Молдавский государственный университет

Факультет математики и информатики

Департамент Информатики

**Курсовая работа**

Разработка RESTful API приложений с использованием платформы ASP.NET 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](#_Toc74495502)

[Глава 1 – RESTful API веб-сервисы. 4](#_Toc74495503)

[Глава 2 – Проработка бпроекта. 8](#_Toc74495504)

[Глава 3 – Реализация проекта. 16](#_Toc74495505)

[Заключение 27](#_Toc74495506)

[Литература и документация: 28](#_Toc74495507)

# 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.

Репозиторий на GitHub: <https://github.com/ArtiomCiobanu/NewsWebsite>

# Глава 1 – RESTful API веб-сервисы.

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:

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

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

* **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:

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

Например:

200 – Ok – Запрос выполнен успешно.

* 3xx – Перенаправление (Redirection).
* 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.

* 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 обмена 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 **автономным** (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.** Сами данные в заголовке должны выглядеть следующим образом: **{тип} {данные для входа}**

# Глава 2 – Проработка проекта.

2.1 Планирование 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.

2.2 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 |
|  | **Метод** | POST | POST | GET |
|  | **Описание** | Авторизация пользователя по Логику и Паролю | Регистрация пользователя | Получение данных пользователя |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ресурс **Posts** | | **Путь** | | |
| /posts/add | /posts/get/{limit} | /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 |
|  | **Метод** | POST | GET | GET |
|  | **Описание** | Добавляет новость с данными Заголовком и Содержимым | Возвращает последние новости в количестве не более чем limit. | Возвращает данные о новости с данным идентификатором |

* 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.

2.3 Object Relational Mapping

В проекте используется **Entity Framework** – стандартная ORM библиотека платформы .NET.

**Object Relational Mapping (ORM)** – конвертирование объектов определённой системы типов в объекты Объектно-Ориентированного языка программирования. В данном случае – объекты базы данных в экземпляр класса, с помощью которого представляется модель данных.

Класс, состоящий только из полей, являющимися представляемой сущности называется **Моделью**.

**Преимущества и недостатки использования ORM:**

Использование ORM позволяет разместить логику получения данных в виде кода на используемом языке программирования, используя все его преимущества. Также, такой код является более понятным для разработчика проекта. Кроме того, это часто помогает избежать дублирования кода.

Разумеется, преимущества, полученные благодаря использованию ORM не покрывают все нужды проекта. Один из самых больших минусов ORM – это необходимость **развёртывания** (deployment) приложения после каждого изменения, что не даёт возможность быстро исправлять ошибки. Во многих приложениях вызываются хранимые процедуры, написанные на SQL, так как они не имеют этого недостатка.

Необходимо отметить, что многие разработчики, регулярно работающие с SQL часто ощущают недостаток возможностей при использовании ORM. Например, язык SQL содержит оператор **PIVOT**, являющийся удобнее для определённого вида задач.

2.4 Шаблоны проектирования Репозиторий и Сервис

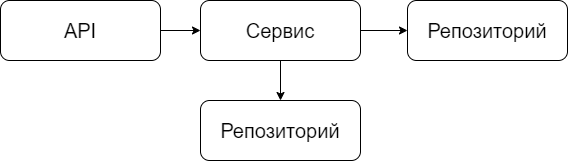
Говоря об ORM библиотеках, важно заметить, что хоть во многих платформах существуют стандартные (как в случае с .NET – Entity Framework), но не все разработчики используют только одну из них. Более того, в процессе разработки и поддержки проекта может возникнуть необходимость изменения, и даже полного отказа от ORM. Поэтому, разработчики используют шаблон проектирования (паттерн) **Репозиторий**.

