МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Компоненти програмної інженерії Архітектура програмного забезпечення

Лабораторний практикум

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за спеціальністю 121 Інженерія програмного забезпечення

Укладач: Б. Я. Корнієнко

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2023

Рецензент *Ладієва Л. Р.*, канд. техн. наук, доцент,
доцент кафедри технічних та програмних засобів автомаитзації,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Відповідальний
редактор *Ролік О.І.,* докт. техн. наук, професор

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № _7_ від _27_ 04_____2023 р.)
за поданням Вченої ради Факультету інформатики та обчислювальної техніки
(протокол № _11__ від _24_04_ 2023 р.)

Навчальний посібник розрахований на одержання студентами практичних навичок з архітектури програмного забезпечення і складається з 6 лабораторних робіт, що охоплюють питання клієнт-серверної архітектури, веб-сервісів та сервіс-орієнтованої архітектури. В навчальному посібнику приділено увагу основним підходами та платформам при створенні веб-додатків та веб-служб.

Містить теоретичні положення, порядок виконання лабораторних робіт, завдання та список літератури з дисципліни «Компоненти програмної інженерії. Частина 3. Архітектура програмного забезпечення».

Для студентів всіх форм навчання, які навчаються за спеціальністю 121 «Інженерія програмного забезпечення» факультету інформатики та обчислювальної техніки КПІ ім. Ігоря Сікорського.

Реєстр. № НП 22/23-656. Обсяг 2,4 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

Вступ

Останнім часом виникла потреба інтеграції та взаємодії додатків у рамках сукупності великої кількості інформаційних систем підприємства чи кількох підприємств. Спостерігаються тенденції переходу до бізнесу реального часу та створення систем розширеного підприємства, що поєднує саме підприємство, його постачальників, партнерів та клієнтів у єдину систему. Тому потрібна взаємодія програм як в одній інформаційній системі, так і між системами окремих учасників бізнес-процесу.

Вихід із становища, що активно розвивається провідними постачальниками інформаційних технологій, такими як IBM, Microsoft, Oracle та іншими, полягає в переході від централізованої інфраструктури інформаційних технологій і замкнутого на собі функціоналу прикладних систем до архітектури, що забезпечує можливості швидкого створення нових систем з набору доступних сервісів, тобто переході на сервісні моделі взаємодії між додатками загальної системи в рамках сервіс-орієнтованої архітектури (Service-oriented Architecture SOA) та її реалізації за допомогою модельно-орієнтованих архітектурних рішень (Model-driven Architecture MDA).

Сервіс-орієнтована архітектура (SOA) є стилем архітектури програмного забезпечення, яка забезпечує використання інформаційних технологій у вигляді модульних бізнес-сервісів для досягнення конкретних бізнес-цілей. Архітектура SOA — концептуальна архітектура бізнесу, де бізнес-функціональність, або логіка додатків стає доступною для користувачів SOA, як і безліч сервісів, що багаторазово спільно використовуються в IT-мережі. Сервіси в SOA відповідають модулям бізнес-процесу або модулям функціональності програми з відкритими інтерфейсами, що викликаються за допомогою повідомлень.

Основна мотивація появи SOA - бажання індустрії програмування замінити "ручне" кодування програм "від і до" на "промислове" складання додатків із "стандартних комплектуючих", як це відбувається в автомобільній або інших "традиційних" галузях промисловості. Компоненти програми можуть розміщу-

ватися на різних вузлах мережі та являти собою незалежні, слабко пов'язані, замінювані сервіси-додатки.

Для бізнесу SOA означає прискорене задоволення потреб клієнтів, реальну гнучкість бізнесу, швидкий час виходу на ринок, простоту співробітництва та низьку вартість бізнесу. Для IT-організацій SOA - це підвищення продуктивності, зниження витрат на IT-рішення за рахунок прискорення розробки додатків, повторне використання сервісів, покращення якості додатків, і загалом швидке реагування на запити бізнес-клієнтів для модифікації системи. Додатково до цього є можливість використання сервісів незалежних постачальників, що забезпечує ще більшу цінність SOA.

В навчальному посібнику викладені завдання та приклади виконання лабораторних робіт. Комплекс лабораторних робіт призначений для ознайомлення, вивчення та практичного застосування сучасних технологій та мов програмування з розробки веб-додатків та веб-служб з використанням різних платформ та підходів. Для успішного виконання лабораторних робіт студентам необхідно мати базові знання та навички роботи у середовищах розробки, програмування мовою C#, а також базові знання про веб-застосунки.

В процесі виконання лабораторних робіт студенти отримають знання про клієнт-серверні архітектури, веб-сервіси, сервіс-орієнтовану архітектуру. Ознайомляться з роботою та придбають практичні навички у роботі з .NET фреймворк у Visual Studio.

Виконання лабораторних робіт сприяє практичному закріпленню теоретичних знань, пов’язаних із завданнями реалізації веб-застосунків різними засобами. Середовищем розробки для виконання лабораторних робіт є Visual Studio.

Лабораторна робота 1

Розробка .asmx веб-сервісу. Тестування сервісу.

Мета роботи: здобути практичні навички у створені .asmx веб-сервісу та його тестування.

Теоретичні положення

Веб-сервіс - ідентифікована веб-адресою програмна система зі стандартизованими інтерфейсами. Веб-служби можуть взаємодіяти один з одним і зі сторонніми додатками за допомогою повідомлень, заснованих на певних протоколах.

Термін «веб-сервіс» введено організацією W3C і застосовується до багатьох різних систем, але в основному термін стосується клієнтів та серверів, що взаємодіють за допомогою повідомлень протоколу SOAP. В обох випадках припускається, що існує також опис доступних операцій у форматі WSDL. Хоча наявність цього опису не є вимогою SOAP, а радше передумовою для автоматичного генерування коду на платформах Java та .NET на стороні клієнта.

SOAP (англ. Simple Object Access Protocol) — протокол обміну структурованими повідомленнями в розподілених обчислювальних системах, який базується на форматі XML.

Спочатку SOAP призначався, в основному, для реалізації віддаленого виклику процедур (RPC), а назва була абревіатурою: Simple Object Access Protocol — простий протокол доступу до об'єктів. Зараз протокол використовується для обміну повідомленнями в форматі XML, а не тільки для виклику процедур. SOAP є розширенням мови XML-RPC.

Повідомлення SOAP - це XML-документ, що містить такі елементи:

• Envelope - кореневий елемент, який ідентифікує документ XML як повідомлення SOAP;

• Header - необов’язковий елемент, який містить інформацію про конкретну програму, таку як дані автентифікації. Якщо присутній елемент Header, він повинен бути першим дочірнім елементом Envelope;

• Body - необхідний елемент, який містить повідомлення SOAP, призначене для одержувача;

• Fault - необов’язковий елемент, який використовується для позначення повідомлень про помилки. Якщо присутній елемент Fault, він повинен бути дочірнім елементом Body.

SOAP можна використовувати з будь-яким протоколом прикладного рівня: SMTP, FTP, HTTP та інші. Найчастіше SOAP використовується разом з HTTP. SOAP є одним зі стандартів, на яких ґрунтується технологія веб-сервісів.

Приклад SOAP POST повідомлення:

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Приклад SOAP-запиту на сервер інтернет-магазину:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>12345</productID>
```

```
    </getProductDetails>
  </soap:Body>
 </soap:Envelope>
```

Приклад відповіді:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productID>12345</productID>
        <productName>Склянка ребриста</productName>
        <description>Склянка ребриста. 200 мл.</description>
        <price>9.95</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
 </soap:Envelope>
```

ASP.NET Web Services (ASMX) надають можливість створювати вебсервіси, які відправляють повідомлення по протоколу HTTP за допомогою протокола SOAP. Споживачі служби ASMX не повинні знати нічого про платформу, об'єктні моделі або мову програмування, що використовується для реалізації служби. Їм потрібно тільки розібратися, як відправляти і отримувати повідомлення SOAP. SOAP може працювати з багатьма транспортними протоколами, включаючи HTTP, SMTP, TCP та UDP. Однак служба ASMX може працювати лише через HTTP. Платформа Xamarin підтримує стандартні реалізації SOAP 1.1 через HTTP, і це включає підтримку багатьох стандартних конфігурацій служби ASMX.

Цей зразок включає мобільні програми, що працюють на фізичних або емуляційних пристроях, і службу ASMX, яка надає методи отримання, додавання, редагування та видалення даних.

Створення TodoService proxy

Проксі-клас, який називається TodoService, розширює SoapHttpClientProtocol та забезпечує методи зв'язку зі службою ASMX через HTTP. Проксі-сервер генерується шляхом додавання веб-посилання до кожного конкретного проекту платформи у Visual Studio 2019 або Visual Studio 2 017. Веб-посилання генерує методи та події для кожної дії, визначеної в документі WSDL (Мова опису веб-служб) служби.

Наприклад, дія служби GetTodoItems призводить до методу GetTodoItemsAsync та події GetTodoItemsCompleted у проксі. Сформований метод має тип повернення void і викликає дію GetTodoItems над батьківським класом SoapHttpClientProtocol. Коли викликаний метод отримує відповідь від служби, він запускає подію GetTodoItemsCompleted і надає дані відповіді у властивості Result події.

Створення реалізації ISoapService

Щоб спільний міжплатформенний проект міг працювати зі службою, зразок визначає інтерфейс ISoapService, який відповідає моделі асинхронного програмування Task у C #. Кожна платформа реалізує ISoapService для викриття конкретного проксі-сервера. Зразок використовує об'єкти TaskCompletionSource, щоб виставити проксі як асинхронний інтерфейс завдання. Детальна інформація про використання TaskCompletionSource міститься у реалізаціях кожного типу дій у розділах нижче.

Зразок SoapService:

Застосовує TodoService як екземпляр класу. Створює колекцію під назвою "Елементи" для зберігання об'єктів TodoItem. Вказує власну кінцеву точку для необов’язкової властивості Url на TodoService

```
public class SoapService : ISoapService
{
```

```
    ASMXService.TodoService todoService;
    public List<TodoItem> Items { get; private set; } = new List<TodoItem>();

    public SoapService ()
    {
        todoService = new ASMXService.TodoService ();
        todoService.Url = Constants.SoapUrl;
    }
}
```

Створення DTO

Зразок програми використовує клас TodoItem для моделювання даних. Щоб зберегти елемент TodoItem у веб-службі, його потрібно спочатку перетворити на тип TodoItem, згенерований проксі-сервером. Це досягається методом ToASMXServiceTodoItem, як показано в наступному прикладі коду:

```
ASMXService.TodoItem ToASMXServiceTodoItem (TodoItem item)
{
    return new ASMXService.TodoItem {
        ID = item.ID,
        Name = item.Name,
        Notes = item.Notes,
        Done = item.Done
    };
}
```

Цей метод створює новий екземпляр ASMService.TodoItem і встановлює кожну властивість однаковою властивістю з екземпляра TodoItem. Подібним чином, коли дані отримуються з веб-служби, вони повинні бути перетворені з генерованого проксі-сервером типу TodoItem у екземпляр TodoItem. Це досягається методом FromASMXServiceTodoItem, як показано в наступному прикладі коду:

```
static TodoItem FromASMXServiceTodoItem (ASMXService.TodoItem item)
{
   return new TodoItem {
      ID = item.ID,
      Name = item.Name,
      Notes = item.Notes,
      Done = item.Done
   };
}
```

Цей метод отримує дані із типу TodoItem, згенерованого проксі-сервером, і встановлює їх у новоствореному екземплярі TodoItem.

Отримання даних

Інтерфейс ISoapService очікує, що метод RefreshDataAsync поверне завдання із колекцією елементів. Однак метод TodoService.GetTodoItemsAsync повертає void. Щоб задовольнити шаблон інтерфейсу, потрібно викликати GetTodoItemsAsync, дочекатися запуску події GetTodoItemsCompleted та заповнити колекцію. Це дозволяє повернути дійсну колекцію в інтерфейс користувача.

У наведеному нижче прикладі створюється новий TaskCompletionSource, починається виклик асинхронізації в методі RefreshDataAsync і очікується завдання, надане TaskCompletionSource. Коли викликається обробник події TodoService_GetTodoItemsCompleted, він заповнює колекцію Items і оновлює TaskCompletionSource:

```
public class SoapService : ISoapService
{
   TaskCompletionSource<bool> getRequestComplete = null;
   public SoapService()
   {
      todoService.GetTodoItemsCompleted += TodoService_GetTodoItemsCompleted;
   }

   public async Task<List<TodoItem>> RefreshDataAsync()
```

```
        {
            getRequestComplete = new TaskCompletionSource<bool>();
            todoService.GetTodoItemsAsync();
            await getRequestComplete.Task;
            return Items;
        }

        private void TodoService_GetTodoItemsCompleted(object sender, ASMX-Service.GetTodoItemsCompletedEventArgs e)
        {
            try
            {
                getRequestComplete = getRequestComplete ?? new TaskCompletionSource<bool>();
                Items = new List<TodoItem>();
                foreach (var item in e.Result)
                {
                    Items.Add(FromASMXServiceTodoItem(item));
                }
                getRequestComplete?.TrySetResult(true);
            }
            catch (Exception ex)
            {
                Debug.WriteLine(@"\t\tERROR {0}", ex.Message);
            }
        }
    }
```

Створеннння та редагування даних

Під час створення або редагування даних потрібно реалізувати метод ISoapService.SaveTodoItemAsync. Цей метод виявляє, чи є TodoItem новим або оновленим елементом, і викликає відповідний метод для об'єкта todoService. Обробники подій CreateTodoItemCompleted та EditTodoItemCompleted також повинні бути реалізовані, щоб ви знали, коли служба todoService отримала

відповідь від служби ASMX (їх можна об'єднати в один обробник, оскільки вони виконують одну і ту ж операцію). Наступний приклад демонструє реалізації інтерфейсу та обробника подій, а також об’єкт TaskCompletionSource, який використовується для асинхронної роботи:

```
public class SoapService : ISoapService
{
   TaskCompletionSource<bool> saveRequestComplete = null;

   public SoapService()
   {
      todoService.CreateTodoItemCompleted += TodoService_SaveTodoItemCompleted;
      todoService.EditTodoItemCompleted += TodoService_SaveTodoItemCompleted;
   }

   public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
   {
      try
      {
         var todoItem = ToASMXServiceTodoItem(item);
         saveRequestComplete = new TaskCompletionSource<bool>();
         if (isNewItem)
         {
            todoService.CreateTodoItemAsync(todoItem);
         }
         else
         {
            todoService.EditTodoItemAsync(todoItem);
         }
         await saveRequestComplete.Task;
      }
      catch (SoapException se)
      {
         Debug.WriteLine("\t\t{0}", se.Message);
      }
```

```
            catch (Exception ex)
            {
                Debug.WriteLine("\t\tERROR {0}", ex.Message);
            }
        }

        private void TodoService_SaveTodoItemCompleted(object sender, System.ComponentModel.AsyncCompletedEventArgs e)
        {
            saveRequestComplete?.TrySetResult(true);
        }
    }
```

Видалення даних

Видалення даних вимагає подібної реалізації. Визначте TaskCompletionSource, реалізуйте обробник подій та метод ISoapService.DeleteTodoItemAsync:

```
    public class SoapService : ISoapService
    {
        TaskCompletionSource<bool> deleteRequestComplete = null;

        public SoapService()
        {
            todoService.DeleteTodoItemCompleted += TodoService_DeleteTodoItemCompleted;
        }

        public async Task DeleteTodoItemAsync (string id)
        {
            try
             {
                deleteRequestComplete = new TaskCompletionSource<bool>();
                todoService.DeleteTodoItemAsync(id);
                await deleteRequestComplete.Task;
            }
            catch (SoapException se)
```

```
            {
                Debug.WriteLine("\t\t{0}", se.Message);
            }
            catch (Exception ex)
            {
                Debug.WriteLine("\t\tERROR {0}", ex.Message);
            }
        }

        private void TodoService_DeleteTodoItemCompleted(object sender, System.ComponentModel.AsyncCompletedEventArgs e)
        {
            deleteRequestComplete?.TrySetResult(true);
        }
    }
```

Можна використовувати тест веб-продуктивності для тестування веб-служб. За допомогою параметрів Вставити запит та Вставити запит веб-сервісу можна налаштувати окремі запити в Редакторі тесту веб-продуктивності для пошуку сторінок веб-служби. Як правило, ці сторінки не відображаються у веб-програмі. Тому необхідно налаштувати запит, щоб отримати доступ до цих сторінок.

Створення простої веб-служби

1. Для тестування ви можете скористатися власною веб-службою або скористатися базовим шаблоном веб-служби (ASMX), що входить до складу Visual Studio. Щоб створити просту веб-службу за допомогою цього шаблону

2. У Visual Studio створіть новий проект, використовуючи шаблон веб-програми ASP.NET (.NET Framework), і виберіть "Порожній шаблон" за запитом. Введіть ім'я та створіть проект.

3. У Провіднику рішень клацніть правою кнопкою миші вузол проекту, виберіть Додати> Новий елемент, а потім виберіть Веб-служба (ASMX). Додайте веб-службу.

4. Відкрийте WebService1.asmx і замініть веб-метод HelloWorld за замовчуванням таким кодом.

```
public string HelloWorld(string str)     {
  return "Hello, " + str;
}
```

Установка та завантаження компоненту для тестування

1. Якщо у вас ще не встановлений компонент веб-продуктивності та інструментів тестування навантаження, вам доведеться встановити його за допомогою інсталятора Visual Studio.

2. Відкрийте інсталятор Visual Studio у меню «Пуск» Windows. Ви також можете отримати до нього доступ у Visual Studio з нового діалогового вікна проекту або вибравши Інструменти> Отримати інструменти та функції на панелі меню.

