Молдавский государственный университет

Факультет математики и информатики

Департамент Информатики

**Курсовая работа**

Разработка RESTful API приложений с использованием

ASP.NET WEB API Core и языка программирования C#

**444.1 Информатика**

Выполнена студентом II курса,

специальности Информатика

**Чобану Артёмом, группа I1902**

Руководитель, преподаватель

Департамента Информатики

**Георгий Латул**

Кишинэу – 2021

Оглавление

[Введение 3](#_Toc70337541)

[Глава 1 4](#_Toc70337542)

[Глава 2 6](#_Toc70337543)

[Глава 3 6](#_Toc70337544)

[Заключение 6](#_Toc70337545)

[Литература и ссылки 6](#_Toc70337546)

[Вторая часть практики: Shadowing 6](#_Toc70337547)

[Литература: 6](#_Toc70337548)

[Заключение: 7](#_Toc70337549)

# Введение

На данный момент **веб-сервисы** – это наиболее часто создаваемый тип приложений. Без них не обходится ни один проект, даже самый мелкий. Кроме того, в крупных проектах несколько общающихся между собой веб-сервисов, объединённых в одно приложение. В последнее время идея веб-сервисов развивается в сторону микросервисной архитектуры – множества небольших веб-сервисов, объединённых в качестве модуля одного большего приложения.

В течение последних двадцати лет, было созданно несколько технологий и протоколов для создания веб-сервисов. Одним из ранее принятых способов было создание ASMX и WCF (Windows Communication Foundation) сервисов, использующих протокол SOAP (Simple Object Access Protocol).

На данный момент наиболее актуальным является создание **RESTful Web API** веб-сервисов, использующих протокол **HTTP** (HyperText Transfer Protocol). Это самый распространённый протокол передачи данных в современное время, который изначально был предназначен для передачи гипертекстовых документов.

Протокол HTTP в качестве дальнейшего развития получил расширение **HTTPS** (HyperText Transfter Protocol Secure), поддерживающего шифрование, благодаря чему является безопаснее.

**Цель данной работы** – создать пример наиболее правильной имплементации **WEB API** проекта на языке на платформе ASP.NET и с использованием языка программирования **C#**. В качестве примера будет создан веб-сервис, использующийся для новостного веб-сайта. Веб-сервис будет взаимодействовать с другими приложениями посредством HTTP **GET** и **POST** запросов.

# Глава 1

1.1 Назначение и структура HTTP запроса.

**HTTP (HyperText Transfer Protocol)** – протокол 7 (прикладного) уровня сетевой модели OSI, изначально разработанный для доступа к гипертекстовым документам. Из-за этого, основными реализациями **клиентов** является браузеры.

HTTP реализует архитектуру “**Клиент-Сервер**”, в которой:

* **Клиент** – заказчик услуг, который создаёт подключение к серверу, посылает запрос и получает ответ.
* **Сервер** – поставщик услуг, принимающий запросы от клиента, и отправляющий ответ, если это необходимо.

Структура HTTP запроса:

* **Стартовая строка** (Request line – строка запроса) – содержит метаданные, связанные с конфигурацией запроса, такие как метод, версия HTTP, URI и другие.
* Например: “*GET /posts/ HTTP/1.1*”
* **Поля заголовка/Заголовок** (Header fields) – параметры запроса. Зачастую через них передаются и метаданные. Например: “Language: RU”, “Encoding: UTF-8”
* **Тело запроса** (Request body) – сами данные, отделяющиеся от заголовка пустой строкой. Тело не является обязательным. Часто данные передаются в формате **JSON**.

Последовательность сетевых транзакций “Запрос-Ответ” называется **сессией**. Вызов любого HTTP **метода** равноценен одной сессии.

Самые часто использующиеся методы HTTP:

1. **GET** – запрос на получение данных. Может использоваться в качестве команд для сервера.

Пример GET запроса с параметром limit: /posts/get?limit=10

1. **POST** – используется для отправки данных серверу. Данные помещаются в тело запроса, вследствие чего тело является обязательным для POST запросов.

Структура ответа:

**Код состояния** (Status Code) – трёхзначное целое число, возвращаемое сервером в ответе. Они делятся на 5 групп:

1. 1xx – Информирование о процессе передачи (Informational).
2. 2xx – Успешные (Successful).

Например:

200 – Ok – Запрос выполнен успешно.

1. 3xx – Перенаправле ние (Redirection).
2. 4xx – Клиентская ошибка (Client Error).

Например:

404 – Не найдено (Not Found) – ресурса по указанному URL не существует.

400 – Неверный запрос (Bad Request) – клиент неверно сформировал запрос.

1. 5xx – Ошибка сервера (Server Error).

500 – Internal Server Error – внутренняя серверная ошибка в процессе обработки запроса.

1.2 Архитектура REST.

**REST** **(Representational State Transfer)** – архитектурный стиль, использующийся для создания веб-сервисов. Веб-сервисы, следующие принципам данного архитектурного стиля, называются **RESTful**.