**Репозиторий** – это структурный паттерн, предназначенный для инкапсуляции ORM при помощи CRUD операций. Данный шаблон проектирования создаёт дополнительный слой абстракции между между слоями бизнес-логики и доступа к данным. Очевидно, что это пример следования принципу **Инверсии Зависимостей**, уменьшающего **зацепление** в программе. Благодаря этому, слой бизнес-логики зависит от интерфейса – то есть только от CRUD операций, а не от конкретной ORM. Это позволяет с лёгкостью заменить или полностью отказаться от ORM.

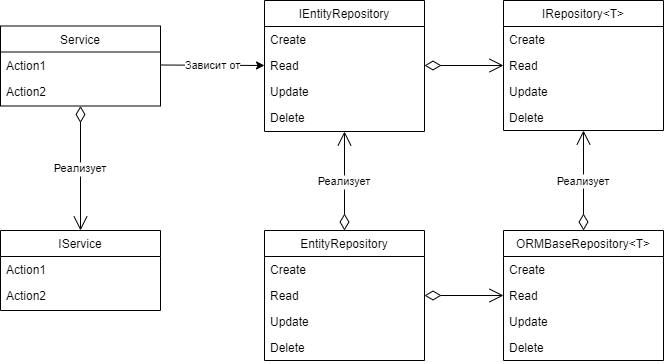


**Бизнес-логика** также должна находиться в отдельном слое проекта. Вызывающий код не должен содержать бизнес-логику. Это позволяет улучшить читаемость кода, а также определить ответственности вызывающего кода, о чём будет сказано позже.

**Сервис** – это структурный паттерн, находящийся в слое бизнес-логики проекта. Зачастую один сервис использует несколько репозиториев. Сервис имеет несколько методов, каждый из которых представляет одну команду или запрос, связанные с бизнес-процессами.



Комбинация паттернов Репозиторий и Сервис в виде иерархии классов и интерфейсов:



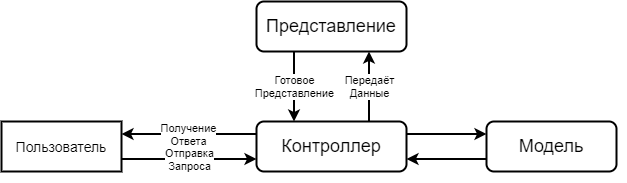
Дополнительное преимущество использования данных паттернов является возможность создания для них **Unit-тестов**. Следуя принципу инверсии зависимости, вызывающий код будет зависеть от интерфейса. Имея интерфейс, можно создать **мок** (фальшивый объект) сервиса или репозитория и благодаря этому протестировать поведение слоёв, которые их используют.

В случае с Web API приложением в качестве **вызывающего кода** выступают методы API **Контроллера.**

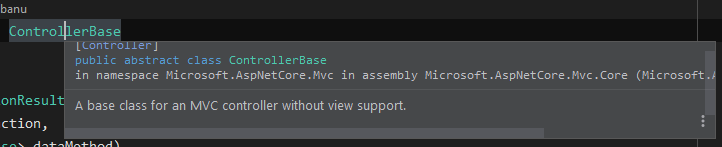
2.5 Model-View-Controller

**Model-View-Controller (MVC)** – шаблон проектирования, разделяющий ответственности приложения на три части:

* **Модель** (Model) – данные, необходимые для выполнения запроса.
* **Представление** (View) – данные, возвращаемые контроллером после выполнения запроса.
* **Контроллер** (Controller) – получает запросы от пользователя в виде модели, выполняет его и возвращает представление.



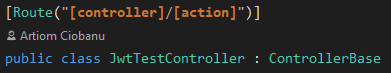
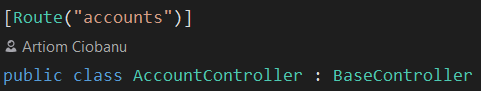
.NET имеет встроенный фреймворк, реализующий паттерн MVC. В данном фреймворке Контроллер – это класс, наследующийся от класса **ControllerBase**. Каждый метод контоллера представляет собой отдельный HTTP запрос, возвращающий представление.