3. У програмі інсталятора Visual Studio виберіть вкладку Окремі компоненти та прокрутіть вниз до розділу Налагодження та тестування. Виберіть веб-продуктивність та інструменти тестування навантаження.

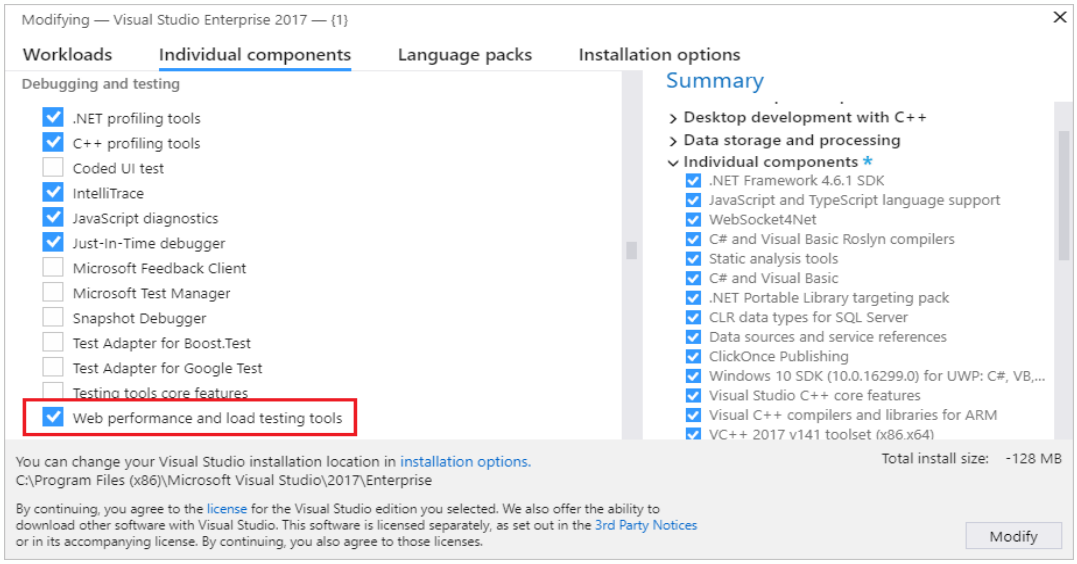

Рис. 1.1 – Встановлення веб-служб

4. Виберіть кнопку «Змінити». Інстальовано компонент веб-продуктивності та інструментів тестування навантаження.

Створення проекту для тестування веб-сервісу

Веб-тест вимагає шаблону проекту «Веб-продуктивність та завантаження тестового проекту». Ви також можете створити проект тесту навантаження Visual Basic, якщо хочете.

1. Відкрийте Visual Studio. Якщо ви використовуєте зразок шаблону веб-служби (ASMX), ви можете додати проект веб-тесту до того ж рішення.

2. У вікні запуску виберіть Створити новий проект.

3. На сторінці Створення нового проекту введіть веб-тест у вікно пошуку, а потім виберіть шаблон веб-продуктивності та завантаження тестового проекту для C #. Виберіть Далі.

4. Введіть ім'я проекту, якщо ви не хочете використовувати ім’я за замовчуванням, а потім виберіть Створити.

5. Visual Studio створює проект і відображає файли в Провіднику рішень. Спочатку проект містить один веб-файл тесту з іменем WebTest1.webtest.

Тестування веб-сервісу

1. Запустіть свою веб-службу та, якщо потрібно, виберіть Зупинити, щоб призупинити службу.

2. У проекті веб-тесту відкрийте WebTest1.webtest, який відкриє редактор тесту веб-продуктивності. У редакторі тесту клацніть правою кнопкою миші тест веб-продуктивності та виберіть Додати запит веб-служби.

3. У властивості Url нового запиту введіть ім'я веб-служби, наприклад https: // localhost: 44318 / WebService1.asmx.

4. Для веб-служби відкрийте окремий сеанс браузера та введіть URL-адресу сторінки .asmx на панелі інструментів «Адреса». У верхній частині веб-сторінки виберіть метод, який ви хочете перевірити, і вивчіть повідомлення

SOAP. (У прикладі веб-служби методом є HelloWorld.) Коли ви відкриваєте метод, ви бачите, що він містить SOAPAction.

5. У Редакторі тесту веб-продуктивності клацніть правою кнопкою миші на запиті та виберіть Додати заголовок, щоб додати новий заголовок. У властивості Name введіть SOAPAction. У властивості Value введіть значення, яке ви бачите в SOAPAction, наприклад http://tempuri.org/HelloWorld.

6. Розгорніть вузол URL-адреси в тестовому редакторі, виберіть вузол «Текстовий рядок» і у властивості «Тип вмісту» введіть значення text / xml.

7. Поверніться до браузера на кроці 4, виберіть XML-частину запиту SOAP на сторінці опису веб-служби та скопіюйте її в буфер обміну. Вміст XML нагадує такий приклад:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <HelloWorld xmlns="http://tempuri.org/">
      <str>string</str>
    </HelloWorld>
  </soap:Body>
</soap:Envelope>
```

1. Поверніться до редактора тесту веб-продуктивності, а потім виберіть еліпсис (...) у властивості "Рядок". Вставте вміст буфера обміну у властивість.

2. Замініть усі значення заповнювачів у XML дійсними значеннями, щоб тест міг пройти. У попередньому прикладі ви б замінили екземпляр рядка на ім'я.

3. Клацніть правою кнопкою миші запит веб-служби та виберіть Додати URL-адресу QueryString Parameter.

4. Призначте параметру рядка запиту ім'я та значення. У попередньому прикладі ім'я є op, а значення HelloWorld. Це визначає операцію веб-служби, яку потрібно виконати. ПримечаниеВи можете використовувати прив'язку даних у тілі SOAP, щоб замінити будь-яке значення заповнювача значеннями, прив'язаними до даних, використовуючи синтаксис {{DataSourceName.TableName.ColumnName}}.

5. Запустіть тест. На верхній панелі засобу перегляду результатів тесту веб-продуктивності виберіть запит веб-служби. На нижній панелі виберіть вкладку Веб-браузер. Відобразиться XML, який повертає веб-служба, та результати будь-яких операцій. Шукайте результати для запиту веб-служби.

Завдання на лабораторну роботу

1. Створити простий веб-сервіс .asmx з одним методом, який приймає на вхід параметри та виводить їх на екран.

2. Протестувати даний веб-сервіс за допомогою компонента веб-продуктивності та інструментів тестування навантаження.

Контрольні питання

1. Що таке веб-сервіс? Які протоколи для передачі даних використовуються у бев-сервісах?

2. Що таке протокол SOAP? Яка у нього структура? Які він надає переваги у взаємодії сервісу з клієнтами?

3. За допомогою якого протоколу ASP.NET Web Services (ASMX) взаємодіє з клієнтами?

4. Як створити .asmx веб-сервіс?

5. З яких етапів складається процес тестування веб-сервісу?

Лабораторна робота 2

Розробка клієнта (веб-серверної програми) для .asmx веб-сервісу. Тестування програми.

Мета роботи: здобути практичні навички у створені клієнта для .asmx веб-сервісу та його тестування.

Теоретичні положення

Архітектура клієнт-сервер є одним із архітектурних шаблонів програмного забезпечення та є домінуючою концепцією у створенні розподілених мережних застосунків і передбачає взаємодію та обмін даними між ними. Вона передбачає такі основні компоненти:

• набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;

• набір клієнтів, які використовують сервіси, що надаються серверами;

• мережа, яка забезпечує взаємодію між клієнтами та серверами.

Сервери є незалежними один від одного. Клієнти також функціонують паралельно і незалежно один від одного. Немає жорсткої прив'язки клієнтів до серверів. Більш ніж типовою є ситуація, коли один сервер одночасно обробляє запити від різних клієнтів; з іншого боку, клієнт може звертатися то до одного сервера, то до іншого. Клієнти мають знати про доступні сервери, але можуть не мати жодного уявлення про існування інших клієнтів.

Створення веб-сервісу та клієнту для нього

Створіть в Visual Studio новий проект "ASP.NET Empty Web Application" або "ASP.NET Web Service Application" з ім'ям FinReportWebService. Додайте в нього два файли: FinReport.asmx і FinReportService.cs, причому FinReport.asmx додайте як Text File, а не Web Service, щоб це був поодинокий файл.

FinReport.asmx

```
<%@ Class="FinReportWebService.FinReportService" %>
```

FinReportService.cs

```
using System;
```

```
using System.Web.Services;

namespace FinReportWebService{

    public class FinReportService {
        [WebMethod]
        public int[] GetReportIdArray(DateTime dateBegin, DateTime dateEnd){
            int[] array = new int[] {357, 358, 360, 361};
            return array;
        }

        [WebMethod]
        public FinReport GetReport(int reportID){
            FinReport finReport = new FinReport(){
                ReportID = reportID,
                Date = new DateTime(2015, 03, 15),
                Info = "Some info"
            };

            return finReport;
        }
    }

    public class FinReport {
        public int ReportID { get; set; }
        public DateTime Date { get; set; }
        public string Info { get; set; }
    }
}
```

Натисніть F5 для запуску веб-сервера і відкрийте в браузері FinReport.asmx, ви повинні побачити

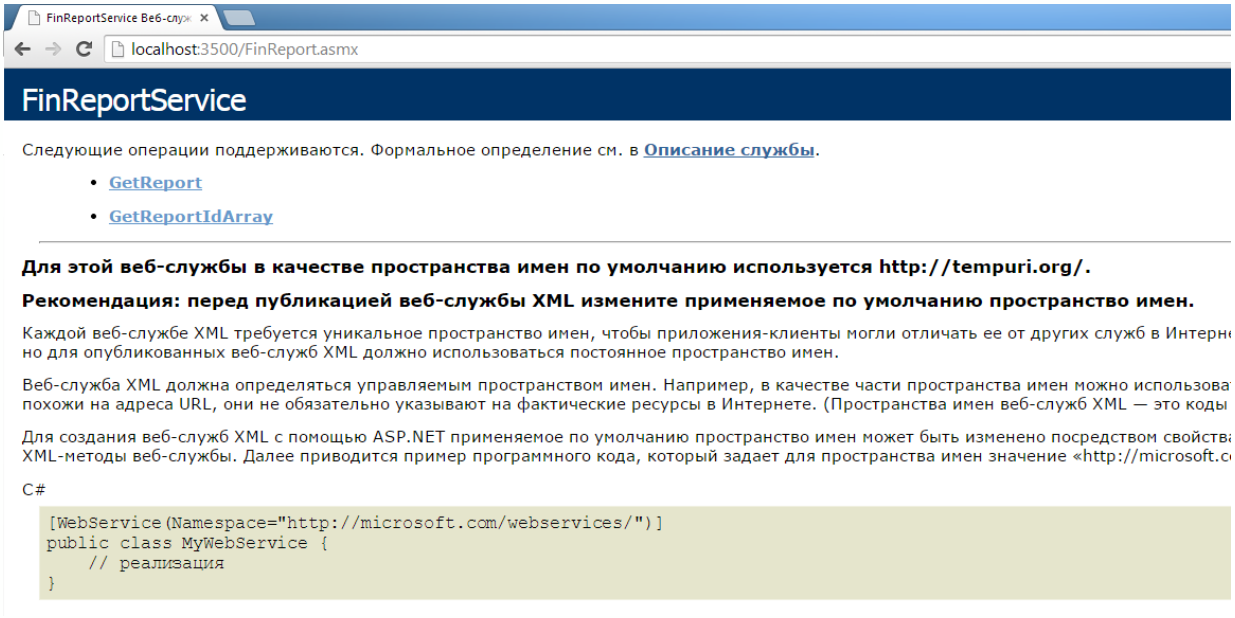

Рис. 2.1. Запуск веб-серверу

Тепер розберемо по порядку. Веб-сервіс представлений одним звичайним класом з однієї лише обов'язкової особливістю - деякі його методи позначені спеціальним атрибутом [WebMethod]. Такі методи класу стають веб-методами веб-сервісу з відповідною сигнатурою виклику. Цей клас повинен володіти конструктором за замовчуванням. При кожному новому запиті IIS його створює його дефолтним конструктором і викликає відповідний метод.

Друга обов'язкова частина мінімальної конструкції - це файл з розширенням asmx, всередині якого необхідно вказати цей клас.

Цікаво порівняти цей вручну створений asmx файл з тим, який створить Visual Studio. Припустимо, що ми хочемо зробити ще один веб-сервіс, який повертає курс обміну валют. Додайте через меню Add New Item файл ExchangeRate.asmx з типом Web Service.

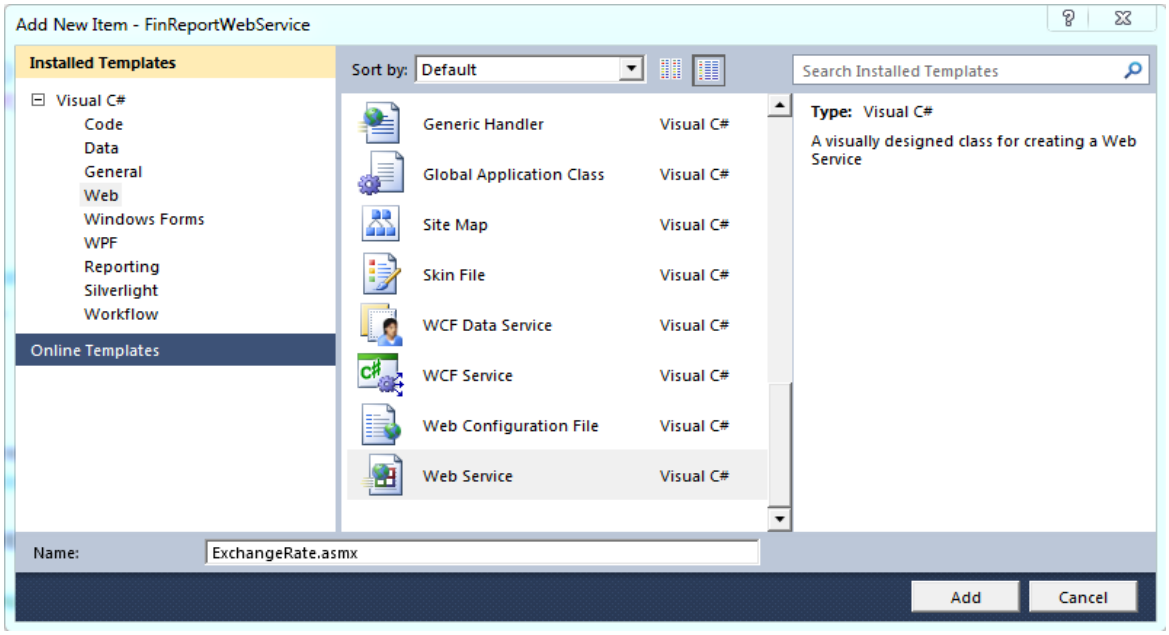

Рис. 2.2. Створення веб-серверу в Visual Studio

Натиснувши один-два рази на F7, можна побачити наступне:

<%@ WebService Language="C#" CodeBehind="ExchangeRate.asmx.cs" Class="FinReportWebService.ExchangeRate" %>

Оператор Language = "C #» є рудиментарним, і потрібен тільки якщо ви будете писати вихідний код безпосередньо всередині asmx файлу. Такий код компілюватиметься динамічно. В цілому динамічна компіляція веб-сервісу - не дуже гарна практика. Оператор CodeBehind = «ExchangeRate.asmx.cs» просто пов'язує два файли на рівні Visual Studio.

У цьому прикладі той же самий веб-сервіс реалізований більш коректним чином. У файлі FinReportService.cs вміст замініть на наступний вихідний код:

FinReportService.cs

```
using System;
using System.Web.Services;
using System.Xml.Serialization;

namespace FinReportWebService{
```

```
        [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
        [WebService(Description = "Фин. отчеты", Namespace = XmlNS)]
        public class FinReportService : WebService{
            public const string XmlNS = "http://asmx.habrahabr.ru/";

            [WebMethod(Description = "Получение списка ID отчетов по периоду")
]
            pub-
lic GetReportIdArrayResult GetReportIdArray(GetReportIdArrayArg arg){
                return new GetReportIdArrayResult(){
                    ReportIdArray = new int[] {357, 358, 360, 361}
                };
            }

            [WebMethod(Description = "Получение отчета по ID")]
            public GetReportResult GetReport(GetReportArg arg){
                return new GetReportResult(){
                    Report = new FinReport{
                        ReportID = arg.ReportID,
                        Date = new DateTime(2015, 03, 15),
                        Info = getReportInfo(arg.ReportID)
                    }
                };
            }

            private string getReportInfo(int reportID){
                return "ReportID = " + reportID;
            }
        }
```

