Лабораторная работа №3. Проектирование и создание Web-сервисов с использованием Windows Communication Foundation (WCF), Node.js, JAX-WS и асинхронных коммуникаций.

Цель: научиться создавать сервисы с асинхронными коммуникациями на основе WCF, Node.js, JAX-WS.

Теоретические сведения.

WCF

WCF - это единая программная модель от Microsoft, предназначенная для создания сервис-ориентированных приложений. WCF позволяет разработчикам строить безопасные, надежные решения с поддержкой транзакций, которые могут взаимодействовать с различными платформами и уже существующими решениями.

Вообще, WCF предназначен для создания как сервис-ориентированных приложений, так и распределенных приложений. Никто не мешает нам создать одноранговую сеть с дуплексными связями на WCF. Но все-таки основной ориентацией WCF являются сервисы.

WCF поддерживает основные существующие на данный момент протоколы и технологии для передачи данных: HTTP/HTTPS, TCP, именованные каналы, MSMQ, и так далее. Взаимодействовать можно как угодно и с кем угодно. Модель обладает рядом положительных характеристик.

Во-первых, это надежные сеансы. Между клиентом и сервером устанавливается сеанс (session, почти как в ASP.NET) и WCF может гарантировать нам 100%-ую доставку сообщений (конечно, за счет производительности). Здесь есть 2 варианта:

1. Неупорядоченный сеанс - если сообщение потерялось, то после обнаружения потери оно будет запрошено заново и доставлено получателю. Сообщения, отправленные после потерянного сообщения, но до обнаружения потери, будут доставлены получателю до доставки потерянного сообщения.
2. Упорядоченный сеанс - как только обнаруживается пропажа, входящие сообщения ставятся в очередь на принимающей стороне и "замораживаются". Они не будут переданы получателю до передачи потерянного сообщения. Это гарантирует нам, что получатель примет сообщения именно в том порядке, в каком их послал отправитель. Это очень удобно, если нужно передавать большие объемы данных побуферно (например, мы хотим их еще шифровать, при этом скорость уже не так важна, поточность уже не поддерживается, а вот надежность и безопасность - на первом месте).

Во-вторых - безопасность. WCF поддерживает различные механизмы аутентификации и авторизации (Windows-авторизация, сертификаты, ASP.NET

membership). WCF также предлагает цифровую подпись сообщений (SHA-хэши) для гарантии целостности передаваемых данных и шифрование сообщений (TripleDes, Basic, RSA как "key wrap"-алгоритм) для защиты наших данных. Вообще, безопасность в WCF делится на два вида - уровня транспорта и уровня сообщений, но об этом в одной из следующих статей.

Основные концепции WCF. Рассмотрим сначала понятие **конечной точки** (это буквальный, но довольно распространенный перевод английского слова endpoint, дальше будем называть это **"точка соединения"**) - это абстрактный объект, от которого "уходят" данные, и к которому "приходят" данные (некий высокоуровневый аналог TCP-сокетов). Точка соединения объединяет в себе три "столпа WCF" - адрес, привязку и контракт (address, binding, connection - ABC, "азбука WCF").

Адрес. У точки соединения может быть только один адрес, у разных точек соединения может быть один базовый адрес (тогда адреса самих точек соединения будут задаваться относительно этого базового адреса). С помощью точки соединения адрес однозначно связывается с привязкой и контрактом **Привязка**, в свою очередь, определяет то, как наша точка соединения общается с окружающим миром. Она позволяет управлять такими вещами, как: сеанс, безопасность, поточность, транзакции, транспорт, кодировка сообщений. **Контракт** же, по сути, является **интерфейсом службы**, помеченным специальным атрибутом *ServiceContractAttribute*. Он определяет **требования** службы к безопасности, сеансу, задает параметры операций (начало сеанса, завершение сеанса, является ли операция односторонней, асинхронность операций на сервере).

WCF позволяет передавать любые объекты. Единственное требование - классы, которые мы опишем для использования при взаимодействии клиента со службой, должны быть помечены атрибутом *DataContractAttribute*, их поля и свойства - *DataMemberAttribute*, а члены перечислений - *EnumMemberAttribute*.

