Лабораторная работа

Основой для выполнения этой работы послужило веб приложение на MVC 4.

По умолчанию в этом приложении уже есть логика по работе с пользователем. Авторизация, создание нового пользователя уже доступна. Так же доступны пару пунктов меню из которых в последствии можно сделать вызовы скрипта для прогона всех методов API.

В шаблоне в первую очередь заменили заголовки About,Contact на другие см. рисунок 1

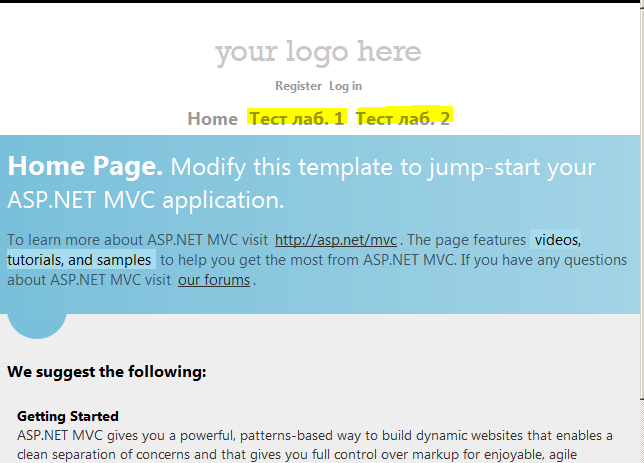


Рисунок 1 – Главная страница веб приложения.

Для реализации API используется технология WebAPI. В папке Controllers создан класс MainCotrollers.cs который содержит все методы API.  
Диаграмма классов для этого модуля выглядит так (см. рисунок 2)

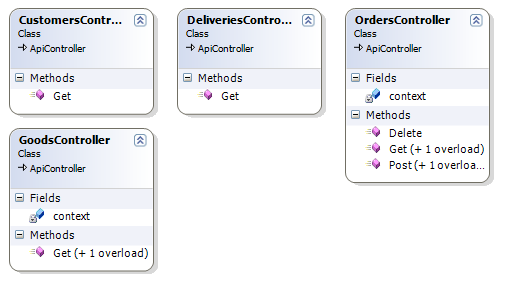


Рисунок 2 – Диаграмма классов модуля MainControlles.cs

Как видно из рисунка 2 на диаграмме представлены 4 основные сущности.

Customers (Покупатели), Deliveries (Доставки), Orders (Заказы), Goods (Товары).

Остальные таблицы представлены на рисунке 3.

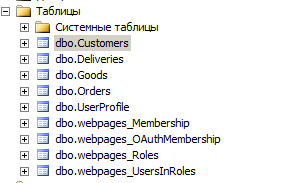


Рисунок 3 – Список таблиц базы данных

Таблицы UserProfile, webpages\_Membership, webpages\_OAuthMembership, webpages\_Roles, webpages\_UserInRoles. Это таблицы, которые были созданы шаблоном Web Application в среде Microsoft Visual Studio 2010. Таблицы для основных сущностей создаём в Microsoft Sql Management Studio.

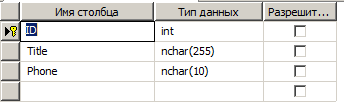


Рисунок 4 – Создание таблицы Customers

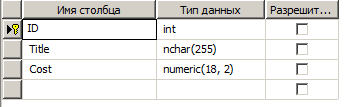


Рисунок 5 – Создание таблицы Deliveries

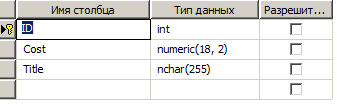


Рисунок 6 – Создание таблицы Goods

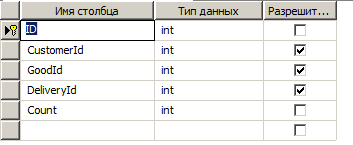


Рисунок 7 – Создание таблицы Orders

Также необходимо изменить стандартную таблицу UserProfile для поддержки авторизации OAuth 2.

