чуть ли не богохульством).

Чтобы приготовить тесто для одной пиццы по этому рецепту, вам понадобятся следующие ингредиенты:

- 2 стакана муки (250 г);
- 1 чайная ложка дрожжей (5 г);
- 1/2 стакана чуть теплой воды (120 мл);
- щепотка соли;
- 1/2 столовой ложки сахара (6 г);
- 1/2 столовой ложки оливкового масла (7 мл).

Размешайте дрожжи, сахар и ложку муки в четверти стакана теплой воды. Тщательно перемешайте и дайте подняться при комнатной температуре в течение получаса.

Смешайте соль, оливковое масло, смесь воды и дрожжей, а также оставшиеся четверть стакана воды и муку. Все это хорошо перемешайте. Месите тесто

в течение приблизительно 10 минут. Месить можно вручную или в машине, если она у вас есть.

Положите горсть муки в чистую кастрюлю. Равномерно распределите муку, а затем переложите тесто в кастрюлю.

Включите духовку на разогрев до максимальной температуры.

> **ВНИМАНИЕ.** Если вы не уверены, как поведет себя ваша духовка при максимальной температуре, поставьте ее на 220 °C. Некоторые духовки могут вести себя непредсказуемо при максимальном разогреве – мы не хотим, чтобы вы сожгли кухню.

Пока духовка нагревается, приготовьте противень. Отрежьте кусок пергаментной бумаги (бумага для выпечки) по размеру противня. Уложите ее на противень. Поставьте кастрюлю и противень с пергаментной бумагой в духовку по отдельности. Когда тесто хорошо прогреется, можно продолжить готовить его.

Не мешая тесто, растяните его по размеру листа пергаментной бумаги, а затем положите на бумагу. Смажьте оливковым маслом. Теперь положите на него начинку. Когда дело доходит до начинки, можете изобретать до бесконечности, но вот как готовится пицца в пиццерии тетушки Марии. Сначала идет знаменитый томатный соус тетушки Марии.

> **ПРИМЕЧАНИЕ.** Мы не будем рассказывать, как готовит соус тетушка Мария. Это семейный секрет, который никогда не будет раскрыт! Вы же можете использовать обычный томатный соус.

В центр положите несколько кусочков сыра моцарелла, а затем по кругу несколько кусочков пеперони, оливок и немного орегано.

> **ВНИМАНИЕ.** Не кладите слишком много ингредиентов на пиццу!

Выньте разогретый противень и переложите пиццу на пергаментную бумагу. Быстро поставьте противень обратно в духовку.

Выпекайте пиццу 5–10 минут. Правильное время зависит от вашей духовки, но обращайте внимание на края пиццы, они должны быть золотистыми или коричневатыми. Если хотите, налейте немного оливкового масла.

Предметный указатель

W

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.a-planeta.ru**.

Оптовые закупки: тел**. +7(499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Слободан Стоянович
Александар Симович

Бессерверные приложения на JavaScript

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100$^{1}/_{16}$. Печать цифровая.
Усл. печ. л. 54,6. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**