```
//    [Serializable]
//    [XmlType(Namespace = FinReportService.XmlNS)]
    public class FinReport {
        public int ReportID { get; set; }
        public DateTime Date { get; set; }
        public string Info { get; set; }
    }

    public class GetReportIdArrayArg {
        public DateTime DateBegin { get; set; }
        public DateTime DateEnd { get; set; }
    }

    public class GetReportIdArrayResult {
        public int[] ReportIdArray { get; set; }
    }

    public class GetReportArg {
        public int ReportID { get; set; }
    }

    public class GetReportResult {
        public FinReport Report { get; set; }
    }
}
```

Атрибут [WebServiceBinding (ConformsTo = WsiProfiles.BasicProfile1_1)] означає, що веб-сервіс перевіряється на відповідність специфікації WSI Basic Profile 1.1. Наприклад, згідно з нею заборонена перевантаження імені операції, або застосування атрибута [SoapRpcMethod]. Такі порушення будуть приводити до помилки веб-сервісу «Служба" FinReportWebService.FinReportService "не відповідає специфікації Simple SOAP Binding Profile Version 1.0.». При відсутності цього атрибуту порушення будуть приводити тільки до попередження «Ця веб-служба не відповідає вимогам WS-I Basic Profile v1.1.». У загальному випадку рекомендується додавати цей атрибут, що забезпечує більшу сумісність.

Атрибут [WebService (Description = «Фін. Звіти», Namespace = XmlNS)] має всього три властивості:

• Namespace - дефолтний ХМЛ неймспейс - вказувати обов'язково

• Description - опис веб-сервісу, що відображається в браузері

•Name - ім'я веб-сервісу (по дефолту береться ім'я класу)

Успадкування від класу WebService дає доступ до об'єктів HttpContext, HttpSessionState і деяким іншим, що в деяких випадках може бути корисно.

В атрибуті [WebMethod (Description = «Отримання звіту по ID»)] як правило вказують тільки Description, який описує веб-метод в браузері, інші властивості використовуються рідко.

Вхідні параметри і повертаються значення я особисто рекомендую инкапсулировать в спеціальні класи. Наприклад, я їх називаю, додаючи суфікси -Arg і -Result до назви методу, що означає аргумент і результат. У цьому прикладі для спрощення вони всі знаходяться в одному файлі FinReportService.cs, але в реальних проектах кожен з них я розміщую в окремому файлі в спеціальній папці типу FinReportServiceTypes. Також їх зручно успадковувати від загальних класів.

За ідеєю, до всіх власним класах веб-методах необхідно вказувати атрибути [Serializable] і [XmlType (Namespace = FinReportService.XmlNS)]. Однак в даному випадку це не обов'язково. Адже якщо проводиться тільки XML-сериализация, то атрибут [Serializable] не потрібен, а XML неймспейс і так за замовчуванням береться з атрибуту [WebService]. Зазначу, що на відміну від WCF в ASMX використовується звичайний XmlSerializer, що дозволяє широко управляти сериализацией за допомогою таких стандартних атрибутів як [XmlType], [XmlElement], [XmlIgnore] і т.д.

Утиліта wsdl.exe є відповідною для asmx технікою споживання SOAP веб-сервісів. За wsdl файлу або посиланням вона генерує проксі-клас - спеціальної клас, максимально спрощує звернення до даного веб-сервісу. Зрозуміло, не важливо на якій технології реалізований сам веб-сервіс, це може бути що завгодно - ASMX, WCF, JAX-WS або NuSOAP. До речі, у WCF аналогічна утиліта називається SvcUtil.exe.

Утиліта розташована в папці C: \ Program Files (x86) \ Microsoft SDKs \ Windows, більш того, вона там представлена в різних версіях, в залежності від версії .net, розрядності, версії windows і visual studio.

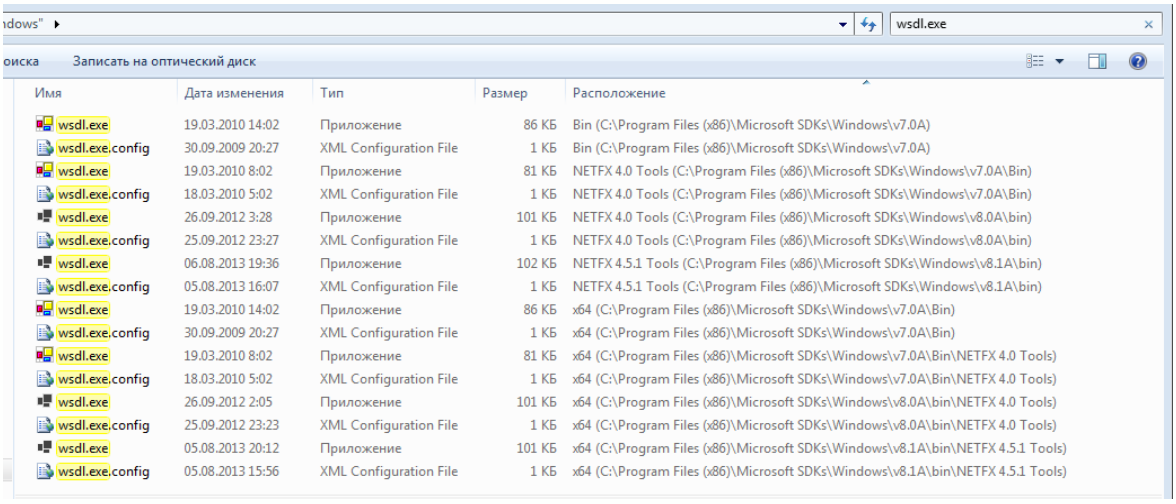

Рис. 2.3. Утиліта wsdl.

приклади використання

wsdl http://192.168.1.101:8080/SomeDir/SomeService?wsdl

wsdl HabraService.wsdl

Давайте зробимо клієнта для FinReportWebService. У поточному або новому рішенні створіть новий Windows Forms проект FinReportWebServiceClient. Додайте в ньому папку ProxyClass, скопіюйте в неї утиліту wsdl.exe і створіть в ній батник GenProxyClass.bat:

wsdl /n:FinReportWebServiceClient.ProxyClass http://localhost:3500/FinReport.asmx?wsdl

pause

За допомогою аргументу /n:FinReportWebServiceClient.ProxyClass ми вказуємо неймспейс для класу. Запустивши його, ви повинні отримати файл FinReportService.cs. Через Solution Explorer - Show All Files, включите всі три файли в СОЛЮШЕН.

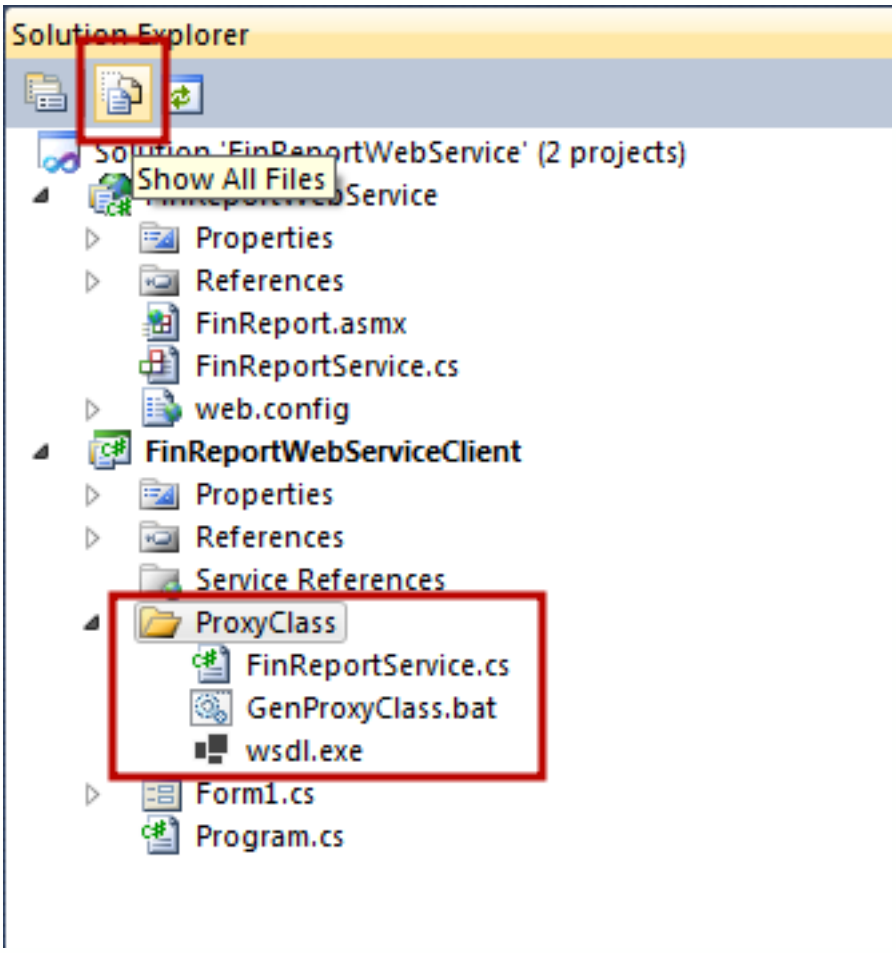

Рис. 2.4. Додавання файлів в Солюшен.

На формі додайте кнопку, а у вихідний код форми наступні три методи:

```
public static FinReportService GetFinReportService(){
    var service = new FinReportService();
    service.Url = "http://localhost:3500/FinReport.asmx";
    service.Timeout = 100 * 1000;
    return service;
}


private void webMethodTest_GetReportIdArray() {
    var service = GetFinReportService();
    var arg = new GetReportIdArrayArg();
    arg.DateBegin = new DateTime(2015, 03, 01);
    arg.DateEnd = new DateTime(2015, 03, 02);

    var result = service.GetReportIdArray(arg);
    Message-
Box.Show("result.ReportIdArray.Length = " + result.ReportIdArray.Length);
}


private void webMethodTest_GetReport() {
    var service = GetFinReportService();
    var arg = new GetReportArg();
    arg.ReportID = 45;

    var result = service.GetReport(arg);
    MessageBox.Show(result.Report.Info);
```

}

Найважливішими властивостями проксі-класу є Url і Timeout, причому таймаут вказується в мілісекундах і 100 секунд це його дефолтний значення. Тепер за допомогою них ви можете протестувати роботу веб-сервісу. Демонстрація роботи подальших прийомів буде показана через виклик методу GetReport і заповнення поля result.Report.Info.

У разі створення проксі-класу по wsdl файлу, який посилається на зовнішні xsd схеми, всі ці схеми необхідно перерахувати в команді:

wsdl /n:MyNamespace HabraService.wsdl Data.xsd Common.xsd Schema.xsd

Однак крім ручного створення проксі-класу Visual Studio дозволяє його створити автоматично. Пункт «Add Service Reference» дозволяє створити проксі-клас по технології WCF, і там же в «Advanced» є кнопка «Add Web Reference», яка створює його вже за технологією ASMX.

Як відомо, wsdl опис веб-сервісу в технології ASMX генерується автоматично. Однак іноді виникає зворотна задача: по даним wsdl файлу розробити відповідний йому веб-сервіс. Вирішується вона за допомогою тієї ж утиліти wsdl.exe. Вона може створити необхідний скелет з класів і вам залишиться тільки реалізувати програмну логіку веб-методів.

Для прикладу візьмемо wsdl нашого веб-сервісу. Збережіть його з браузера як файл FinReport.wsdl або скопіюйте звідси:

FinReport.wsdl

Створіть в солюшене новий порожній web-проект з ім'ям FinReportWebServiceByWsdl. У нього додайте папку ServerClass, в яку необхідно скопіювати файли FinReport.wsdl і wsdl.exe. Створіть в ній батник GenServerClass.bat:

wsdl / server /n:FinReportWebServiceByWsdl.ServerClass FinReport.wsdl

pause

Запустивши його, ви повинні отримати файл FinReportService.cs. Всі чотири файли включите в СОЛЮШЕН.

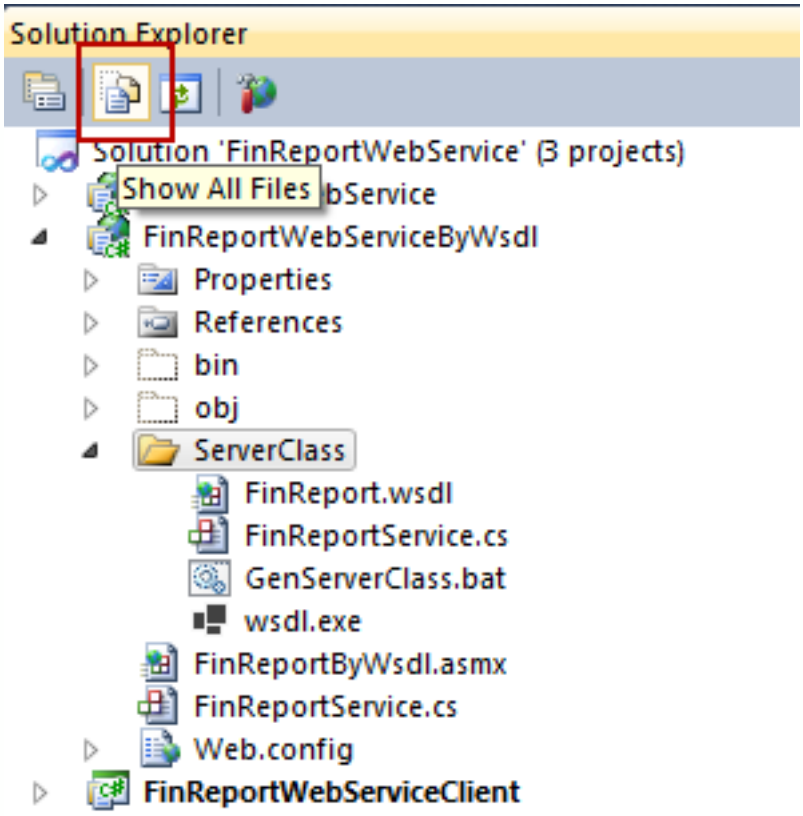

Рис. 2.5. FinReportService.cs.

Отже, як бачимо, єдина відмінність від генерації проксі-класу - це атрибут server. При цьому створюється абстрактний клас успадкованих від WebService з абстрактно описаними веб-методами. Можна від нього успадковуватися, але при цьому все одно доведеться копіювати всі атрибути, тому пропоную зробити наступним чином. Скопіювати визначення класу в новий файл і простір імен, прибрати слово abstract і написати реалізацію методів. Після форматування коду виходить наступний файл

```
using System;
using System.Web.Services;
using System.Web.Services.Description;
using System.Web.Services.Protocols;
using FinReportWebServiceByWsdl.ServerClass;
```

```
namespace FinReportWebServiceByWsdl {

    [WebService(Namespace="http://asmx.habrahabr.ru/")]
    [WebServiceBinding(Name="FinReportServiceSoap", Namespace="http://
asmx.habrahabr.ru/")]
    public class FinReportService : WebService {

        [WebMethod]
        [SoapDocumentMethod("http://asmx.habrahabr.ru/GetReportIdArray",
            RequestNamespace = "http://asmx.habrahabr.ru/",
            ResponseNamespace = "http://asmx.habrahabr.ru/",
            Use = SoapBindingUse.Literal,
            ParameterStyle = SoapParameterStyle.Wrapped)]

        pub-
lic GetReportIdArrayResult GetReportIdArray(GetReportIdArrayArg arg) {
            return new GetReportIdArrayResult();
        }


        [WebMethod]
        [SoapDocumentMethod("http://asmx.habrahabr.ru/GetReport",
            RequestNamespace = "http://asmx.habrahabr.ru/",
            ResponseNamespace = "http://asmx.habrahabr.ru/",
            Use = SoapBindingUse.Literal,
            ParameterStyle = SoapParameterStyle.Wrapped)]

        public GetReportResult GetReport(GetReportArg arg) {
```

```
            return new GetReportResult() {
                Report = new FinReport {
                    ReportID = arg.ReportID,
                    Date = new DateTime(2015, 03, 15),
                    Info = "ByWSDL"
                }
            };
        }
    }
}
```

У цьому коді утиліта явно описала за допомогою атрибутів ті параметри веб-сервісу, які неявно визначалися за замовчуванням. Залишається тільки додати файл FinReportByWsdl.asmx, який буде вказувати на цей новий клас:

<%@ Class="FinReportWebServiceByWsdl.FinReportService" %>

ASMX веб-сервіс може приймати і повертати дані в форматі JSON, що дозволяє реалізувати техніку ajax. Для роботи прикладу в вашому веб-проект повинні бути наступні три файли:

FinReport.asmx - такий же, що і в перших прикладах, всього 1 рядок

<%@ Class="FinReportWebService.FinReportService" %>

FinReportService.cs –код змінюємо на наступний

```
using System;
using System.Text;
using System.Web.Script.Serialization;
using System.Web.Script.Services;
using System.Web.Services;
```

```
using Newtonsoft.Json;

namespace FinReportWebService{
   [ScriptService]
   [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
   [WebService(Description = "Фин. отчеты", Namespace = "http://asmx.habra
habr.ru/")]
   public class FinReportService : WebService{

      [ScriptMethod(ResponseFormat = ResponseFormat.Json)]
      [WebMethod]
      public GetReportResult Method_1_POST_Objects(GetReportArg arg) {
         return getFinReportResult(arg.ReportID, "Method_1_POST_Objects");
      }


      [ScriptMeth-
od(ResponseFormat = ResponseFormat.Json, UseHttpGet = true)]
      [WebMethod]
      public string Method_2_GET(int id){
         var result = getFinReportResult(id, "Method_2_GET");
         string text =  JsonConvert.SerializeObject(result);
         return text;
      }


      [ScriptMethod(ResponseFormat = ResponseFormat.Json)]
      [WebMethod]
      public string Method_3_POST(int id) {
```

```
        var result = getFinReportResult(id, "Method_3_POST");
        JavaScriptSerializer js = new JavaScriptSerializer();
        return js.Serialize(result);
    }


    [ScriptMethod(ResponseFormat = ResponseFormat.Json)]
    [WebMethod]
    public string Method_4_POST_ComplexArg(string json) {
        var arg = JsonConvert.DeserializeObject<GetReportArg>(json);
        var result = getFinReportResult(arg.ReportID, arg.Token + " Получен.")
;
        return JsonConvert.SerializeObject(result);
    }


    [ScriptMethod(ResponseFormat = ResponseFormat.Json)]
    [WebMethod]
    public DateTime Method_5_TransformDate(DateTime dateTime){
        return dateTime.AddYears(-3).AddDays(-5).AddHours(-
2).AddMinutes(6);
    }


    [ScriptMethod(ResponseFormat = ResponseFormat.Json)]
    [WebMethod]
    public void Method_6_POST_NonStandard(int id) {
        var result = getFinReportResult(id, "Method_6_POST_NonStandard, Mo
й текст");
```

```
        string text = JsonConvert.SerializeObject(result);
        byte[] data = Encoding.UTF8.GetBytes(text);