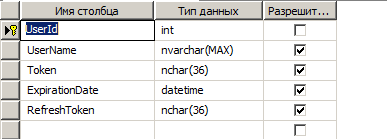


Рисунок 8 – Правка таблицы UserPofile  
В UserProfile добавляются поля Token, ExpirationDate,RefreshToken.

Для доступа к данным используется Entity Framework. Пакет для 6 версии данного фреймворка можно установить из Nget. Т.к. планируется использовать похожие модели и во второй лабораторной работе, то имеет смысл поместить все необходимые классы в отдельную библиотеку.

Для этого создадим проект ModelLib. И в класс MainModel.cs поместим следующий набор прокси классов для таблиц.

namespace ModelLib

{

[Table("UserProfile")]

public class UserProfile

{

[Key]

[DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]

public int UserId { get; set; }

public string UserName { get; set; }

public string Token { get; set; }

public string RefreshToken { get; set; }

public DateTime? ExpirationDate { get; set; }

}

[Table("Goods")]

public class Good

{

[Key]

[DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]

public int ID { get; set; }

public decimal Cost { get; set; }

public string Title { get; set; }

}

[Table("Deliveries")]

public class Delivery

{

[Key]

[DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]

public int ID { get; set; }

public decimal Cost { get; set; }

public string Title { get; set; }

}

[Table("Customers")]

public class Customer

{

[Key]

[DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]

public int ID { get; set; }

public string Title { get; set; }

public string Phone { get; set; }

}

[Table("Orders")]

public class Order

{

[Key]

// [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]

public int ID { get; set; }

public int? GoodId { get; set; }

public int? CustomerId { get; set; }

public int? DeliveryId { get; set; }

public int Count { get; set; }

}

}

В HomeController.cs в обработчик About внесём код, который будет проверять все вызовы Api и результат выведет на страницу. Особенностью представления About будет то, что она будет работать отображать результаты как для авторизованного апи так и для не авторизованного АПИ. На странице существует кнопка перехода

<a href = "/Account/Authorize?response\_type=code&client\_id=464119&redirect\_uri=http://localhost:60116/Home/About" > Авторизовать приложение по OAuth 2.0</a>

Данная кнопка инициализирует процесс авторизации по OAuth 2.0.

Как видно из URL view Authorize.cshtml принимает запрос. Чтобы это view отобразилось понадобиться авторизация пользователя. После успешной авторизации у пользователя запрашивается подтверждение на доступ по OAuth 2.0

@using (Html.BeginForm("AccessSuccess", "Account", new { redirect\_uri = ViewBag.redirect\_uri, client\_id = ViewBag.client\_id }))

{

<fieldset>

<input type="submit" value="Подтвердить" />

</fieldset>

}

Принимает ответ контроллер AccessSuccess (в модуле AccountController.cs). Контролер перенаправляет на redirect\_uri запрос добавляя в параметры code, который равен имени пользователя. Когда контроллер About получает не пустое значение code инициализируется процесс получения токена авторизации. По url

[имя\_сервера]/api/oauth/token делается post запрос с

Grant\_type = authorization\_code, client\_id =464119, client\_secret= “deadbeef”, code=[код из параметров].

За выдачу токена отвечает контроллер OauthController.cs

public class OauthController : ApiController

{

[Route("~/api/oauth/token")]

public AuthParams Post(TokenParams tokenParams)

{

AuthParams par = new AuthParams();

// создать access\_token

using (UsersContext context = new UsersContext())

{

UserProfile profile = null;

if (tokenParams.grant\_type == "authorization\_code")

{

profile = context.UserProfiles.Where(x => x.UserName == tokenParams.code).FirstOrDefault();

}

else

{

profile = context.UserProfiles.Where(x => x.RefreshToken == tokenParams.refresh\_token).FirstOrDefault();

}

if (profile != null)

{

par.access\_token = Guid.NewGuid().ToString();

par.token\_type = "bearer";

par.refresh\_token = Guid.NewGuid().ToString();

par.expires\_in = 60;

profile.Token = par.access\_token;

profile.RefreshToken = par.refresh\_token;

profile.ExpirationDate = DateTime.Now.AddSeconds(par.expires\_in);

// сохраняем параметры доступа по токену

context.SaveChanges();

}

}

return par;

}

}

В случае успешности операции будет возвращён набор для дальнейшей работы с методами API.

public class AuthParams

{

public string access\_token { get; set; }

public string token\_type { get; set; }

public string refresh\_token { get; set; }

public int expires\_in { get; set; }

}

Эти параметры сохраняются в сессии для дальнейшего использования (в том числе для обновления ключа авторизации когда его срок будет истекать).