WCF предполагает несколько вариантов хостинга сервисов:

1. Хостинг в управляемом приложении.
2. Хостинг в управляемом сервисе Windows.
3. Хостинг с использованием Web-контейнера IIS.
4. Хостинг с использованием механизма Windows Activation Service (WAS).

Рассмотрим создание простого сервиса WCF, размещаемого в управляемом приложении и возвращающего приветствие на основании ввода пользователя.

Для этого, в первую очередь, создадим консольное приложение и определим контракт сервиса.

```
[ServiceContract]
public interface IHelloWorldService
{
    [OperationContract]
    string SayHello(string name);
}

public class HelloWorldService : IHelloWorldService
{
    public string SayHello(string name)
    {
        return string.Format("Hello, {0}", name);
    }
}
```

Создадим экземпляр класса Uri и зададим базовый адрес сервиса.

```
Uri baseAddress = new Uri("http://localhost:8080/hello");
```

Далее создадим экземпляр класса ServiceHost, передавая тип, представляющий тип сервиса и базовый URI ServiceHost. Разрешим передачу метаданных и подготовим сервис к приёму сообщений:

```
using (ServiceHost host = new ServiceHost(typeof(HelloWorldService), baseAddress))
{
    // Разрешим передачу метаданных.
    ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
    smb.HttpGetEnabled = true;
    smb.MetadataExporter.PolicyVersion = PolicyVersion.Policy15;
    host.Description.Behaviors.Add(smb);

    // Слушаем сообщения. Так как нет явно сконфигурированных
    // точек соединения, будет создано по отдельной точке на контракт
    host.Open();

    Console.WriteLine("The service is ready at {0}", baseAddress);
    Console.WriteLine("Press <Enter> to stop the service.");
    Console.ReadLine();

    // Закрываем ServiceHost.
    host.Close();
}
```

Протестировать работу сервиса можно с помощью приложения WCF Test Client.exe

Данный пример продемонстрировал односторонний (синхронный) обмен

информацией между сервером и клиентом. Часто возникают ситуации, когда требуется двухсторонняя асинхронная связь, то есть сервис должен отправить сообщение клиенту, когда тот его специально не ожидает. Часто с помощью такого интерфейса реализуются асинхронные уведомления клиента. Модель WCF позволяет реализовать такую связь с использованием двухстороннего контракта.

Двухсторонний контракт моделируется путем использования двух интерфейсов, связанных вместе с помощью ServiceContract. Ниже показан пример сервиса, моделирующего заказ пиццы с отображением прогресса её приготовления и уведомлением о готовности.

```
[ServiceContract(CallbackContract=typeof(IPizzaProgress))]
interface IOrderPizza
{
    [OperationContract]
    void PlaceOrder(string PizzaType);
}

interface IPizzaProgress {

    [OperationContract]
    void TimeRemaining(int minutes);

    [OperationContract]
    void PizzaReady();
}


class PingService : IOrderPizza
{
    IPizzaProgress callback;

    public void PlaceOrder(string PizzaType)  {

        callback = OperationContext.Current.GetCallbackChannel();

        Action preparePizza = PreparePizza;
        preparePizza.BeginInvoke(ar => preparePizza.EndInvoke(ar), null);

    }


    void PreparePizza()    {

        for (int i = 10 - 1; i >= 0; i--) {
            callback.TimeRemaining(i);
            Thread.Sleep(1000);
        }

        callback.PizzaReady();
    }
}
```

Жирным шрифтом показано получение callback-интерфейса и уведомление клиента.

Если клиенту необходимо иметь возможность получать сообщения от сервиса, он должен предоставить реализацию callback-контракта. Его определение может быть получено из метаданных или из сборки с общим контрактом. При использовании метаданных контракт обратной связи будет именован так же, как и контракт сервиса с суффиксом Callback.

```
class Program {

    static void Main(string[] args) {
        InstanceContext ctx = new InstanceContext(new Callback());

        OrderPizzaClient proxy = new OrderPizzaClient(ctx);
        proxy.PlaceOrder("Pepperoni");
        Console.WriteLine("press enter to exit");
        Console.ReadLine();
    }
}

class Callback : IOrderPizzaCallback {

    public void TimeRemaining(int minutes)  {
        Console.WriteLine("{0} seconds remaining", minutes);
    }

    public void PizzaReady()   {
        Console.WriteLine("Pizza is ready");
    }
}
```

OrderPizzaClient — класс-прокси, сгенерированный с помощью утилиты SysUtil, который будет наследовать DuplexClientBase<T> и контракт.