        Context.Response.Clear();
        Context.Response.ContentType = "application/json; charset=utf-8";
        Context.Response.AddHeader("content-length", data.Length.ToString());
        Context.Response.BinaryWrite(data);
        Context.Response.Flush();
    }


    private GetReportResult getFinReportResult(int id, string info) {
        return new GetReportResult() {
            Report = new FinReport() {
                ReportID = id,
                Info = info,
                Date = new DateTime(2015, 03, 15),
            }
        };
    }
}


public class FinReport {
    public DateTime Date { get; set; }
    public string Info { get; set; }
    public int ReportID { get; set; }
}
```

```
    public class GetReportArg {
        public int ReportID { get; set; }
        public string Token { get; set; }
    }

    public class GetReportResult {
        public FinReport Report { get; set; }
    }
}
```

Page.htm – веб-сторінка

```
<!doctype html>
<html>
<head>

<meta charset=utf-8>
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></scri
pt>

<script>
    $(document).ready(function () {
        $("#btn1").click(function () {
            var arg = { arg: { ReportID: 1} };

            $.ajax({
                type: "POST",
                contentType: "application/json; charset=utf-8",
                url: "/FinReport.asmx/Method_1_POST_Objects",
                data: JSON.stringify(arg),
```

```
        dataType: "json",
        success: function (data, status) {
            $("#div1").html(data.d.Report.Info);
        },
        error: function (request, status, error) { alert("Error"); }
    });
});

$("#btn2").click(function () {
    $.ajax({
        type: "GET",
        contentType: "application/json; charset=utf-8",
        url: "/FinReport.asmx/Method_2_GET",
        data: { id: 2 },
        dataType: "json",
        success: function (data, status) {
            $("#div2").html(JSON.parse(data.d).Report.Info);
        },
        error: function (request, status, error) { alert("Error"); }
    });
});

$("#btn3").click(function () {
    $.ajax({
        type: "POST",
        contentType: "application/json; charset=utf-8",
        url: "/FinReport.asmx/Method_3_POST",
        data: '{"id":3}',
        dataType: "json",
```

```
        success: function (data, status) {
            $("#div3").html(JSON.parse(data.d).Report.Info);
        },
        error: function (request, status, error) { alert("Error"); }
    });
});


$("#btn4").click(function () {
    var arg = { ReportID: 4, Token: "Токен метода 4." };
    var argObj = { json: JSON.stringify(arg) };

    $.ajax({
        type: "POST",
        contentType: "application/json; charset=utf-8",
        url: "/FinReport.asmx/Method_4_POST_ComplexArg",
        data: JSON.stringify(argObj),
        dataType: "json",
        success: function (data, status) {
            $("#div4").html(JSON.parse(data.d).Report.Info);
        },
        error: function (request, status, error) { alert("Error"); }
    });
});

$("#btn5").click(function () {
    var now = new Date();
    var arg = { dateTime: now };
```

```
            $.ajax({
                type: "POST",
                contentType: "application/json; charset=utf-8",
                url: "/FinReport.asmx/Method_5_TransformDate",
                data: JSON.stringify(arg),
                dataType: "json",
                success: function (data, status) {
                    var date = new Date(parseInt(data.d.replace("/Date(", "").replace(")/
", ""), 10));
                    $("#div5").html(date.toString());
                },
                error: function (request, status, error) { alert("Error"); }
            });
        });

        $("#btn6").click(function () {
            $.ajax({
                type: "POST",
                contentType: "application/json; charset=utf-8",
                url: "/FinReport.asmx/Method_6_POST_NonStandard",
                data: '{"id":6}',
                dataType: "json",
                success: function (data, status) {
                    $("#div6").html(data.Report.Info);
                },
                error: function (request, status, error) { alert("Error"); }
            });
        });
```

```
    });
</script>

</head>
<body>

    <div id="div1">Div 1</div>
    <div id="div2">Div 2</div>
    <div id="div3">Div 3</div>
    <div id="div4">Div 4</div>
    <div id="div5">Div 5</div>
    <div id="div6">Div 6</div>
    <br/>
    <button id="btn1">Method 1</button>
    <br/>
    <button id="btn2">Method 2</button>
    <br/>
    <button id="btn3">Method 3</button>
    <br/>
    <button id="btn4">Method 4</button>
    <br/>
    <button id="btn5">Method 5</button>
    <br/>
    <button id="btn6">Method 6</button>

</body>
</html>
```

Також в прикладі використовується бібліотека Json.NET aka Newtonsoft.Json.Щоб веб-сервіс міг працювати з JSON, потрібно застосувати 2 нових атрибута:

[ScriptService] - без властивостей, і [ScriptMethod], в якому властивість ResponseFormat відповідає за формат відповіді - JSON або XML, а UseHttpGet - за тип запиту - GET або POST.

У цьому веб-сервісі 6 методів, які демонструють різні способи реалізації ajax.

Метод 1. Як і у 2 прикладі він приймає і повертає об'єкти класів GetReportArg і GetReportResult. У отформатированном вигляді запит і відповідь виглядають наступним чином:

```
{
  "arg": {
    "ReportID": 1
  }
}
```

```
{
  "d": {
    "__type": "FinReportWebService.GetReportResult",
    "Report": {
      "ReportID": 1,
      "Date": "/Date(1426356000000)/",
      "Info": "Method_1_POST_Objects"
    }
  }
}
```

Якщо з аргументом все зрозуміло, то відповідь потрібно прокоментувати. Все JSON-відповіді веб-сервіс з міркувань безпеки загортає в вузол «d». Також можемо бачити назву класу "__type": «FinReportWebService.GetReportResult». А ось формат дати "/ Date (1426356000000) /" є проблемою. Що робити з таким форматом, описано в 5 методі, крім того, подібного формату легко уникнути, як показано далі.

Метод 2. Принципово інший спосіб реалізації. Використовується тип запиту GET, приймає число, повертає json-рядок, а не сам об'єкт і застосовується сторонній серіалайзер. В скрипті аргумент задається як data: {id: 2}, що доповнює URL запиту до виду http: // localhost: 3500 / FinReport.asmx / Method_2_GET? Id = 2, однак можна відразу вказати цей кінцевий URL.


```
{
  "d": "{\"Report\":{\"ReportID\":2,\"Date\":\"2015-03-
15T00:00:00\",\"Info\":\"Method_2_GET\"}}"
}
```


Як бачимо, потрібно зробити парсинг JSON.parse (data.d), щоб отримати вихідний об'єкт виду:


```
{
  "Report": {
    "ReportID": 2,
    "Date": "2015-03-15T00:00:00",
    "Info": "Method_2_GET"
  }
}
```


Зверніть увагу, що дані завдяки бібліотеці Json.NET тут представлена стандартно. Також потрібно відзначити обов'язковість заголовка contentType: «application / json; charset = utf-8 », незважаючи на те, що це GET. Зроблено це

для захисту від CSRF. З цієї причини спроба відкрити URL запиту в браузері призведе до помилки. В цілому використовувати GET не рекомендується.

Метод 3. Аналогічний попередньому методу, але використовуються тип запиту POST і для різноманітності нативний серіалайзер. Запит виглядає так:


```
{
  "id": 3
}
```


Відповідь відрізняється незручним форматом дати:


```
{
  "d": "{\"Report\":{\"ReportID\":3,\"Date\":\"\\/Date(1426356000000)\\/\",\"Info\":\"Method_3_POST\"}}"
}
```


Метод 4. Ускладнений варіант попереднього, передає складний аргумент var arg = {ReportID: 4, Token: "Токен методу 4." }; в серіалізовані вигляді:


```
{
  "json": "{\"ReportID\":4,\"Token\":\"Токен метода 4\"}"
}
```


Відповідь аналогічний.

Метод 5. Тепер опишемо роботу з датою при використанні нативного серіалайзера на прикладі методу, який приймає і повертає дату. У методі 1 веб сервіс повернув дату як «Date»: "/ Date (1426356000000) /". Число в дужках - це кількість мілісекунд, що пройшли з півночі 1 січня 1970 року UTC (UNIX epoch). При цьому згадаємо, що у типу Date є відповідний конструктор new Date (milliseconds), тобто досить виділити це число і використовувати в конструкторі дати:

var date = new Date (parseInt (data.d.replace ("/ Date (", "") .replace (") /", ""), 10));

При цьому сам веб-сервіс коректно розуміє алермальний формат даты в змінній:

```
{
  "dateTime": "2015-03-25T05:49:13.604Z"
}
```

Метод 6. Це нестандартний спосіб формування відповіді, і в деяких випадках він може бути єдиною можливістю отримати необхідний результат. Як можна бачити, код сам встановлює заголовки відповіді і на бінарному рівні визначає контент.

```
{
  "Report": {
    "Date": "2015-03-15T00:00:00",
    "Info": "Method_6_GET_NonStandard, Мой текст",
    "ReportID": 6
  }
}
```

Зверніть увагу, що тут немає кореневого вузла «d». Також можна використовувати GET. Більш того, таким способом можна повернути і щось відмінне від «application / json;».

maxJsonLength

Для повернення важких відповідей необхідно збільшити значення maxJsonLength в web.config:

```
<?xml version="1.0"?>
```

```
<configuration>

  <system.web>
    <compilation debug="true" targetFramework="4.0" />
  </system.web>

 <system.web.extensions>
  <scripting>
   <webServices>
    <jsonSerialization maxJsonLength="1073741824"/>
   </webServices>
  </scripting>
 </system.web.extensions>

</configuration>
```

Метадані запиту

Крім самих даних запиту, часто має сенс Залогуватися його метадані, такі як вихідний IP-адреса і запитаний URL. Це дозволяє визначати хто, коли і через яку мережу робив запити. Крім цих двох головних параметрів, ви можете зберігати і будь-які заголовки. І навіть спробувати отримати DNS-ім'я, хоча це не завжди можливо і вимагає час. Змініть код метод getReportInfo () на наступ-ний

```
private string getReportInfo() {
  var request = this.Context.Request;

  StringBuilder sb = new StringBuilder();
  sb.Append("IP = ").AppendLine(request.UserHostAddress);
  sb.Append("URL = ").AppendLine(request.Url.OriginalString);
```

```
        sb.Append("Header 'Connection' = ").AppendLine(request.Headers["Connecti
on"]);

        DateTime dnsDate = DateTime.Now;
        TimeSpan dnsSpan;

        try {
    //        throw new Exception("Закомментируй меня.");
            var entry = Dns.GetHostEntry(request.UserHostAddress);
            dnsSpan = DateTime.Now.Subtract(dnsDate);
            sb.Append("HostName = ").AppendLine(entry.HostName);

        } catch (Exception ex) {
            dnsSpan = DateTime.Now.Subtract(dnsDate);
            sb.AppendLine(ex.Message);
        }

        sb.Append("dnsSpan = ").AppendLine(dnsSpan.ToString());
        return sb.ToString();
    }
```

Доступ до файлів

Іноді є необхідність отримати доступ до файлів по шляху, який відносний цього положення веб-сервісу. Для прикладу змініть метод getReportInfo на наступний код:

```
    private string getReportInfo(int reportID) {
        string dirRoot = HttpContext.Current.Server.MapPath("~");

        StringBuilder sb = new StringBuilder();
```

```
        sb.AppendLine("dirRoot = " + dirRoot);

        string fileCars = HttpContext.Current.Server.MapPath("~/bin/MyFiles/Cars.txt");
        sb.AppendLine("fileCars = " + fileCars);

        try {
            sb.AppendLine("Line 1 = " + File.ReadAllLines(fileCars)[0]);
            File.AppendAllText(fileCars, Environment.NewLine + DateTime.Now + " ReportID = " + reportID);

        } catch (Exception ex) {
            sb.AppendLine(ex.Message);
        }

        string dirFiles = Path.Combine((new DirectoryInfo(dirRoot)).Parent.FullName, "FinReportWebService_Files");
        sb.AppendLine("dirFiles = " + dirFiles);

        try{
            Directory.CreateDirectory(dirFiles);
            string newFile = Path.Combine(dirFiles, Guid.NewGuid() + ".txt");
            File.WriteAllText(newFile, "ReportID = " + reportID);
            sb.AppendLine(newFile);