Вызов всех методов API осуществляется посредством WebRequest

В HomeController.cs за это отвечает метод

GetResult(string serviceUrl, string method = "GET", Dictionary<string, string> param = null)

Он возвращает строку – результат выполнения операции.

Проверяет доступность вызова API класс MyOAuth и его метод CheckAccessToken , который выбрасывает исключение в случае неуспешной проверки.

Результат проверки работы АПИ без авторизации изображен на рисунке 9

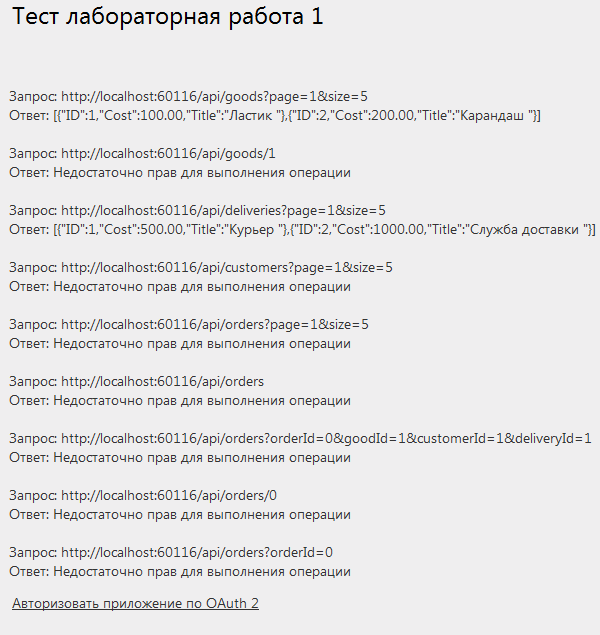


Рисунок 9 – Результат вызова API монолитного сервиса

После успешной авторизации по OAuth 2.0

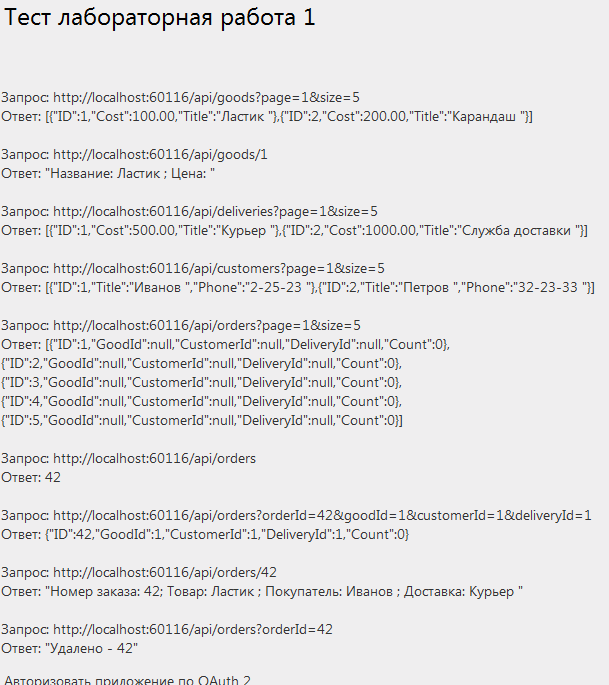


Рисунок 10 – Результат проверки API после успешной авторизации.