Стандартные атрибуты для настройки конфигурации контроллера и его методов:

* **Route** – возможно использование и для контроллеров, и для каждого метода по-отдельности. Параметр атрибута определяет шаблон URN. Возможно непосредственное использование имён контроллера или метода, а также параметра.

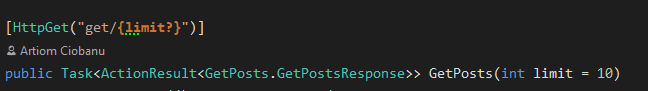
Примеры:

Методы контроллера будут вызваны по адресу**: JwtTest/{Название метода}**

* **HttpGet, HttpPost** и др. – Атрибуты, позволяющие указать тип HTTP запроса. Существует перегрузка, принимающая адрес метода, благодаря чему можно не использовать предыдущий атрибут для методов.

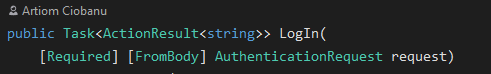
Пример:



Метод GetPosts будет вызван по следующему адресу: **posts/get/{количество}**

Допускается вызов без указания параметра, так как существует значение по умолчанию.

* **Authorize** – использует авторизацию, добавленную в **middleware** проекта.
* **Required** – вместо вызова метода, возвращает BadRequest в случае отсутствия параметра.



* **FromBody, FromQuery, FromUri, FromRoute, FromHeader** – осуществляют принудительную привязку части HTTP запроса, из которой будет получен параметр
* **ApiController** – применяет конфигурацию из **middleware** проекта, а также позволяет использовать атрибуты, такие как FromBody, FromQuery.

2.6 Внедрение зависимостей.

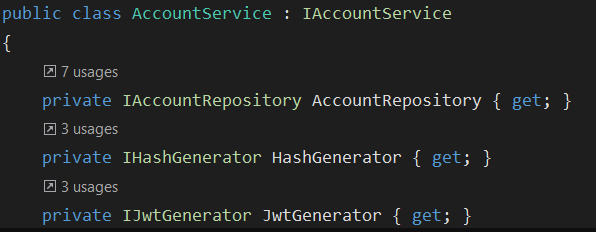
**Внедрение зависимостей** (Dependency Injection) – процесс передачи объектов другим объектам, зависящих от них.

Внедрение зависимостей можно осуществлять прямой передачей объекта в конструктор, поле и свойства. Из-за того, что в больших проектах существует огромное количество классов, становится очень трудно правильно организовать этот процесс. Для решения этой проблемы были созданы **фреймворки для Внедрения Зависимостей**.

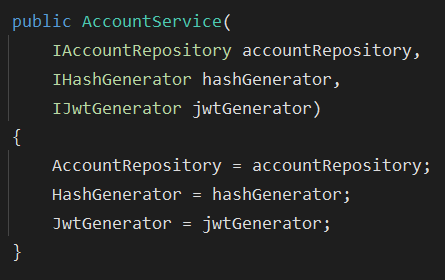
Я использовал стандартный фреймворк для ASP.NET Core, так как он покрывает большую часть потребностей при разработке проекта. Кроме него часто используется **Autofac**.

Пример:

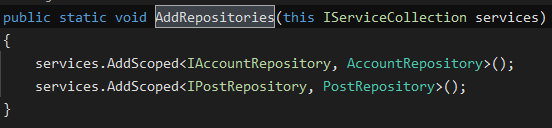
Класс **AccountService** зависит от трёх интерфейсов: **IAccountRepository**, **IHashGenerator** и **IJwtGenerator**.