        } catch (Exception ex) {
            sb.AppendLine(ex.Message);
        }
```

```
    return sb.ToString();
}
```

У проекті створіть папку MyFiles і в ній текстовий файл Cars.txt з параметрами Build Action: None / Copy always і будь-якої першим рядком:

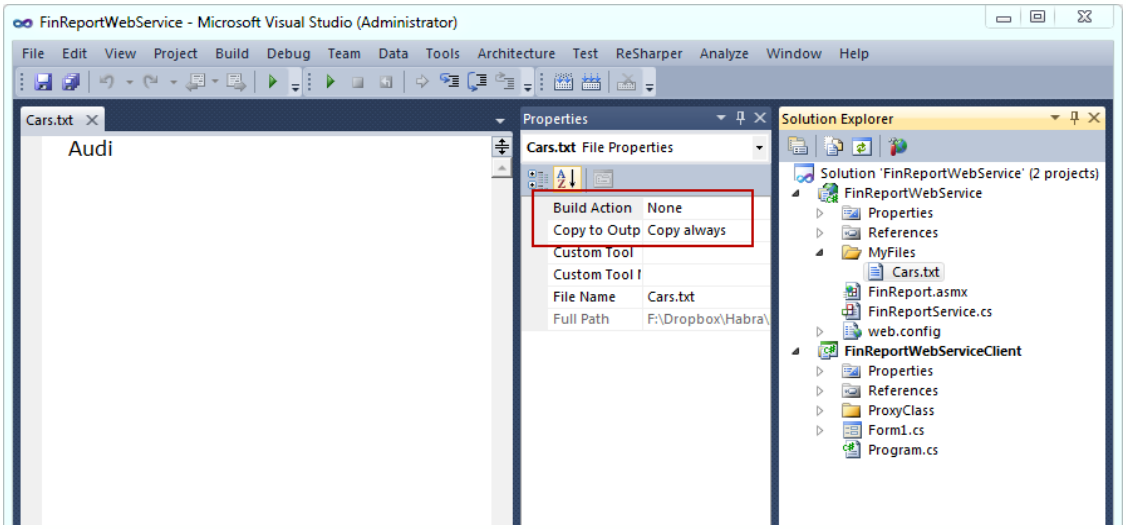

Рис. 2.6.

Таким чином при компіляції в папці bin буде автоматично створюватися папка MyFiles із зазначеним файлом, до якого ми будемо звертатися. Також цей код демонструє звернення до вищестоящої папці FinReportWebService_Files, яка створюється автоматично.

Отже, метод getReportInfo повертає текст, який містить наступну інформацію:

- Розташування самого веб-сервісу
- Шлях до вкладеного файлу Cars.txt
- Перший рядок цього файлу
- Текст можливої помилки його читання або модифікації
- Шлях до вищестоящої папки FinReportWebService_Files
- Шлях до успішно створеного в ній файлу
- Текст можливої помилки створення папки або файлу

Після публікації веб-сервісу в IIS, де його облікового запису дозволено тільки читати файли ми отримаємо наступний текст з помилками:

dirRoot = C:\CommonFolder\publish_FinReport

fileCars = C:\CommonFolder\publish_FinReport\bin\MyFiles\Cars.txt

Line 1 = Audi

Access to the path 'C:\CommonFolder\publish_FinReport\bin\MyFiles\Cars.txt' is denied.

dirFiles = C:\CommonFolder\FinReportWebService_Files

Access to the path 'C:\CommonFolder\FinReportWebService_Files' is denied.

Завдання на лабораторну роботу:

1. Створити програму клієнт для .asmx веб-сервісу та налаштувати їх взаємодію.

2. Протестувати веб-серверну програму, за допомогою відправки або отримання повідомлень з веб-сервісу.

Контрольні питання

1. Що такое клієнт-серверна архітектура?
2. За допомогою яких протоколів клієнт може спілкуватися з сервером?
3. Які є методи передачі повідомлень між клієнтом та сервером?
4. Які групи помилок існують в передачі даних між сервером та клієнтом? Який тип помилкок описує кожна група?
5. У яких форматах можуть бути представлені дані, якими обмінюється клієнт та сервер?

Лабораторна робота 3

Реалізація WCF-служби з використанням Visual Studio.

Мета роботи: ознайомлення з WCF-службами, реалізація WCF-служби з використанням Visual Studio.

Теоретичні положення

Windows Communication Foundation (WCF) – це платформа для створення сервіс-орієнтованих додатків. Використовуючи WCF, ви можете відправляти дані у вигляді асинхронних повідомлень від однієї кінцевої точки служби до іншої. Кінцева точка служби може бути частиною постійно доступної служби, розміщеної в IIS, або це може бути служба, розміщена в додатку. Кінцева точка може бути клієнтом служби, яка запитує дані з кінцевої точки служби. Повідомлення можуть бути простими, як один символ або слово, відправлене у форматі XML, або складними, як потік двійкових даних. Кілька прикладів сценаріїв включають в себе:

• Безпечний сервіс для обробки бізнес-транзакцій.

• Служба, яка надає поточні дані іншим користувачам, наприклад звіт про трафік або іншу службу моніторингу.

• Чат-сервіс, який дозволяє двом людям спілкуватися або обмінюватися даними в режимі реального часу.

• Додаток панелі моніторингу, який опитує одну або кілька служб для отримання даних і представляє їх в логічному поданні.

• Представлення робочого процесу, реалізованого за допомогою Windows Workflow Foundation в якості служби WCF.

• Додаток Silverlight для опитування служби на наявність останніх каналів даних.

Хоча створення таких додатків було можливо ще до появи WCF, WCF робить розробку кінцевих точок простіше, ніж будь-коли. Таким чином, WCF пропонує керований підхід до створення веб-служб і клієнтів веб-служб.

Особливості WCF

WCF включає в себе наступний набір функцій:

• Орієнтація на обслуговування

Одним із наслідків використання стандартів WS є те, що WCF дозволяє створювати сервіс-орієнтовані додатки. Сервіс-орієнтована архітектура (SOA) – це використання веб-сервісів для відправки та отримання даних. Послуги мають загальну перевагу в тому, що вони слабо пов'язані, а не жорстко закодовані від однієї програми до іншої. Слабо пов'язані відносини означають, що будь-який клієнт, створений на будь-якій платформі, може підключитися до будь-якої послуги, поки виконуються основні контракти.

• Сумісність

WCF впроваджує сучасні галузеві стандарти взаємодії веб-сервісів.

• Кілька шаблонів повідомлень

Обмін повідомленнями відбувається в одному з декількох патернів. Найбільш поширеним шаблоном є шаблон запиту/відповіді, коли одна кінцева точка запитує дані у другої кінцевої точки. Друга кінцева точка відповідає. Існують і інші шаблони, такі як одностороннє повідомлення, в якому одна кінцева точка відправляє повідомлення без очікування відповіді. Більш складний шаблон – це дуплексний шаблон обміну, в якому дві кінцеві точки встановлюють з'єднання і відправляють дані туди і назад, подібно програмі миттєвого обміну повідомленнями.

• Метадані служби

WCF підтримує публікацію метаданих служби з використанням форматів, зазначених у галузевих стандартах, таких як WSDL, XML Schema і WS-Policy. Ці метадані можна використовувати для автоматичного створення та налаштування клієнтів для доступу до служб WCF. Метадані можуть бути опубліковані через HTTP і HTTPS або за допомогою стандарту обміну метаданими веб-служби.

• Контракт даних

Оскільки WCF побудований з використанням Платформи. NET Framework, він також включає в себе зручні для коду методи надання контрактів, які ви хочете примусово виконати. Одним з універсальних типів контрактів є контракт на передачу даних. По суті, при кодуванні служби за допомогою Visual C# або Visual Basic найпростіше обробляти дані, створюючи класи, що представляють сутність даних з властивостями, що належать цій сутності. WCF включає в себе комплексну систему для роботи з даними таким простим способом. Після створення класів, що представляють дані, служба автоматично генерує метадані, що дозволяють клієнтам відповідати розробленим вами типам даних.

• Безпека

Повідомлення можуть бути зашифровані для захисту конфіденційності, і ви можете вимагати, щоб користувачі автентифікувалися, перш ніж їм буде дозволено отримувати повідомлення. Безпека може бути реалізована з використанням добре відомих стандартів, таких як SSL або WS-SecureConversation. Додаткові відомості див.у розділі Безпека.

• Кілька транспортів і кодувань

Повідомлення можуть бути відправлені по будь-якому з декількох вбудованих транспортних протоколів і кодувань. Найбільш поширеним протоколом і кодуванням є відправка текстових закодованих soap-повідомлень з використанням протоколу передачі гіпертексту (HTTP) для використання у всесвітній павутині. Крім того, WCF дозволяє відправляти повідомлення по протоколу TCP, іменованим каналам або MSMQ. Ці повідомлення можуть бути закодовані у вигляді тексту або з використанням оптимізованого двійкового формату. Двійкові дані можуть бути ефективно відправлені за допомогою стандарту MTOM. Якщо жоден з наданих транспортів або кодувань не відповідає вашим потребам, ви можете створити свій власний транспорт або кодування.

• Надійні і поставлені в чергу повідомлення

WCF підтримує надійний обмін повідомленнями з використанням надійних сеансів, реалізованих через WS-Reliable Messaging і використовують

MSMQ. Щоб отримати додаткові відомості про надійну підтримку обміну повідомленнями та чергами у WCF, див. розділ черги та надійні сеанси.

• «Міцні» повідомлення

«Міцне» повідомлення – це те, що ніколи не втрачається через порушення зв'язку. Повідомлення в довготривалому шаблоні повідомлень завжди зберігаються в базі даних. У разі збою база даних дозволяє відновити обмін повідомленнями після відновлення з'єднання. Ви також можете створити довговічне повідомлення за допомогою Windows Workflow Foundation (WF).

• Операція

WCF також підтримує транзакції з використанням однієї з трьох моделей транзакцій: WS-AtomicTransactions, API в системі.Простір імен транзакцій і координатор розподілених транзакцій Microsoft.

• Підтримка AJAX і REST

REST – це приклад розвивається технології Web 2.0. WCF можна налаштувати для обробки "простих" XML-даних, які не загорнуті в конверт soap. WCF також може бути розширений для підтримки певних форматів XML, таких як ATOM (популярний стандарт RSS), і навіть не xml-форматів, таких як JavaScript Object Notation (JSON).

• Розтяжність

Архітектура WCF має ряд точок розширення. Якщо потрібна додаткова можливість, існує ряд точок входу, які дозволяють налаштувати поведінку служби.

Інтеграція WCF з іншими технологіями Microsoft

WCF – це гнучка платформа. Через цю надзвичайну гнучкість WCF також використовується в кількох інших продуктах Microsoft.

Першою технологією для сполучення з WCF стала Windows Workflow Foundation (WF). Робочі процеси спрощують розробку додатків, інкапсулюючи кроки в робочий процес як «дії». У першій версії Windows Workflow Foundation розробник повинен був створити вузол для робочого процесу. Наступна версія

Windows Workflow Foundation була інтегрована з WCF. Це дозволило легко розмістити будь-який робочий процес у службі WCF. Це можна зробити, автоматично вибравши тип проекту WF/WCF у Visual Studio.

Microsoft BizTalk Server R2 також використовує WCF як комунікаційну технологію. BizTalk призначений для прийому і перетворення даних з одного стандартизованого формату в інший. Повідомлення повинні бути доставлені в його центральне вікно повідомлень, де повідомлення може бути перетворено або за допомогою суворого зіставлення, або за допомогою однієї з функцій BizTalk, таких як механізм робочого процесу. Тепер BizTalk може використовувати адаптер WCF Line of Business (LOB) для доставки повідомлень у вікно повідомлень.

Microsoft Silverlight – це платформа для створення сумісних, багатих вебдодатків, які дозволяють розробникам створювати мультимедійні веб-сайти (наприклад, потокове відео). Починаючи з версії 2, Silverlight включила WCF як комунікаційну технологію для підключення додатків Silverlight до кінцевих точок WCF.

Функції хостингу Windows Server AppFabric application server спеціально розроблені для розгортання та управління додатками, що використовують WCF для зв'язку. Функції хостингу включають в себе багатий інструментарій і параметри конфігурації, спеціально розроблені для додатків з підтримкою WCF.

Створення простої служби WCF в Windows Forms

Створення Служби

Запустіть Visual Studio.

На початковому екрані виберіть Створити проект.

Введіть бібліотеку служб WCF в поле пошуку на сторінці створення нового проекту . Виберіть шаблон C# або Visual Basic для бібліотеки служби WCF, а потім натисніть кнопку Далі.

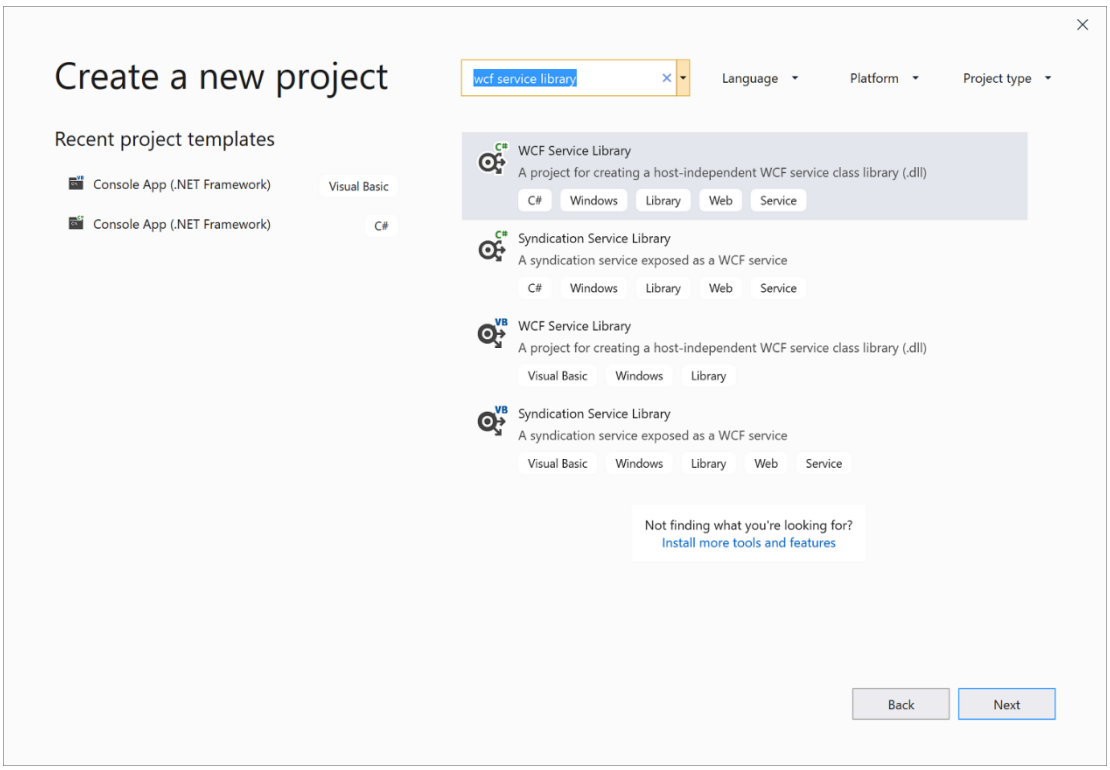

Рис. 3.1. Створення WCF додатку.

Якщо шаблони не відображаються, може знадобитися встановити компонент Windows Communication Foundation Visual Studio. Виберіть встановити додаткові інструменти та компоненти, щоб відкрити Visual Studio Installer. Перейдіть на вкладку окремі компоненти, прокрутіть вниз до пункту дії розробки, а потім виберіть Windows Communication Foundation. Натисніть кнопку Змінити.

На сторінці Налаштування нового проекту натисніть кнопку Створити.

Буде створена працююча служба, яку можна протестувати і використовувати. Наступні дві дії демонструють, як можна змінити метод за замовчуванням для використання іншого типу даних. У реальному додатку необхідно також додати до служби Її спеціальні функції.

У браузері рішень двічі клацніть IService1.vb або IService1.CS.

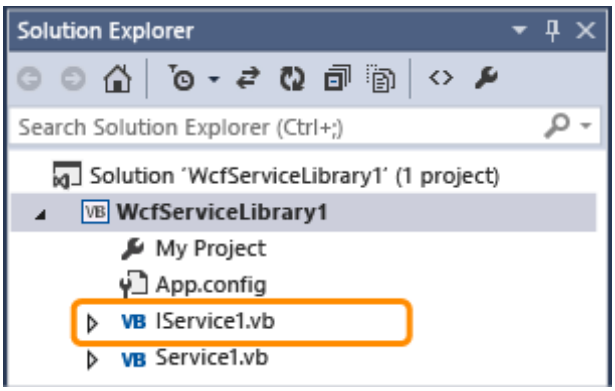

Рис. 3.2. IService1.vb

В рядку тип value параметра

```
C#

[OperationContract]
string GetData(int value);
```

Рис. 3.3. Присвоєння значень даним.

Змініть на string

```
C#

[OperationContract]
string GetData(string value);
```

Рис. 3.4. Присвоєння значення string даним.

У наведеному вище коді зверніть увагу на атрибути <OperationContract ()> або [OperationContract]. Ці атрибути є обов'язковими для будь-якого методу, наданого службою.

У браузері рішень двічі клацніть Service1.vb або Service1.CS.

Знайдіть наступний рядок:

```
C#

public string GetData(int value)
{
    return string.Format("You entered: {0}", value);
}
```

Рис. 3.5.

Змініть тип value параметра на string:

```C#
public string GetData(string value)
{
    return string.Format("You entered: {0}", value);
}
```

Рис. 3.6.

Тестування служби

Натисніть клавішу F5, щоб запустити службу. Відкриється форма тестового клієнта WCF, яка завантажує службу.

У формі тестовий клієнт WCF двічі клацніть метод GetData() в розділі IService1. Відкриється вкладка GetData .

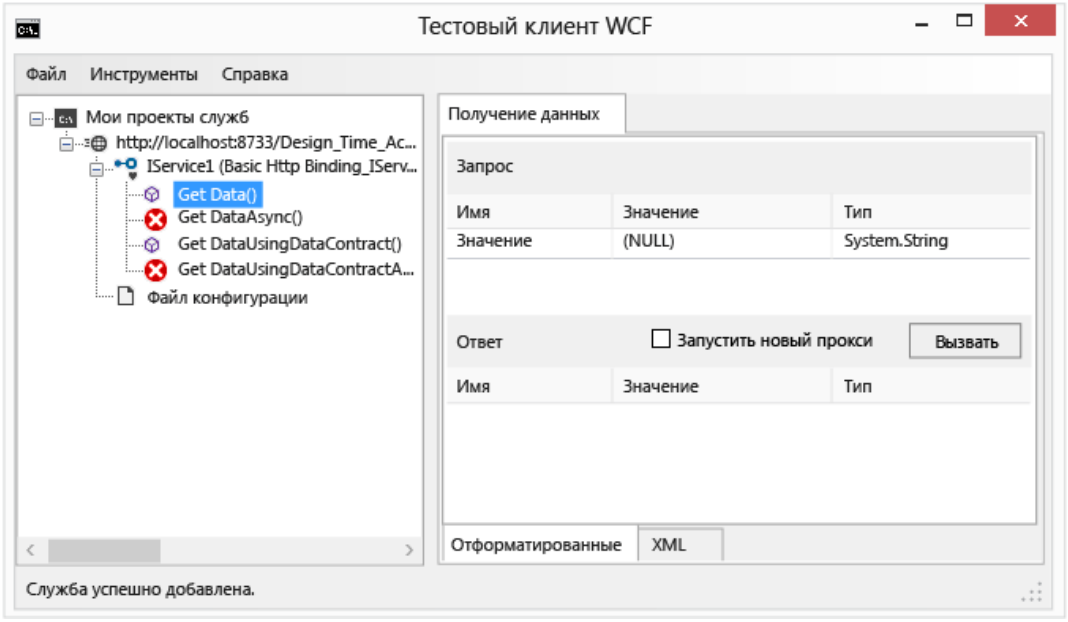

Рис. 3.7. Тестування служби.

У діалоговому вікні запит виберіть поле значення і введіть Hello.

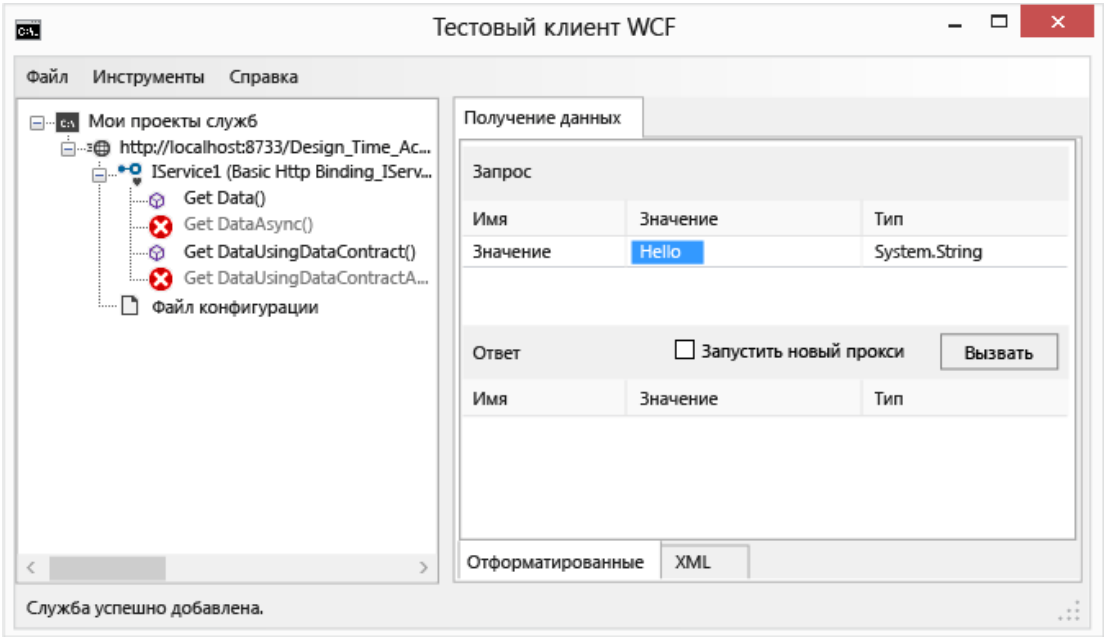

Рис. 3.8. Присвоєння значень полям.

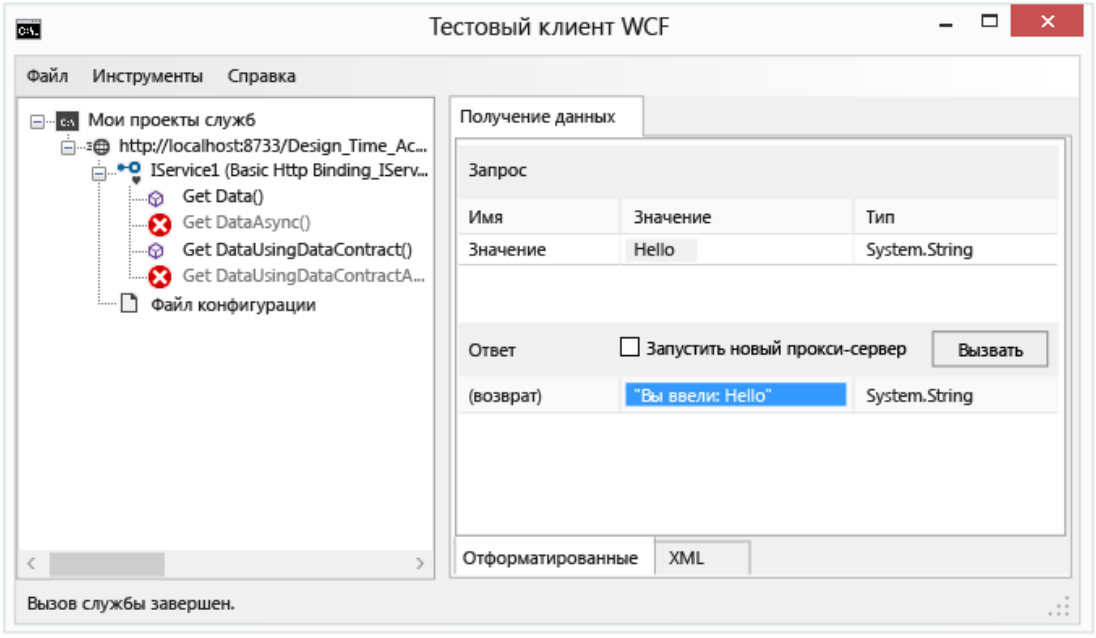

Рис. 3.9. Присвоєння значень полям.

Натисніть кнопку Викликати. Якщо з'явиться діалогове вікно "попередження системи безпеки", натисніть кнопку ОК. Результат відобразиться в полі відповідь.

Доступ до служби

Посилання на службу WCF

У меню файл виберіть команду Додати, а потім-новий проект.

У діалоговому вікні новий проект розгорніть вузол Visual Basic або Visual C#, виберіть пункт Windows, а потім виберіть Windows Forms додаток.

Натисніть кнопку ОК, щоб відкрити проект.

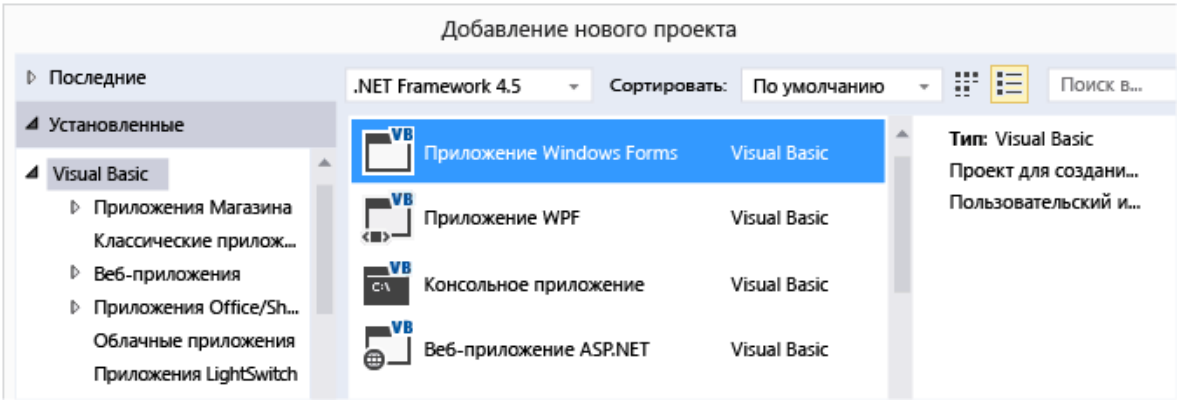

Рис. 3.10. Відкриття проекту WCF.

У меню Файл клацніть вихід, щоб закрити тестову форму.

Клацніть правою кнопкою миші WindowsApplication1 і виберіть команду Додати посилання на службу. Відкриється діалогове вікно Додати посилання на службу.

У діалоговому вікні Додавання посилання на службу виберіть знайти.

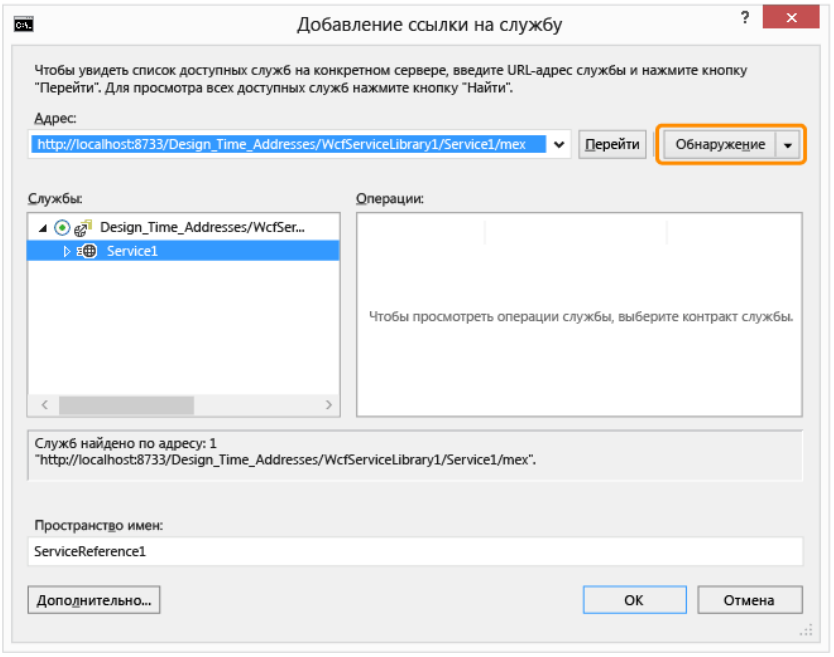

Рис. 3.11. Додавання посилання на службу.

Service1 відображається в області Служби .

Натисніть кнопку ОК, щоб додати посилання на службу.

Створення клієнтської програми

У браузері рішень двічі клацніть Form1.vb або Form1.cs, щоб відкрити конструктор Windows Forms, якщо він ще не відкритий.

З панелі Елементів перетягніть на форму елемент керування TextBox, елемент керування та елемент керування Button.

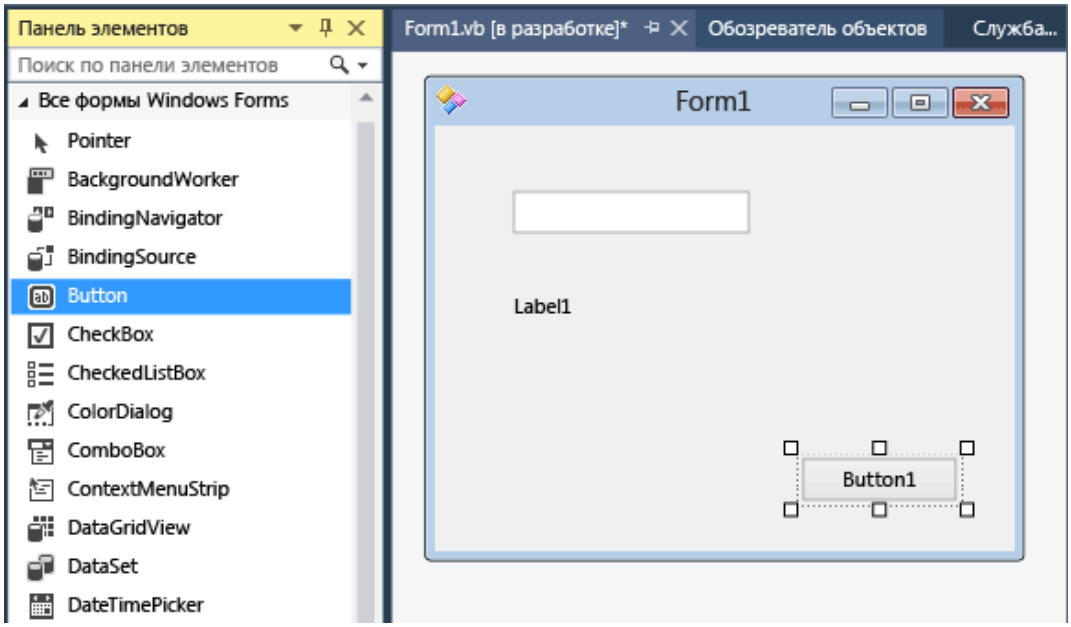

Рис. 3.12. Створення елементів у веб-формі.

Двічі клацніть Button і додайте наступний код до обробника подій Click:

```
private void button1_Click(System.Object sender, System.EventArgs e)
{
    ServiceReference1.Service1Client client = new
        ServiceReference1.Service1Client();
    string returnString;

    returnString = client.GetData(textBox1.Text);
    label1.Text = returnString;
}
```

Рис. 3.13. Лістинг коду обробника подій Click.

У браузері рішень клацніть правою кнопкою миші WindowsApplication1 і виберіть команду призначити запускається проектом.

Натисніть клавішу F5, щоб запустити проект. Введіть будь-який текст і натисніть кнопку. На цій мітці відображається введений текст:

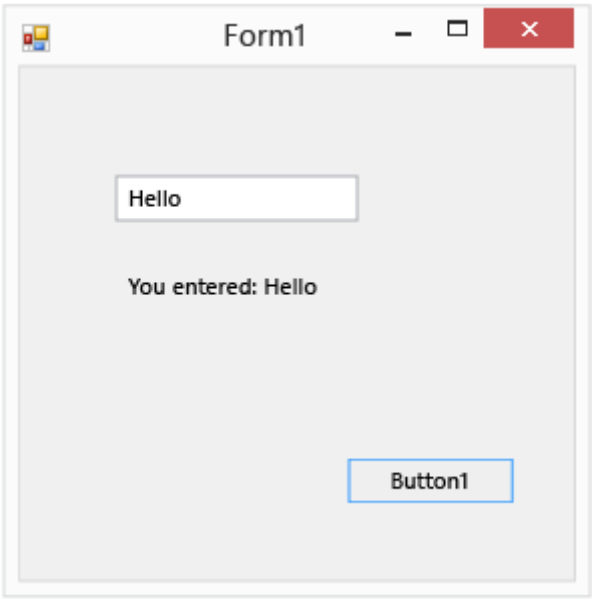

Рис. 3.14. Запуск проекту.

Завдання до лабораторної роботи

1. Ознайомитися з короткими теоретичними відомостями.

2. Розробити логіку роботи серверної частини додатка і продемонструвати успішну обробку клієнтських запитів.

3. До існуючого проекта підключити службу WCF в Windows Forms.

Контрольні питання

1. Що таке WCF-форми?

2. Назвіть основні компоненти WCF-форм?

3. Які дії можна виконувати над WCF-формами?

Лабораторна робота 4

Реалізація WCF-служби як REST сервісу за рахунок використання стандартних прив'язок. Розробка веб- і мобільного клієнтів сервісу.

Мета роботи: реалізувати WCF-службу як REST сервіс за рахунок використання стандартних прив'язок. Розробити веб- і мобільний клієнт сервісу.

Теоретичні положення

При виклику rest-служби зі звичайної (на основі SOAP) служби WCF контекст операції в методі служби (з відомостями про вхідний запит) перевизначає контекст, який повинен використовуватися вихідним запитом. В результаті запити HTTP GET перетворюються на запити HTTP POST. Щоб змусити службу WCF використовувати правильний контекст для виклику rest-служби, створіть новий об'єкт OperationContextScope і викличте rest-службу з області контексту операції. У цьому розділі описано створення простого прикладу, що ілюструє даний спосіб.

Визначте простий контракт служби для REST-служби.

```C#
[ServiceContract]
public interface IRestInterface
{
    [OperationContract, WebGet]
    int Add(int x, int y);

    [OperationContract, WebGet]
    string Echo(string input);
}
```

Рис. 4.1. Контракт служби для REST-служби.

Реалізуйте контракт rest-служби.

```C#
public class RestService : IRestInterface
{
    public int Add(int x, int y)
    {
        return x + y;
    }

    public string Echo(string input)
    {
        return input;
    }
}
```

Рис. 4.2. Реалізація контракту rest-служби.

Визначте контракт служби WCF, яка буде використовуватися для виклику rest-служби.

```C#
[ServiceContract]
public interface INormalInterface
{
    [OperationContract]
    int CallAdd(int x, int y);

    [OperationContract]
    string CallEcho(string input);
}
```

Рис. 4.3. Визначення контракту служби WCF.

Реалізуйте контракт служби WCF.

```
public class NormalService : INormalInterface
{
    static MyRestClient client = new MyRestClient(RestServiceBaseAddress);
    public int CallAdd(int x, int y)
    {
        return client.Add(x, y);
    }

    public string CallEcho(string input)
    {
        return client.Echo(input);
    }
}
```

Рис. 4.4. Реалізація контракту служби WCF.

Створіть клієнтський проксі-клас для REST-служби

Використання ClientBase<TChannel> для реалізації проксі клієнта. Для кожного викликається методу створюється і використовується для виклику операції Нова OperationContextScope.

```
public class MyRestClient : ClientBase<IRestInterface>, IRestInterface
{
    public MyRestClient(string address)
        : base(new WebHttpBinding(), new EndpointAddress(address))
    {
        this.Endpoint.Behaviors.Add(new WebHttpBehavior());
    }

    public int Add(int x, int y)
    {
        using (new OperationContextScope(this.InnerChannel))
        {
            return base.Channel.Add(x, y);
        }
    }

    public string Echo(string input)
    {
        using (new OperationContextScope(this.InnerChannel))
        {
            return base.Channel.Echo(input);
        }
    }
}
```

Рис. 4.5. Створення клієнтського проксі-класу для REST-служби.

Розмістіть обидві служби в консольному додатку, додавши необхідні кінцеві точки і поведінку. Потім викличте звичайну службу WCF.

```
public static void Main()
{
    ServiceHost restHost = new ServiceHost(typeof(RestService), new Uri(RestServiceBaseAddress));
    restHost.AddServiceEndpoint(typeof(IRestInterface), new WebHttpBinding(), "").Behaviors.Add(new WebHttpBehavior());
    restHost.Open();

    ServiceHost normalHost = new ServiceHost(typeof(NormalService), new Uri(NormalServiceBaseAddress));
    normalHost.AddServiceEndpoint(typeof(INormalInterface), new BasicHttpBinding(), "");
    normalHost.Open();

    Console.WriteLine("Hosts opened");

    ChannelFactory<INormalInterface> factory = new ChannelFactory<INormalInterface>(new BasicHttpBinding(), new EndpointAddress(NormalServiceBaseAddress));
    INormalInterface proxy = factory.CreateChannel();