Node.js

Node или Node.js — программная платформа, основанная на движке V8 (транслирующем JavaScript в машинный код), превращающая JavaScript из узкоспециализированного языка в язык общего назначения. Node.js добавляет возможность JavaScript взаимодействовать с устройствами ввода-вывода через свой API (написанный на C++), подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода. Node.js применяется преимущественно на сервере, выполняя роль веб-сервера, но есть возможность разрабатывать на Node.js и десктопные оконные приложения (при помощи NW.js, AppJS или Electron для Linux, Windows и Mac OS) и даже программировать микроконтроллеры (например, tessel и espruino). В основе Node.js лежит событийно-

ориентированное и асинхронное (или реактивное) программирование с неблокирующим вводом/выводом.

Создание и запуск HTTP-сервера на Node.js, выдающего Hello, world! :

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

JAX-WS

Java API for XML Web Services (JAX-WS) — это прикладной программный интерфейс языка Java для создания веб-служб, являющийся частью платформы Java EE. JAX-WS является заменой технологии JAX-RPC, предоставляя более документо-ориентированную модель сообщений и упрощая разработку веб-служб за счёт использования аннотаций, впервые появившихся в Java SE 5. Технология JAX-WS является стандартом и описана в JSR 224.

Реализация класса сервиса на сервере.

В этом примере класс реализации, Hello, помечается как веб-служба конечной точки с использованием @WebService аннотации. В классе Hello объявлен один метод с именем SayHello, помеченный @WebMethod аннотацией, которая выставляет аннотированный метод для клиентов веб-служб. Метод SayHello возвращает приветствие клиенту, используя имя, переданное ему. Класс реализации должен также определить конструктор по умолчанию, публичный, без аргументов.

```
package helloservice.endpoint;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Hello {
    private String message = new String("Hello, ");

    public void Hello() {
    }

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

Реализация клиентского приложения

При вызове удаленных методов по порту, клиент выполняет следующие действия:

1. Использует сгенерированный helloservice.endpoint.HelloService класс, который представляет собой услугу в URI файле WSDL развернутого сервиса:

```
import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
      "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private static HelloService service;
```

2. Извлекает прокси к сервису, также известный как порт, путем вызова getHelloPort из сервиса:

```
helloservice.endpoint.Hello port = service.getHelloPort();
```

3. Запускает метод SayHello порта, передавая строку на сервер:

```
return port.sayHello(arg0);
```

Ниже приведен полный код HelloAppClient:

```
package appclient;

import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
      "META-INF/wsdl/localhost_8080/helloservice/HelloService.wsdl")
    private static HelloService service;

    /**
```

```
     * @param args the command line arguments
     */
    public static void main(String[] args) {
       System.out.println(sayHello("world"));
    }

    private static String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port = service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```

Асинхронная передача данных

Чтобы реализовать асинхронную связь с WCF необходимо выполнить одно из двух:

- использовать методы с именами *Async, которые уже сгенерированы (асинхронная модель на основе событий);
- изменить сигнатуры методов WCF с целью возврата Task<> (асинхронная модель на основе задач).

Реализация на основе событий:

При условии наличия операции Add(x1, x2) будет сгенерирован метод AddAsync и событие AddOperationCompleted. При этом его параметры будут описаны в классе AddCompletedEventArgs.

```
double value1 = 100.00D;
double value2 = 15.99D;
client.AddCompleted += new EventHandler<AddCompletedEventArgs>(AddCallback);
client.AddAsync(value1, value2);
Console.WriteLine("Add({0},{1})", value1, value2);
```

Результат будет получен локальной callback-функцией:

```
static void AddCallback(object sender, AddCompletedEventArgs e)
{
   Console.WriteLine("Add Result: {0}", e.Result);
}
```

Реализация на основе задач.