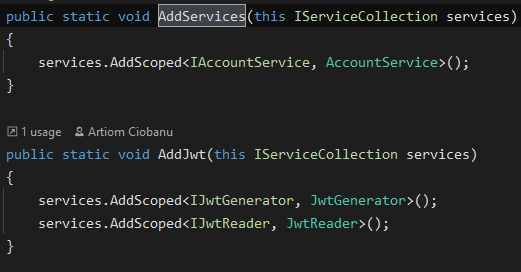
Имеется конструктор, получающий их:



Добавление Репозиториев в **контейнер** зависимостей:



Добавление Сервисов и классов, работающих с JWT:



Таким образом, классы зарегистрированы в качестве имплементации интерфейса, которая будет использоваться. Далее контейнер будет автоматически передавать зависимости объектам, которые тоже должны быть добавлены в контейнер.При возникновении необходимости использовать другую имплементацию одного из интерфейсов, необходимо будет зарегистрировать её вместо старой. Очевидно, что важен порядок регистрации зависимостей.

В зависимости от необходимого жизненного цикла, зависимости могут быть добавлены как:

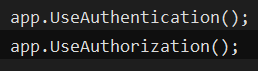
* **Transient** – создаётся новый объект для каждого обращения.
* **Scoped** – один объект для каждого запроса.
* **Singleton** – один объект на всё приложение.

2.7 Middleware

**Middleware** – промежуточное Программное обеспечение.

Говоря о внедрении зависимостей и авторизации необходимо рассказать про механизм middleware в ASP.NET Core. Он из себя представляет часть приложения, которая вызывается до начала выполнения запроса, и настраивается в классе Startup.

В методе Configure определяется последовательность вызова каждого компонента:



2.8 Принцип CQS и паттерн CQRS

**Command Query Separation** (CQS) – принцип, гласящий, что каждый метод должен быть либо **командой**, либо **запросом**. Если метод является и тем, и другим одновременно это считается нарушеним данного принципа.

**Команда** – это метод не возвращающей значение. Команды изменяют состояние ресурсов.

**Запрос** – метод, возвращаемый значение. Запросы предназначены для чтения состояния ресурсов, но не для его изменения. Метод является запросом если он является ссылочно-прозрачной функцией, а также не допускает побочных эффектов.

**Ссылочная прозрачность** – это свойство выражения, котором его замена на значение, возвращаемое этим выражением при данных аргументах, не влияет на работу программы.



Необходимо отметить, что соблюдение принципа CQS не всегда возможно. К примеру, структура данных **Стек** имеет метод **Pop**, одновременно возвращающий и удаляющий значение из стека. В данном случае нет смысла разделять эти ответственностями между двумя методами.

CQS не применим для методов, которые должны выполнить несколько операций в **атомарном виде**.

**Command Query Responsibility Segregation** (CQRS) – развитие принципа CQS, говорящий о том, что команды и запросы должны использовать **разные модели**. Таким образом, все команды и запросы становятся полностью самостоятельными.

CQRS представляется как альтернатива паттерну Сервис. Он тоже находится в слое бизнес-логики проекта.

**CQRS** выделяет каждый метод в **отдельный класс**, использующий отдельную модель, в то время как **Сервис** представляет собой **один класс**, имеющий множество методов.

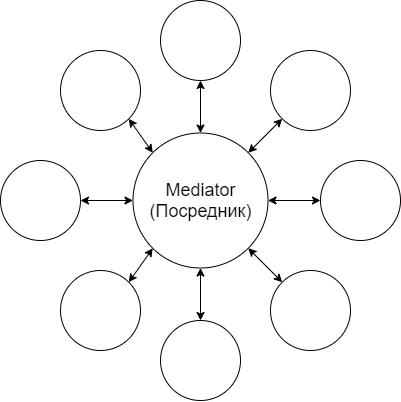
В небольших и простых приложениях Сервис является предпочтительнее. В больших – разделение ответственностей и моделей на команды и запросы увеличивает читаемость кода, а также позволяет увеличить производительность.

В моём проекте используются **оба** шаблона проектирования.