REST предоставляет пять **обязательных** ограничений:

1. Архитектура “**Клиент-Сервер**” – подразумевает **разделение ответственностей** и **независимость** между клиентскими и серверными приложениями, что позволяет осуществлять их разработку параллельно.

Независимость клиентской части упрощает её переносимость на другие платформы. Его работоспособность не зависит от платформы или операционной системы. Сервер сможет одинаково работать с любыми приложениями, если присланные запросы удовлетворяют API веб-сервиса. Более того, серверу будет невозможно получить такую информацию о клиенте, как платформа, если это не предусмотрено на уровне спецификации API.

Пример преимущества независимости серверной части – это возможность повысить **масштабируемость** системы.

1. **Отсутствие состояния** (Statelessness) – Так как один HTTP запрос осуществляется подключением, передачей данных, получением ответа от сервера и дальнейшим разрывом соединения, веб-сервис **не имеет** такого понятия как “**сессия**” пользователя. Каждый запрос должен содержать достаточно данных для осуществления операций. Этот принцип также отражается и в названии архитектурного стиля.

Пример из созданного мною проекта – передача **токена авторизации** в заголовке:

“Authorization”: “Bearer {токен}”

Где “Bearer” – название используемой в проекте схемы авторизации.

Токен позволяет индентифицировать пользователя, и совершить необходимые операции.

1. **Кэширование** (Cacheability) – ответы сервера должны иметь указание, являются ли они кэшированными.
2. **Система слоёв** (Layered System) – возможность использования нескольких веб-сервисов в таких целях как распределение нагрузки. В таком случае API также не должен меняться. Клиент не должен иметь возможность определить к какому из сервисов был послан запрос: к промежуточному узлу или напрямую.
3. **Единообразие интерфейса** (Uniform Interface) – Клиент и сервер должны использовать один интерфейс. Веб-сервис должен предоставлять спецификацию собственных **Веб-Ресурсов**, в которые входят URI, формат передачи данных и другие.
4. **Код по требованию** (Code On Demand) – необязательное ограничение, подразумевающие расширяемость клиентского функционала засчёт получения исполняемого кода с сервера.

**Веб-Ресурс** – это уникальный узел веб-сервиса, идентифицированный собственным URI.

**URI** (Uniform Resource Identifier) – Унифицированный идентификатор ресурса. URI включает в себя:

* **URL** (Universal Resource Locator) – адрес ресурса в сети (https://newsWebsite.com)
* **URN** (Universal resource Name) – имя ресурса в сети (accounts/register)

URI с данными URL и URN выглядит следующим образом: https://newsWebsite.com/accounts/register

1.3 Семантика HTTP методов, спецификации REST и форматы обмана данными.

Семантика HTTP методов и их **CRUD** (Create, Read, Update, Delete) эквиваленты:

|  |  |  |
| --- | --- | --- |
| **HTTP Метод** | **CRUD эквивалент** | **Описание** |
| GET | Read | Возвращает данные, состояние о запрашиваемом ресурсе. |
| POST | Create | Передача данных о каком-либо ресурсе. |
| PUT | Update | Обновление данных об указанном ресурсе. |
| DELETE | Delete | Удаление данных об указанном в запросе ресурсе. |

**Официально рекомендуемые** принципы для разработки API Веб-Сервиса:

* **GET /ресурс** – Получение списка ресурса.

Пример: “/posts” - получение списка новостей.

* **GET /ресурс/{идентификатор ресурса}** – Получение данные о ресурсе с данным идентификатором.

Пример: “/posts/8091db53-6938-4934-b75d-137ca30c1156” – получение новости с данным идентификатором.

* **POST /ресурс** – Создание нового ресурса
* **PUT /ресурс/{идентификатор ресурса}** – Обновление ресурса
* **DELETE /ресурс/{идентификатор ресурса}** – Удаление ресурса.

Названия ресурсов рекомендуется называть во **множественном** числе: /users, /accounts, /posts.

На практике преимущественно используются **GET** и **POST** методы, так как с их помощью возможно реализовать любую операцию. Использование тех или иных методов, а также официальных спецификаций – это решение, принимаемое разработчиками проекта. Например, **API Dropbox** имеет узел “/users/get\_account”, который по официальным рекомендациям должен выглядеть следующим образом: ”/users/{userID}”

Выбор **формата представления данных** зависит от задач и решения каждой команды разработчиков. Самым популярным форматом для представления объектов является **JSON**.

**JSON (JavaScript Object Notation)** – формат текстового представления данных, отличительная особенность которого – простота, благодаря чему он легко читается людьми. Формат был изначально разработан для языка программирования **JavaScript**, но из-за простоты используется повсеместно.