В лабораторной работе №3 необходимо было разбить монолитный сервис на несколько микросервисов. Во-первых, нужно было разделить данные по разным базам. Созданы дополнительно 3 базы. База с таблицей Customers и Deliveries. База с таблицей Goods и база с таблицей Orders. Во-вторых, созданы 3 микросервиса для работы с этими базами. Для реализации второй части (микросервисы) применена технология WCF сервиса с возможностью самостоятельно хостинга без IIS. За это отвечает приложение RunService.

На примере GoodService опишем принцип работы.

Сделаем сервис

[ServiceContract]

[ServiceBehavior(IncludeExceptionDetailInFaults = true)]

public class GoodsService

{

[OperationContract]

public string GetById(int id)

{

using (GoodsContext context = new GoodsContext())

{

return JsonConvert.SerializeObject(context.Goods.Where(x => x.ID == id).FirstOrDefault());

}

}

[OperationContract]

public string GetPage(int page, int size)

{

using (GoodsContext context = new GoodsContext())

{

return JsonConvert.SerializeObject(context.Goods.OrderBy(x => x.ID).Skip((page - 1) \* size)

.Take(size).ToList());

}

}

}

Данный сервис можно запустить командами

ServiceHost host = new ServiceHost(type, baseAddress);

ServiceMetadataBehavior smb = new ServiceMetadataBehavior();

smb.HttpGetEnabled = true;

smb.MetadataExporter.PolicyVersion = PolicyVersion.Policy15;

host.Description.Behaviors.Add(smb);

host.Open();

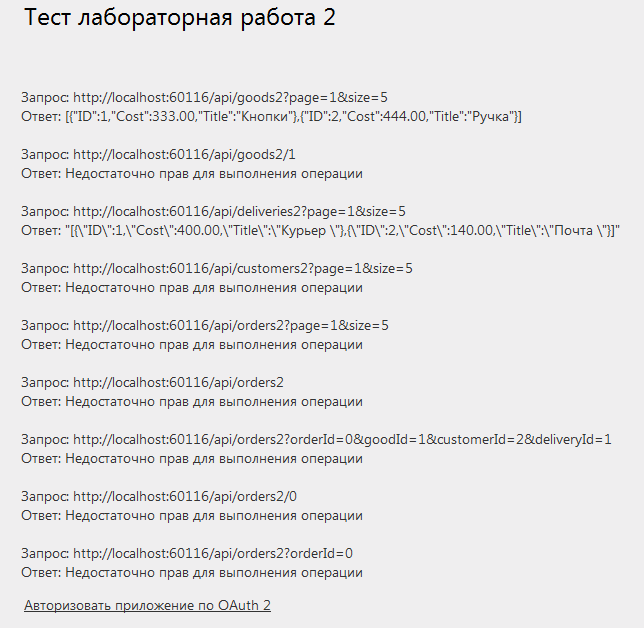
Это позволит обращаться к этому сервису по tcp/ip протоколу.

Консольное приложение содержит 3 микросервиса и после запуска три сервиса становяться активным и доступными для работы (запускать нужно под именем администратора).

По аналогии с классом MainControllers.cs сделаем класс MainControllers2.cs который будет содержать API для работы с микросервисами. Внешне набор API методов не изменился( только добавилась двойка например api/orders2). А вот внутри работа с данными идёт только через сервисы отдельной взятых сущностей.

Сервисы подключаются через диалог AddServiceRererence. По wsdl генерируется клиентские классы, через которые удобно обращаться к методам сервисов.

Процедуры авторизации те же, только страница для инициализации другая – contract, которая содержит всю логику тестирования. После вызова этой страницы получим результат см. рисунок 11.