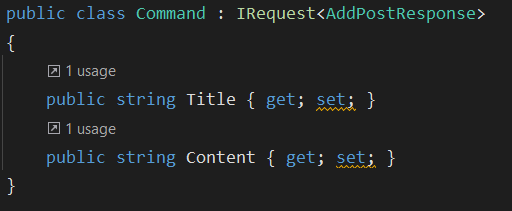
2.9 Шабон проектирования Mediator

В качестве имплементации паттерна CQRS я использовал паттерн Mediator и библиотеку MediatR.

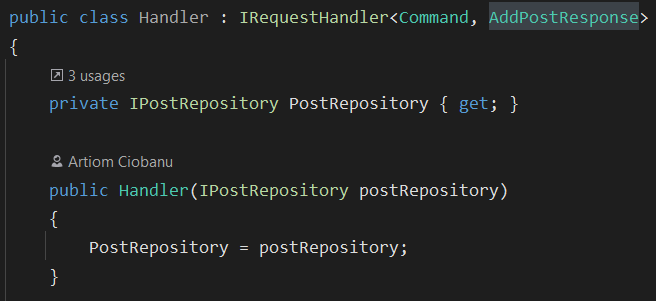
**Mediator** (Посредник) – поведенческий паттерн, в котором все объекты связаны лишь с одним объектом, который называется **Посредником**. Цель паттерна состоит в избегании взаимных ссылок между множеством объектов.



Для моделей запроса/ответа существует маркерный интерфейс **IRequest**<T>



Для создания класса-обработчика запроса он должен реализовать интерфейс **IRequestHandler**<**TCommand**, **TResponse**>



Отправка запроса или команды объекту – посреднику:



Посредник находит обработчик запроса и вызывает его.

# Глава 3 – Реализация проекта.

3.1 Создание репозиториев

Сначала был создан интерфейс Репозитория:

public interface IRepository<T>  
{  
 public Task CreateAsync(T entity);  
 public Task UpdateAsync(T entity);  
 public Task<T> GetAsync(Guid id);  
 public Task DeleteAsync(Guid id);  
  
 public Task SaveChangesAsync();  
  
 public Task<bool> ExistsWithIdAsync(Guid id);  
}

К CRUD операциям я добавил методы **ExistsWithIdAsync** для проверки существования записи и **SaveChangesAsync**, применяющий изменения.

Базовый репозиторий для Entity Framework:

public class EfBaseRepository<TEntity> : IRepository<TEntity>  
 where TEntity : BaseEntity  
{  
 private DbContext EntityContext { get; }  
 private DbSet<TEntity> EntityDbSet => EntityContext.Set<TEntity>();  
  
 public EfBaseRepository(DbContext entityContext)  
 {  
 EntityContext = entityContext;  
 }  
  
 public async Task CreateAsync(TEntity entity)  
 {  
 await EntityDbSet.AddAsync(entity);  
 }  
 public Task UpdateAsync(TEntity entity)  
 {  
 return Task.FromResult(EntityDbSet.Update(entity));  
 }  
 public async Task<TEntity> GetAsync(Guid id)  
 => await EntityDbSet.FindAsync(id);  
 public async Task DeleteAsync(Guid id)  
 {  
 var entity = await GetAsync(id);  
  
 EntityDbSet.Remove(entity);  
 await EntityContext.SaveChangesAsync();  
 }  
 public Task SaveChangesAsync()  
 => EntityContext.SaveChangesAsync();  
 public Task<bool> ExistsWithIdAsync(Guid id)  
 => EntityDbSet.AnyAsync(entity => entity.Id == id);  
}

Интерфейс репозитория для аккаунтов:

public interface IAccountRepository : IRepository<Account>  
{  
 public Task<Account> GetWithEmailAsync(string email);  
 public Task<bool> ExistsWithEmailAsync(string email);  
}

Я добавил методы **GetWithEmailAsync** и **ExistsWithEmailAsync** для проверки существования аккаунта с указанным email.