Пример JSON массива объектов:

**[**

**{**

**"id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",**

**"title": "Обновление версии веб-сайта до 1.1",**

**"content": "Новое обновление доступно для установки."**

**},**

**{**

**"id": "8091db53-6938-4934-b75d-137ca30c1156",**

**"title": "Технические работы 4/30/2021",**

**"content": "В связи с обновлением проводятся технические работы."**

**}**

**]**

Очевидно, что данные можно передавать в других форматах (Например XML), и даже в “сыром” (raw) виде. Например, если данные – это изображение, то зачастую “сырой” формат гораздо удобнее.

1.4 Авторизация с помощью JWT.

**JSON Web Token** **(JWT)** – Открытый стандарт безопасной передачи данных в виде json объекта. Токен хранит всю необходимую информацию для передачи, т.е. является **атономным** (Self-Contained).

**Структура JWT:**

* **Заголовок** (Header) – содержит следующие метаданные:

**alg** (Algorithm) – алгоритм, используемый для генерации подписи.

**typ** (Type) – Тип токена.

**{**

**"alg": "HS256",**

**"typ": "JWT"**

**}**

* **Полезная нагрузка** (Payload) – данные, с целью передачи которых создаётся токен. (Claims)

Существует несколько стандартных и рекомендуемых полей:

**iss** (Issuer) – определяет создателя токена.

**exp** (Expiration time) – срок действия.

**sub** (Subject) – Информация о предмете, которому принадлежат передаваемые данные.

**aud** (Audience) – Список получателей токена.

Остальные поля определяются пользователем токена. Например, в моём проекте токен содержит иденификатор пользователя и его роль:

**{**

**"UserId": "a7482551-5334-41a5-8821-a68f25d115d6",**

**"Role": "Admin",**

**"exp": 1620049249,**

**"iss": "NewsWebsiteMainAPI",**

**"aud": "NewsWebsiteAudience",**

**}**

* **Подпись** (Signature) – отвечает за валидацию токена.

Перечисленные три части токена разделены точками, в следствие чего токен выглядит следующим образом: **{Заголовок}.{Полезная нагрузка}.{Подпись}**

Перед тем как выполнить запрос, Веб-сервис осуществляет **валидацию** токена. Для этого, токен должен находится в Заголовке “Authorization”.

**Схемы Аутентификации:**

В проекте использовалась “**Bearer**” схема Аутентификации, использующая токен безопасности. Данная схема обьясняется как “Аутентификация носителя токена”. Любая информация, необходимая для аутентификации содержится в заголовке **Authorization.** Сама аутентификация должна выглядить следующим образом: **{тип} {данные для входа}**

# Глава 2

2.1 Планирование спецификации API проекта.

**Функционал Веб-сервиса:**

* Регистрация и авторизация с использованием ролей, благодаря чему в них будут разные полномочия.
* Использование JWT для авторизации без использования логина и пароля:



Отправка JWT клиенту:



* Добавление, чтение и удаление новостей. Очевидно, что обычные пользователи имеют доступ только к чтению.

**Endpoint Спецификация проекта:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ресурс **Accounts** | | **Путь** | | |
| /accounts/login | /accounts/register | /accounts/get\_account |
| **Запрос** | **Заголовок** | - | - | **Authorization**:  Bearer {токен} |
| **Тело** | {  “**Email**”: string,  “**Password**”: string  } | {  “**Email**”: string,  “**Password**”: string,  “**FullName**”: string  } | - |
| **Ответ** | **Тело** | string (токен) | string (токен) | { “**FullName**”: string } |
| **Коды Состояния** | 200, 400, 401 | 200, 400, 409 | 200, 400, 401 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ресурс **Posts** | | **Путь** | | |
| /posts/add | /posts | /posts /{postId} |
| **Запрос** | **Заголовок** | **Authorization**:  Bearer {токен} | **Authorization**:  Bearer {токен} | **Authorization**:  Bearer {токен} |
| **Тело** | {  “**Title**”: string,  “**Content**”: string  } | Массив из объектов:  {  “**Title**”: string,  “**Content**”: string,  “**Date**”: string  } | {  “**Title**”: string,  “**Content**”: string,  “**Date**”: string  } |
| **Ответ** | **Тело** | string (токен) | string (токен) | { “FullName”: string } |
| **Коды Состояния** | 200, 400, 401, 403 | 200, 401 | 200, 401 |

* 200 – **Ok** – Успешное выполнение.
* 400 – **Bad Request** – Клиент неверно сформировал запрос.
* 401 – **Unauthorized** – Пользователь не прошёл этап авторизации.
* 403 – **Forbidden** – Пользователь не имеет достаточно прав.
* 409 – Conflict – Невозможно совершить операцию из-за конфликта, который может быть решён **пользователем**. В данном случае – попытка использования уже существующего Email.

# Глава 3

фыв

# Заключение

фыв

# Литература и ссылки

фыв

# Вторая часть практики: Shadowing

ФВ

# Литература и документация:

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication>

https://swagger.io/docs/specification/authentication/bearer-authentication/