    Console.WriteLine(proxy.CallAdd(123, 456));
    Console.WriteLine(proxy.CallEcho("Hello world"));
}
```

Рис. 4.6.

Завдання на лабораторну роботу

1. Ознайомитись з короткими теоретичними відомостями.

2. Розробити логіку роботи серверної частини мабільного додатка і продемонструвати успішну обробку клієнтських запитів.

3. Продемонструвати виконану роботу.

Контрольні питання

1. Що таке REST-служби?

2. Опишіть логіку роботи REST-служби?

3. Як підключити WCF до REST-служби?

Лабораторна робота 5

Розробка веб-служби Web API.

Мета роботи: ознайомлення з веб-API за допомогою ASP.NET Core.

Теоретичні положення

Для створення API-інтерфейсу знадобиться наступне:

GET /api/TodoItems - отримання всіх елементів задач.

GET /api/TodoItems/{id} - отримання об'єкта за ідентифікатором.

POST /api/TodoItems - додавання нового елемента.

PUT /api/TodoItems/{id} - оновлення існуючого елемента.

DELETE /api/TodoItems/{id} - видалення елемента.

Структура додатку виглядає наступним чином:

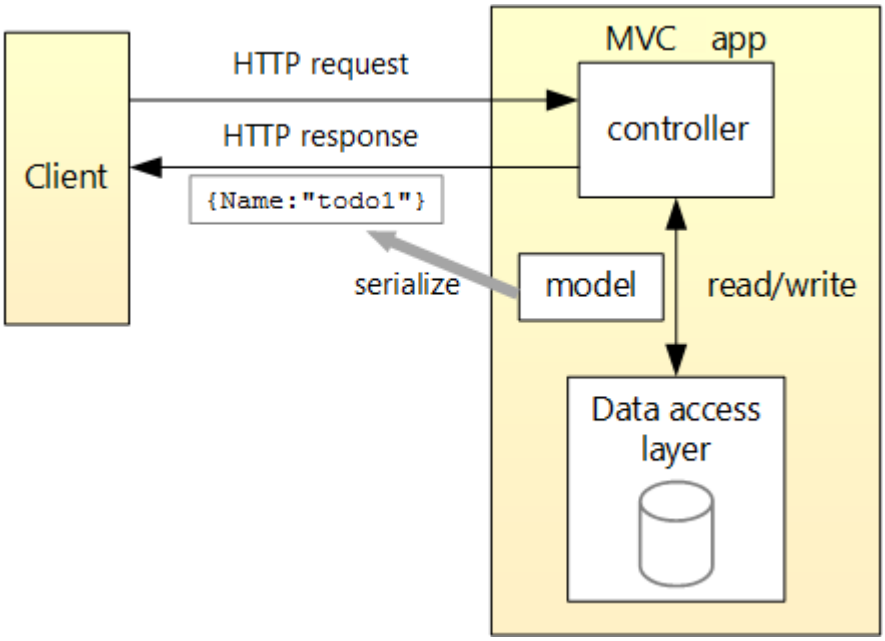

Рис. 5.1 Структура додатку.

Створення веб- проекту

1. У меню Файл виберіть пункт Створити> Проект.
2. Виберіть шаблон Веб-додаток ASP.NET Core і натисніть Далі.
3. Назвіть проект TodoApi і натисніть Створити.

4. У діалоговому вікні Створення веб-додатки ASP.NET Core переконайтеся в тому, що обрані платформи .NET Core і ASP.NET Core 3.1. Виберіть шаблон API і натисніть кнопку Створити.

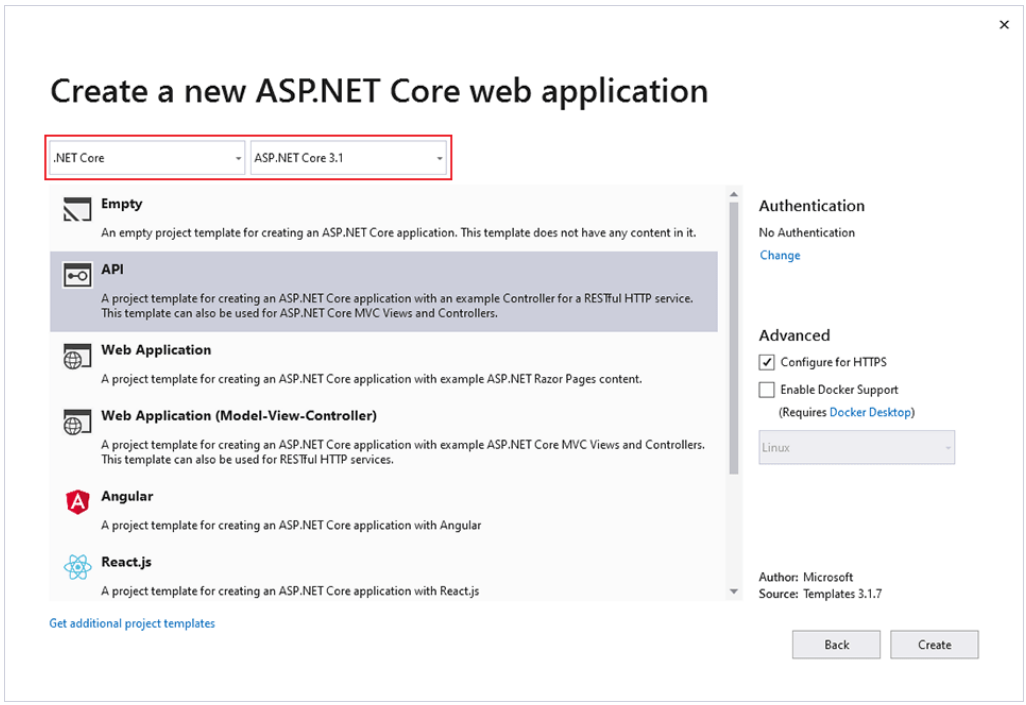

Рис. 5.2. Створення веб-пректу.

Додавання класу моделі

Модель - це набір класів, що представляють дані, якими управляє додаток. Модель цього додатка містить єдиний клас TodoItem.

1. У браузері рішень клацніть проект правою кнопкою миші. Виберіть **Додати**> **Нова папка**. Назвіть папку *Models*.

2. Клацніть папку *Models* правою кнопкою миші і виберіть пункти **Додати**> **Клас**. Дайте класу ім'я *TodoItem* і виберіть **Додати**.

3. Замініть код шаблону наступним кодом:

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Властивість Id виступає в якості унікального ключа реляційної бази даних.

Класи моделей можна розміщувати в будь-якому місці проекту, але зазвичай для цього використовується папка Models.

Додавання контексту бази даних

Контекст бази даних -це основний клас, який координує функціональні можливості Entity Framework для моделі даних. Цей клас є похідним від класу Microsoft.EntityFrameworkCore.DbContext.

Додавання пакетів NuGet

1. У меню **Сервіс** виберіть **Диспетчер пакетів NuGet> Управління пакетами NuGet** для вирішення.

2. Перейдіть на вкладку **Огляд** і введіть **Microsoft.EntityFramework Core.SqlServer** в поле пошуку.

3. На панелі зліва виберіть **Microsoft.EntityFramework Core.SqlServer**.

4. Встановіть прапорець **Проект** на правій панелі і виберіть **Встановити**.

5. Додайте пакет NuGet **пакет NuGet Microsoft.EntityFramework Core.InMemory**, виконавши наведені вище інструкції.

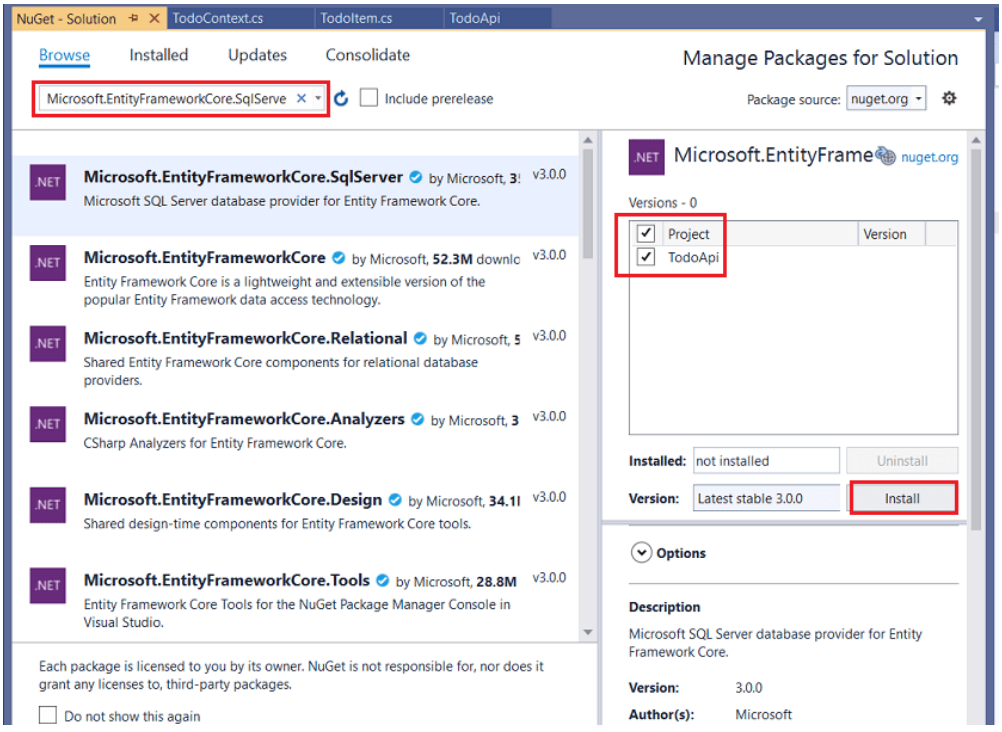

Рис. 5.3. Додавання контексту бази даних.

Додавання контексту бази даних TodoContext

Клацніть папку *Models* правою кнопкою миші і виберіть пункти **Додати**> **Клас**. Назвіть клас *TodoContext* і натисніть **Додати**.

Введіть наступний код:

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Реєстрація контексту бази даних

У ASP.NET Core служби (такі як контекст бази даних) повинні бути зареєстровані за допомогою контейнера впровадження залежностей. Контейнер надає службу контролерам.

Формування шаблонів контролера

1. Клацніть папку Controllers правою кнопкою миші.
2. Клацніть **Додати**> **Створити шаблонний елемент**.
3. Виберіть **Контролер API з діями, що використовує Entity Framework**, а потім виберіть **Додати**.
4. У діалоговому вікні **Контролер API з діями, що використовує Entity Framework** зробіть наступне:
5. Виберіть **TodoItem (TodoApi.Models)** в поле **Клас моделі**.
6. Виберіть **TodoContext (TodoApi.Models)** в поле **Клас контексту даних**.
7. Натисніть **Додати**.

Знайомство з методом створення PostTodoItem

Змініть інструкцію повернення в PostTodoItem і використовуйте оператор nameof:

```
// POST: api/TodoItems
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoI-
tem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id },
todoItem);
}
```

Попередній код є методом HTTP POST, позначених атрибутом [HttpPost]. Цей метод отримує значення елемента списку справ з тексту HTTP-запиту.

Метод CreatedAtAction:

У разі успіху повертає код стану HTTP 201. HTTP 201 являє собою стандартну відповідь для методу HTTP POST, що створює ресурс на сервері.

Додає у відповідь заголовок Location. Заголовок Location вказує URI нової створеної завдання. Додаткові відомості див. У статті 10.2.2 201 "Створено ресурс".

Вказує дію GetTodoItem для створення URI заголовка Location. Ключове слово nameof C # використовується для запобігання жорсткого програмування імені дії у виклику CreatedAtAction.

Тестування PostTodoItem з використанням Postman

1. Створіть новий запит.
2. Встановіть HTTP-метод POST.
3. Задайте для URI значення https: // localhost: <port> / api / TodoItems. Наприклад, https: // localhost: 5001 / api / TodoItems.
4. Відкрийте вкладку **Тіло**.
5. Встановіть перемикач **без обробки**.
6. Задайте тип JSON (додаток / json).

7. У тілі запиту введіть код JSON для елемента списку справ:

```
{
  "name":"walk dog",
      "isComplete":true
}
```

8. Натисніть кнопку Надіслати.

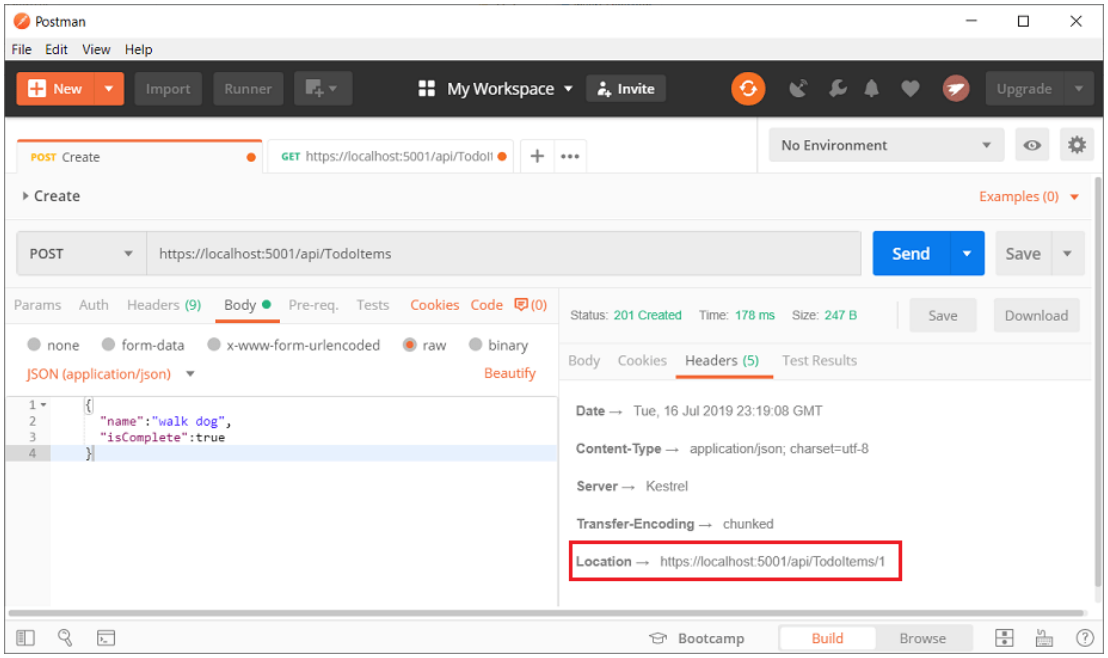

Рис. 5.4. Тестування PostTodoItem з використанням Postman.

Тестування URI заголовка розташування за допомогою Postman

1. Перейдіть на вкладку **Заголовки** в області **Відповідь**.
2. Скопіюйте значення заголовка **Розташування**:
3. Встановіть HTTP-метод GET.
4. Задайте для URI значення https: // localhost: <port> / api / TodoItems / 1. Наприклад, https: // localhost: 5001 / api / TodoItems / 1.
5. Натисніть кнопку **Надіслати**.

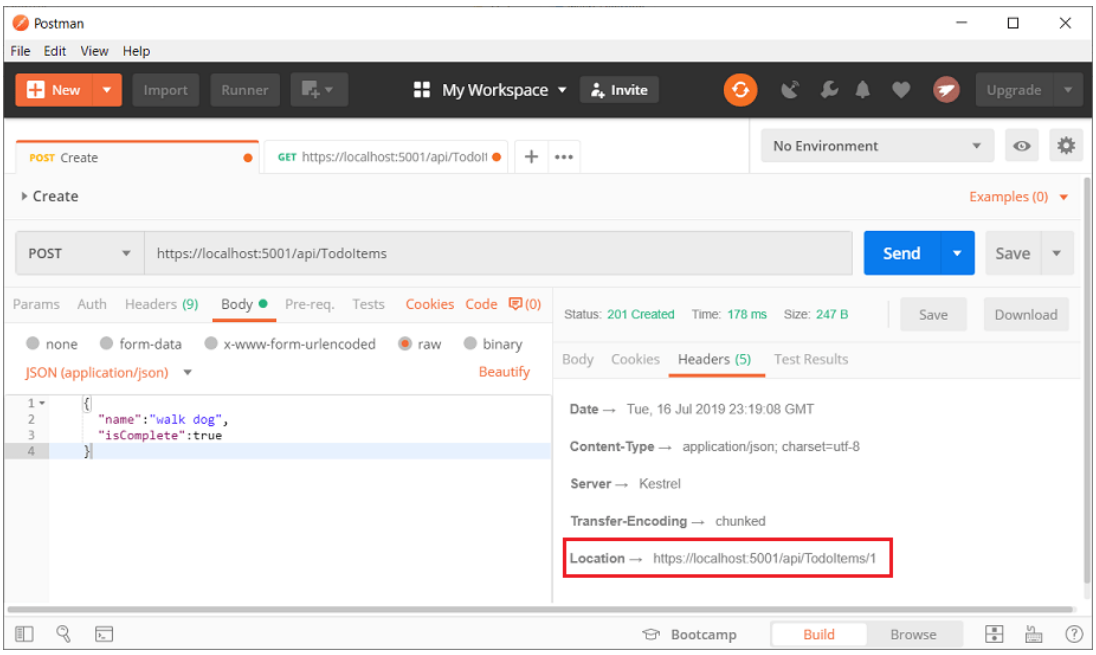

Рис. 5.5 Тестування URI заголовка розташування за допомогою Postman.

Знайомство з методами GET

Ці методи реалізують дві кінцеві точки GET:

GET / api / TodoItems

GET / api / TodoItems / {id}

Протестуйте додаток, викликавши ці дві кінцеві точки в браузері або в Postman. приклад:

https: // localhost: 5001 / api / TodoItems

https: // localhost: 5001 / api / TodoItems / 1

Тестування Get з використанням Postman

1. Створіть новий запит.
2. Вкажіть метод HTTP **GET**.
3. Задайте для URI запиту значення https: // localhost: <port> / api / TodoItems. Наприклад, https: // localhost: 5001 / api / TodoItems.
4. Виберіть режим **Подання з двома областями** в Postman.
5. Натисніть кнопку **Надіслати**.

Ця програма використовує виконувати в пам'яті базу даних. Якщо зупинити і знову запустити його, що передує запит GET не повернеться ніяких даних. Якщо дані не повертаються, дані для програми виходять методом POST.

Виклик веб-API за допомогою JavaScript

Додавання підтримки аутентифікації в веб-API

Посвідчення ASP.NET Core дозволяє використовувати функцію входу в інтерфейсі для веб-додатків ASP.NET Core. Щоб захистити веб-API і односторінкові додатки, використовуйте один з наступних способів:

Azure Active Directory

Azure Active Directory B2C (Azure AD B2C)

IdentityServer4

Завдання на лабораторну роботу

1. Створити веб-API за допомогою ASP.NET Core.
2. Додати клас моделі і контекст бази даних у веб-проект.
3. Провести тестування з використанням Postman.

Контрольні питання

1. Що таке модель?
2. Для чого потрібно створювати контекст бази даних?
3. Що реалізують методи Get?

Лабораторна робота 6

Розробка і використання власного сервісу для впровадження залежностей в ASP.NETCore додатку.

Мета роботи: розробка сервісу для впровадження залежностей з використанням можливостей ASP.NETCore.

Теоретичні положення

ASP.NET Core підтримує проектування програмного забезпечення з можливістю впровадження залежностей. При такому підході досягається інверсія управління між класами і їх залежностями.

Залежність - це будь-який об'єкт, від якого залежить інший об'єкт. Розглянемо наступний клас MyDependency з методом WriteMessage, від якого залежать інші класи:

```
public class MyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage called. Message: {message}");
    }
}
```

Клас може створити екземпляр класу MyDependency, щоб використовувати його метод WriteMessage. У наступному прикладі клас MyDependency виступає залежністю класу IndexModel:

```
public class IndexModel : PageModel
{
    private readonly MyDependency _dependency = new MyDependency();
```

```
    public void OnGet()
    {
        _dependency.WriteMessage("IndexModel.OnGet created this message.");
    }
}
```

Цей клас створює MyDependency і безпосередньо залежить від цього класу. Залежності в коді, як в попередньому прикладі, являють собою певну проблему. Їх слід уникати з наступних причин.

Щоб замінити MyDependency інший реалізацією, клас IndexModel необхідно змінити.

Якщо у MyDependency є залежності, їх конфігурацію повинен виконувати клас IndexModel. У великих проектах, коли від MyDependency залежать багато класів, код конфігурації розтягується по всьому додатком.

Така реалізація погано підходить для модульних тестів. У додатку потрібно використовувати імітацію або заглушку у вигляді класу MyDependency, що при такому підході неможливо.

Впровадження залежностей усуває ці проблеми наступним чином:

Використовується інтерфейс або базовий клас для абстрагування реалізації залежностей.

Залежність реєструється в контейнері служб. ASP.NET Core надає вбудований контейнер служб, IServiceProvider. Як правило, служби реєструються в додатку в методі Startup.ConfigureServices.

Служба впроваджується в конструктор класу там, де він використовується. Платформа бере на себе створення екземпляра залежності і його видалення, коли він більше не потрібен.

У прикладі додатка інтерфейс IMyDependency визначає метод WriteMessage:

```
public interface IMyDependency
{
   void WriteMessage(string message);
   }
```

Цей інтерфейс реалізується конкретним типом, MyDependency.