Контракт и служба.

```
[ServiceContract]
public interface IMessage
{
    [OperationContract]
    Task<string> GetMessages(string msg);
}
public class MessageService : IMessage
{
   async Task<string> IMessage.GetMessages(string msg)
   {
      var task = Task.Factory.StartNew(() =>
                                       {
                                           Thread.Sleep(10000);
                                           return "Return from Server : " + msg;
                                       });
     return await task.ConfigureAwait(false);
   }
}
```

Клиент.

```
class Program
 {
    static void Main(string[] args)
    {
       GetResult();
       Console.ReadLine();
    }

    private async static void GetResult()
    {
       var client = new Proxy("BasicHttpBinding_IMessage");
       var task = Task.Factory.StartNew(() => client.GetMessages("Hello"));
       var str = await task;
       str.ContinueWith(e =>
       {
          if (e.IsCompleted)
           {
              Console.WriteLine(str.Result);
```

```
            }
        });
      Console.WriteLine("Waiting for the result");
    }
 }
```

Полный пример с кодом проекта можно получить здесь: https://www.codeproject.com/Articles/613678/Task-based-Asynchronous-Operation-in-WCF

Пример реализации асинхронного обмена на JAX-WS

Данный пример демонстрирует веб-интерфейс с методами для асинхронных запросов с клиента:

```
@WebService

public interface CreditRatingService {
       // Синхронная операция
       Score getCreditScore(Customer     customer);
       // Асинхронная операция с голосованием
       Response<Score> getCreditScoreAsync(Customer customer);
       // Асинхронная операция с обратным вызовом
       Future<?> getQuoteAsync(Customer customer,
              AsyncHandler<Score> handler);
}
```

Реализация на клиенте:

Использование метода с обратным вызовом

```
CreditRatingService svc = ...;

Future<?> invocation = svc.getCreditScoreAsync(customerTom,
       new AsyncHandler<Score>() {
              public void handleResponse (
                     Response<Score> response)
                 {
                     score = response.get();
                     // обработка запроса
                 }
    }
);
```

Использование метода опроса

```
CreditRatingService svc = ...;
Response<Score> response = svc.getCreditScoreAsync(customerTom);

while (!response.isDone()) {
       // Делаем что-то пока ожидаем
}

score = response.get();
```

Пример реализации асинхронного обмена на Node.js:

Серверная часть. Допустим у нас есть список пользователей, хранимые в док-ом виде на сервере:

```
{
   "user1" : {
      "name" : "mahesh",
           "password" : "password1",
           "profession" : "teacher",
           "id": 1
   },
   "user2" : {
      "name" : "suresh",
           "password" : "password2",
           "profession" : "librarian",
           "id": 2
   },
   "user3" : {
      "name" : "ramesh",
           "password" : "password3",
           "profession" : "clerk",
           "id": 3
   }
}
```

Напишем запрос для получения всего списка пользователей:

```
var express = require('express');
var app = express();
var fs = require("fs");

// Возвращает весь список
app.get('/users', function (req, res) {
```

```
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       console.log( data );
       res.end( data );
   });
})
var server = app.listen(8081,’127.0.0.1’);
```

Реализация в клиентском приложении:

```
var xhr = new XMLHttpRequest();
// запрос на получение данных о пользователях
xhr.open('GET', ‘http://127.0.0.1:8081/users’ , true);

xhr.onreadystatechange = function() {
  if (xhr.readyState != 4) return;

  if (xhr.status != 200) {
// в переменную users сохраняем данные, которые получили с сервера
      var users = JSON.parse(xhr.responseText)
  }
}
xhr.send();
```

Порядок выполнения работы

В соответствии с индивидуальным заданием:

1. Определить функциональные границы веб-сервиса.
2. Определить контракт данных и функциональный контракт службы.
3. Создать сервис с контрактом для двухстороннего обмена информацией сервера и клиента.
4. Реализовать сервис с использованием WCF и протестировать его.

Задание

1. В соответствии с согласованным заданием (темой дипломного проекта или индивидуальным заданием из лабораторной работы №2) выполнить описание выполнения функций в виде Activity-диаграмм.
2. Выбрать одну из функций для демонстрации вызова метода с получением уведомления с использованием механизмов обратной связи.
3. Реализовать вышеуказанный сервис с использованием WCF, JAX-WS или Node.js развернуть его, как управляемое клиентское приложение с интерфейсом.
4. Создать клиентское приложение, с помощью которого продемонстрировать работу с сервисом и асинхронную обработку запросов (для любых технологий).