  
Рисунок 11 – Запуск теста лабораторной работы 2 без авторизации

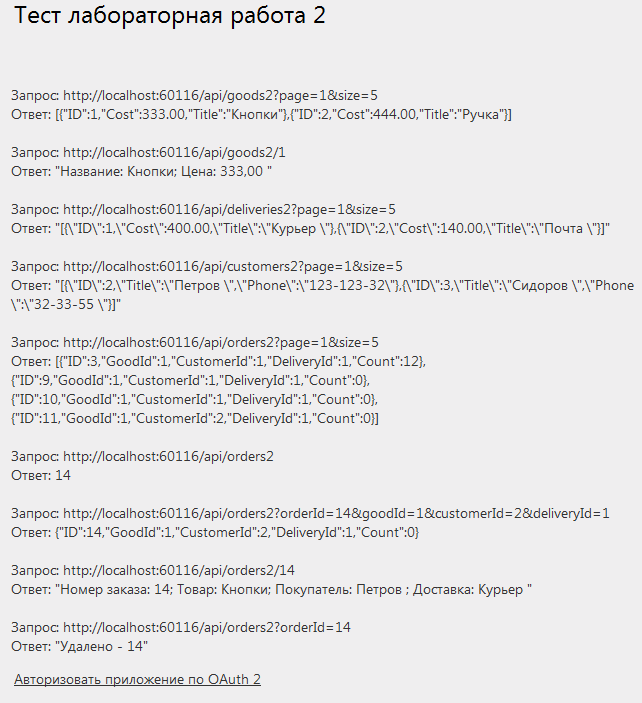


Рисунок 12 – Запуск теста для второй лабораторной работы с авторизацией

Если посмотреть внимательно на содержимое, то видно отличие по данным, т.к. данные для лабораторной работы №1 и №2 берутся из разных источников

Добавлены:

1)

Нижеследующий код получает строку таблицы Goods по ID потом превращает объект Good в строку формата JSON:

public string GetById(int id)

{

using (GoodsContext context = new GoodsContext())

{

return JsonConvert.SerializeObject(context.Goods.Where(x => x.ID ==

id).FirstOrDefault());

}

}

2)

Добавлены нижеследующие дополнительные сервисы:

дожен быть выделен сервис агрегатор (должен быть запрос

на 2 сущности, находящиеся на разных микросервисах), и должен быть выделен сервис проверки сессии.

Orders2Controller метод Get(int id) – аггрегатор.

Новый сервис проверки сессии. Метод MyOAuth.CheckAccessToken -> MyOAuthService

3)

«MvcApplication1/Controllers/AccountController.cs» относится:

К обоим частям, во второй лабораторной работе используется такая же схема работы с автризацией ( за исключением добавления жизни токена).

4)

работа должна быть так, есть 5 сервисов:

a) агрегатор информации, все запросы api идут только на него (

htmlи прочее хранится на нем, это некоторый front-end)

есть

б) сессия - сервис хранит информацию о пользователе и проверяет авторизацию

есть MyOAuthService

с) order/goods/customer - отдельные сервисы

И каждый сервис в отдельном модуле, связи между модулями только по restful api

Есть

Соответствие требованиям

1. Данные каждого сервиса можно хранить как в SQL, так и в

NoSQLбазе. Для упрощения допускается хранить данные на одной базе, но в разных схемах. При этом каждый сервис должен взаимодействовать только со своей схемой, получение данных, не относящихся к текущему сервису строго запрещено.

3 базы MS SQL и каждый микросервис общается со своей базой.

2. Нельзя использовать готовые библиотеки для авторизации по

OAuth2.0.

Сделано в ручную

3. Для токена нужно реализовать время жизни (expires) и обновление токена (через refresh token).

Сделано HomeController/AuthMethod

4. Должен быть хотя бы один запрос, требующий агрегированной информации от двух и более сервисов.

Orders2Controller метод Get(id)

5. Все взаимодействие между сервисами выполнить в парадигме

RESTful.

Исправлено. Есть

6. Предусмотреть работу системы в случае отказа одного из компонентов системы.

В случает отказа микросервиса данные из других сервисов будут читаться.

7. При получении списков данных предусмотреть пагинацию.

Есть

8. Сделать подробное логгирование выполняемых действий н

а каждом сервисе.

Есть. В конфиге Nlog.config в каждом проекте указан файл в который пишутся конфиги. Сейчас он везде один – C:\log.txt.

В него пишут все микрофреймворки выполняя методы.

9. Подготовить шаблоны запросов или маленький скрипт для демонстрации работы.

Есть тестовая веб страница