```
public class MyDependency : IMyDependency
{
   public void WriteMessage(string message)
   {
      Console.WriteLine($"MyDependency.WriteMessage Message: {message}");
   }
}
```

Приклад програми реєструє службу IMyDependency з конкретним типом MyDependency.

Використовуючи шаблон впровадження залежностей, контролер:

• не використовує конкретний тип MyDependency, тільки інтерфейс IMyDependency, який він реалізує. Це спрощує зміна реалізації, використовуваної контролером, без зміни контролера.

• не створює екземпляр MyDependency, він створюється контейнером впровадження залежностей.

Використання ланцюжка впроваджень залежностей не є чимось незвичайним. Кожна запитувана залежність запитує власні залежності. Контейнер дозволяє залежно в графі і повертає повністю дозволену службу. Весь набір залежностей, які потрібно вирішити, зазвичай називають деревом залежностей, графом залежностей або графом об'єктів.

У термінології впровадження залежностей - служба:

• Зазвичай є об'єктом, що надає службу для інших об'єктів, наприклад службу IMyDependency.

• Не відноситься до веб-службі, хоча служба може використовувати веб-службу.

Служби, впроваджені в конструктор Startup

Служби можна впровадити в конструктор Startup і метод Startup.Configure.

При використанні універсального вузла (IHostBuilder) в конструктор Startup можуть впроваджуватися тільки такі служби:

• IWebHostEnvironment

• IHostEnvironment

• IConfiguration

Будь-яка служба, зареєстрована в контейнері впровадження залежностей, може бути впроваджена в метод Startup.Configure:

```
public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
{
  ...
}
```

Реєстрація груп служб за допомогою методів розширення

Для реєстрації групи пов'язаних служб на платформі ASP.NET Core використовується угоду. Угода укладається у використанні одного методу розширення Add {GROUP_NAME} для реєстрації всіх служб, необхідних компоненту платформи. Наприклад, метод розширення <Microsoft.Extensions.Dependency Injection.MvcServiceCollectionExtensions.AddControllers> реєструє служби, необхідні для контролерів MVC.

Наступний код створюється шаблоном Razor Pages з використанням окремих облікових записів користувачів. Він демонструє, як додати додаткові служби в контейнер за допомогою методів розширення AddDbContext і AddDefaultIdentity:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<ApplicationDbContext>(options =>
      options.UseSqlServer(
          Configuration.GetConnectionString("DefaultConnection")));
  services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
      .AddEntityFrameworkStores<ApplicationDbContext>();
  services.AddRazorPages();
}
```

Кожен метод розширення services.Add {GROUP_NAME} додає і, можливо, налаштовує служби. Наприклад, AddControllersWithViews додає контролери MVC служб з необхідними уявленнями.

Час існування служб

Служби можна зареєструвати з одним з наступних варіантів часу існування:

• тимчасовий

• область дії

• одноелементний

Для кожної зареєстрованої служби вибирайте відповідний час існування.

Тимчасовий

Тимчасові служби часу існування створюються при кожному їх запиті з контейнера служб. Це час існування найкраще підходить для простих служб без відстеження стану. Реєструйте тимчасові служби за допомогою AddTransient.

У додатках, що обробляють запити, тимчасові служби видаляються в кінці запиту.

Область дії

Служби часу існування із заданою областю створюються один раз для кожного клієнтського запиту (підключення). Реєструйте служби із заданою областю за допомогою AddScoped.

У додатках, що обробляють запити, служби із заданою областю видаляються в кінці запиту.

При використанні Entity Framework Core метод розширення AddDbContext за замовчуванням реєструє типи DbContext із заданою областю часу існування.

Вирішувати службу із заданою областю з одноелементна об'єкта ***заборонено***. При обробці наступних запитів це може викликати неправильне стан служби. Допускається наступне:

• Дозвіл одноелементної служби зі служби із заданою областю або тимчасової служби.

• Дозвіл служби із заданою областю з іншої служби із заданою областю або тимчасової служби.

За замовчуванням в середовищі розробки дозвіл служби з іншої служби з більш довгим часом існування викликає виняток.

Використовуйте служби із заданою областю в ПО проміжного шару, застосовуючи один з наступних підходів:

• Впровадьте службу в метод Invoke або InvokeAsync ПЗ проміжного шару. За допомогою упровадите конструктор створюється виняток часу виконання, оскільки воно змушує службу із заданою областю вести себе як одноелементні об'єкт.

• Використовуйте фабричне ПЗ проміжного шару. ПЗ проміжного шару, зареєстроване з використанням цього підходу, активується при кожному клієнтському запиті (підключенні), що дозволяє впроваджувати служби із заданою областю в метод InvokeAsync ПО проміжного шару.

Одноелементні

Одноелементні служби часу існування створюються в наступних випадках.

• При першому запиті.

• Розробником при наданні примірника реалізації безпосередньо в контейнер. Цей підхід потрібно досить рідко.

Створений екземпляр використовують всі наступні запити. Якщо в додатку потрібно використовувати одноелементні служби, дозвольте контейнеру служб керувати часом їх існування. Чи не реалізуйте одноелементний підхід і надайте код для видалення одноелементні об'єктів. Служби ніколи не повинні віддалятися кодом, який дозволив службу з контейнера. Якщо тип або фабрика зареєстровані як одноелементні об'єкт, контейнер автоматично видалить одноелементні об'єкти.

Зареєструйте одноелементні служби за допомогою AddSingleton. Одноелементні служби повинні бути потокобезпечна і часто використовуватися в службах без відстеження стану.

У додатках, що обробляють запити, окремі служби видаляються, коли ServiceProvider видаляється після закінчення роботи програми. Оскільки пам'ять не звільняється до завершення роботи програми, рекомендується враховувати використання пам'яті одноелементна об'єктом.

Дозволяти службу із заданою областю з одноелементного об'єкта ***заборонено***. При обробці наступних запитів це може викликати неправильне стан служби. Допускається вирішувати одноелементну службу зі служби із заданою областю або тимчасової служби.

Поведінка впровадження через конструктор

Служби можна вирішувати за допомогою:

• IServiceProvider

• ActivatorUtilities:

1) Створює об'єкти, які не зареєстровані в контейнері.

2) Використовується з функціями платформи, такими як допоміжні додатки для тегів, контролери MVC і засоби прив'язки моделей.

Конструктори можуть приймати аргументи, які не надаються впровадженням залежностей, але ці аргументи повинні призначати значення за замовчуванням.

Коли дозвіл служб виконується через IServiceProvider або ActivatorUtilities, для впровадження через конструктор потрібно відкритий конструктор.

Коли дозвіл служб виконується через ActivatorUtilities, для впровадження через конструктор потрібна наявність лише одного відповідного конструктора. Перевантаження конструктора підтримуються, але може існувати лише одна перевантаження, всі аргументи якої можуть бути оброблені за допомогою впровадження залежностей.

Контексти Entity Framework

За замовчуванням контексти Entity Framework додаються в контейнер служби за допомогою часу існування із заданою областю, оскільки операції бази даних в веб-додатку зазвичай ставляться до області клієнтського запиту. Щоб використовувати інший час існування, вкажіть його за допомогою перевантаження AddDbContext. Служби даного часу існування не повинні використовувати контекст бази даних з часом існування коротше, ніж у служби.

Виклик служб з функції main

Створіть IServiceScope з IServiceScopeFactory.CreateScope для дозволу служби із заданою областю в галузі застосування. Цей спосіб дозволить отримати доступ до служби із заданою областю при запуску для виконання завдань по ініціалізації.

Служби запитів

Використовуючи доступні в межах запиту ASP.NET Core, надаються через колекцію HttpContext.RequestServices. При запиті в рамках запиту служби та їх залежності вирішуються з колекції RequestServices.

Платформа створює область для кожного запиту, а RequestServices постачальник послуг може надати із заданою областю. Всі служби із заданою областю дійсні до тих пір, поки запит активний.

Проектування служб для впровадження залежностей

При розробці служб для впровадження залежностей дотримуйтеся наступних рекомендацій:

Уникайте статичних класів і членів з відстеженням стану. Уникайте створення глобального стану. Для цього проектують додатки для використання окремих служб.

Уникайте прямого створення екземплярів залежних класів всередині служб. Пряме створення екземплярів зобов'язує використовувати в коді певну реалізацію.

Зробіть службою програми невеликими, добре організованими і зручними в тестуванні.

Якщо клас має занадто багато впроваджених залежностей, це може вказувати на те, що у класу занадто багато завдань і він порушує принцип єдиної обов'язки. Спробуйте виконати рефакторинг класу і перенести частину його обов'язків у нові класи. Пам'ятайте, що в класах моделі сторінок Razor Pages і класах контролера MVC повинні переважно виконуватися завдання, пов'язані з призначеним для користувача інтерфейсом.

Видалення служб

Контейнер викликає Dispose для створюваних ним типів IDisposable. Служби, дозволені з контейнера, ніколи не повинні віддалятися розробником. Якщо тип або фабрика зареєстровані як одноелементні об'єкт, контейнер автоматично видалить одноелементні об'єкти.

Загальні рекомендації по IDisposable

Не зазначати екземпляри IDisposable з тимчасовим часом існування. Замість цього використовуйте шаблон фабрики.

Не дозволяйте екземпляри IDisposable з тимчасовим часом існування або часоіснування із заданою областю в кореневу область. Єдиний виняток - це коли додаток створює або повторно створює і видаляє IServiceProvider, але це не є ідеальним варіантом.

Для отримання залежності IDisposable через DI не потрібно, щоб одержувач реалізовував сам інтерфейс IDisposable. Одержувач залежності IDisposable не повинен викликати Dispose для цієї залежності.

Області повинні використовуватися для управління часом існування служб. Області не є ієрархічними, і між ними немає спеціального зв'язку.

Заміна стандартного контейнера служб

Вбудований контейнер служб призначений для задоволення потреб платформи і більшості клієнтських додатків. Ми рекомендуємо використовувати вбудований контейнер, якщо тільки не потрібна конкретна функціональна можливість, яку він не підтримує, наприклад:

- впровадження властивостей
- Впровадження на ім'я
- дочірні контейнери
- Настроюється управління часом існування
- Func <T> підтримує відкладену ініціалізацію
- Реєстрація на основі угоди

Завдання на лабораторну роботу

1. Впровадити залежність в проект.
2. Додати в проект службу.
3. Застосувати один з методів реєстрації служб.

Контрольні питання

1. Що таке залежність і служба?
2. В чому полягає різниця між різними видами часами існування служб?
3. Для чого потрібен IDisposable?

СПИСОК ЛІТЕРАТУРИ

1. Pressman, Roger. Software Engineering: A Practitioner's Approach, McGraw Hill, New York, NY, 2010.
2. Sommerville, Ian. Software Engineering, Addison-Wesley, Boston, MA, 2011.
3. Stephens, Rod. Beginning Software Engineering, Wrox, 2015.
4. Tsui, Frank, Orlando Karam and Barbara Bernal. Essentials of Software Engineering, Jones & Bartlett Learning, Sudbury, MA, 2013.
5. Pfleeger, Shari. Sofwtare Engineering: Theory and Practice, Prentice Hall, Upper Saddle River, NJ, 2001.
6. Корнієнко Б.Я. Дослідження імітаційного полігону захисту критичних інформаційних ресурсів методом IRISK. Моделювання та інформаційні технології. 2018. Вип. 83. С. 34-41.
7. Корнієнко Б.Я. Побудова та тестування імітаційного полігону захисту критичних інформаційних ресурсів. Наукоємні технології. 2017. № 4 (36). С. 316 - 322.
8. Korniyenko B., Yudin A., Galata L. Risk estimation of information system.Wschodnioeuropejskie Czasopismo Naukowe. 2016. № 5. P. 35 - 40.
9. Корнієнко Б.Я., Юдін О.К., Снігур О.С.Безпека аутентифікації у web-ресурсах. Захист інформації. 2012. № 1 (54). С. 20 -25. DOI: 10.18372/2410-7840.14.2056 (ukr).
10. Корнієнко Б.Я., Максімов Ю.О., Марутовська Н.М. Прикладні програми управління інформаційними ризиками. Захист інформації. 2012. № 4 (57). С. 60 – 64. DOI: 10.18372/2410-7840.14.3493 (ukr).
11. Galata, L., Korniyenko, B., Yudin, A.: Research of the simulation polygon for the protection of critical information resources. In: CEUR Workshop Proceedings, Information Technologies and Security, Selected Papers of

the XVII International Scientific and Practical Conference on Information Technologies and Security (ITS 2017), 30 Nov 2017, Kyiv, Ukraine. vol. 2067. pp. 23–31., urn:nbn:de:0074-2067-8.

12. Корнієнко Б.Я. Безпека інформаційно-комунікаційних систем та мереж. Навчальний посібник для студентів спеціальності 125 «Кібербезпека». – К.: НАУ, 2018. – 226 с.
13. Корнієнко Б.Я., Галата Л.П. Оптимізація системи захисту інформації корпоративної мережі. Математичне та комп'ютерне моделювання. Серія: Технічні науки, Випуск 19, 2019. - С. 56-62.
14. Korniyenko B., Galata L. Implementation of the information resources protection based on the CentOS operating system. Conference Proceedings of 2019 IEEE 2nd Ukraine Conference on Electrical and Computer Engineering (UKRCON -2019) July 2 – 6, 2019, Lviv, Ukraine. - pp. 1007-1011.
15. Галата Л.П., Корнієнко Б.Я., Заболотний В.В. Математична модель протидії загрозам у системі захисту критичних інформаційних ресурсів. Наукоємні технології, Том 43, № 3, 2019. – С. 300 – 306.
16. Корнієнко Б.Я. Modeling of information security system in computer network. Безпека інформаційних систем і технологій, Том №1 (1), 2019. – С.36-41.
17. Korniyenko B., Galata L., Ladieva L. Research of Information Protection System of Corporate Network Based on GNS3. Conference Proceedings of 2019 IEEE Inernational Conference on Advanced Trends in Information Theory (IEEE ATIT -2019) Dezember 18 – 20, 2019, Kyiv, Ukraine. - pp. 244-248.
18. Korniyenko B., Galata L., Ladieva L. Mathematical model of threats resistance in the critical information resources protection system. CEUR Workshop Proceedings, Selected Papers of the XIX International Scientific

and Practical Conference "Information Technologies and Security" (ITS 2019) Kyiv, Ukraine, November 28, 2019. Vol-2577. P.281-291.

19. Корниенко Б.Я. Кибернетическая безопасность – операционные системы и протоколы. ISBN 978-3-330-08397-4, LAMBERT Academic Publishing, Saarbrucken, Deutschland. − 2017. – 122 с.
20. Korniyenko B.Y., Galata L.P. Design and research of mathematical model for information security system in computer network. Науковий журнал «Наукоємні технології». – 2017, № 2 (34), С. 114 - 118.
21. Корниенко Б.Я. Информационная безопасность и технологии компьютерных сетей : монография. ISBN 978-3-330-02028-3, LAMBERT Academic Publishing, Saarbrucken, Deutschland. − 2016. – 102 с.
22. Korniyenko B., Galata L., Kozuberda O. Modeling of security and risk assessment in information and communication system. Sciences of Europe. – 2016. – V. 2. – No 2 (2). – P. 61 -63.
23. Korniyenko B. The classification of information technologies and control systems. International scientific journal. – 2016. –№ 2. – P. 78 - 81.
24. Корнієнко Б.Я. Інформаційні технології оптимального управління виробництвом мінеральних добрив: монографія. – К.: Вид-во Аграр Медіа Груп, 2014. – 288 с.
25. Korniyenko B., Galata L., Ladieva L. Security Estimation of the Simulation Polygon for the Protection of Critical Information Resources / B. Korniyenko, //CEUR Workshop Proceedings, Selected Papers of the XVIII International Scientific and Practical Conference "Information Technologies and Security" (ITS 2018) Kyiv, Ukraine, November 27, 2018, Vol-2318, - P. 176-187, urn:nbn:de:0074-2318-4
26. Kornienko Y.M., Liubeka A.M., Sachok R.V., Korniyenko B.Y. Modeling of heat exchangement in fluidized bed with mechanical liquid distribution // ARPN Journal of Engineering and Applied Sciences. 2019. №14(12). P. 2203-2210.

27. Korniyenko B., Galata L., Ladieva L. Security Estimation of the Simulation Polygon for the Protection of Critical Information Resources // CEUR Workshop Proceedings, Selected Papers of the XVIII International Scientific and Practical Conference "Information Technologies and Security" (ITS 2018). Kyiv. Ukraine. 2018. №2318. P. 176-187.
28. Корнієнко Б.Я. Дослідження моделі взаємодії відкритих систем з погляду інформаційної безпеки // Наукоємні технології. 2012. №3(15). С. 83-89.
29. Korniyenko B., Yudin O., Novizkij E. Open systems interconnection model investigation from the viewpoint of information security // The Advanced Science Journal. 2013. №8. P. 53-56.
30. Zhulynskyi A.A., Ladieva L.R., Korniyenko B.Y. Parametric identification of the process of contact membrane distillation // ARPN Journal of Engineering and Applied Sciences Volume 14. 2019. №17. P. 3108-3112.
31. Korniyenko, B., Ladieva, L., Galata, L. Control system for the production of mineral fertilizers in a granulator with a fluidized bed. ATIT 2020 - Proceedings: 2020 2nd IEEE International Conference on Advanced Trends in Information Theory, 2020, pp. 307–310.
32. Корнієнко Б. Я., Галата Л. П. Дослідження імітаційного полігону захисту критичних інформаційних ресурсів методом IRISK. Моделювання та інформаційні технології. 2018. Вип. 83. С. 34–42.