Репозиторий аккаунтов:

public class AccountRepository: EfBaseRepository<Account>,IAccountRepository  
{  
 private AccountContext AccountContext { get; }  
  
 public AccountRepository(AccountContext context) : base(context)  
 {  
 AccountContext = context;  
 }  
  
 public Task<Account> GetWithEmailAsync(string email)  
 => AccountContext.Accounts.FirstOrDefaultAsync(account =>

account.Email == email);  
  
 public Task<bool> ExistsWithEmailAsync(string email)  
 => AccountContext.Accounts.AnyAsync(account => account.Email == email);  
}

Репозиторий новостей:

public interface IPostRepository : IRepository<Post>  
{  
 public Task<Post[]> GetTopAsync(int amount);  
}

Реализация репозитория:

public class PostRepository : EfBaseRepository<Post>, IPostRepository  
{  
 private PostContext PostContext { get; }  
  
 public PostRepository(PostContext postContext) : base(postContext)  
 {  
 PostContext = postContext;  
 }  
 public async Task<Post[]> GetTopAsync(int amount)  
 {  
 var result = PostContext.Posts.OrderByDescending(

x => x.Title).Take(amount).ToArray();  
 return await Task.FromResult(result);  
 }  
}

3.2 Создание сервисов

Сервис Аккаунтов, который использует репозиторий. Он имеет методы Регистрации, Аутентификации и получения данных аккаунта.

public interface IAccountService  
{  
 public Task<TokenResponse> RegisterAsync(

RegistrationRequest registrationRequest);  
 public Task<TokenResponse> LogInAsync(

AuthenticationRequest authenticationRequest);  
 public Task<AccountResponse> GetIfExists(Guid id);  
}

Для регистрации испольузуется класс RegistrationRequest, имеющий поля email, имя и пароль:

public class RegistrationRequest  
{  
 public string Email { get; set; }  
 public string FullName { get; set; }  
 public string Password { get; set; }  
}

Для аутентификации используется класс AuthenticationRequest, имеющий email и пароль:

public class AuthenticationRequest  
{  
 public string Email { get; set; }  
 public string Password { get; set; }  
}

Реализация интерфейса:

public class AccountService : IAccountService  
{  
 private IAccountRepository AccountRepository { get; }  
 private IHashGenerator HashGenerator { get; }  
 private IJwtGenerator JwtGenerator { get; }  
  
 public AccountService(  
 IAccountRepository accountRepository,  
 IHashGenerator hashGenerator,  
 IJwtGenerator jwtGenerator)  
 {  
 AccountRepository = accountRepository;  
 HashGenerator = hashGenerator;  
 JwtGenerator = jwtGenerator;  
 }

public async Task<TokenResponse> RegisterAsync(

RegistrationRequest registrationRequest)  
 {  
 if (await AccountRepository

.ExistsWithEmailAsync(registrationRequest.Email))  
 {  
 return TokenResponse.Unauthorized();  
 }  
  
 var passwordHash = await HashGenerator

.GenerateSaltedHash(registrationRequest.Password);  
  
 var account = new Account  
 {  
 Id = Guid.NewGuid(),  
 Email = registrationRequest.Email,  
 FullName = registrationRequest.FullName,  
 RoleId = (int) UserRole.User,  
 PasswordHash = passwordHash  
 };  
  
 await AccountRepository.CreateAsync(account);  
 await AccountRepository.SaveChangesAsync();  
  
 var token = JwtGenerator.GenerateToken(account.Id);  
 return TokenResponse.Success(token);  
 }  
  
 public async Task<TokenResponse> LogInAsync(

AuthenticationRequest authenticationRequest)  
 {  
 var account = await AccountRepository

.GetWithEmailAsync(authenticationRequest.Email);  
  
 if (account != null)  
 {  
 var passwordHash = await HashGenerator

.GenerateSaltedHash(authenticationRequest.Password);  
  
 if (account.PasswordHash == passwordHash)  
 {  
 var role = (UserRole) account.RoleId;  
 var token = JwtGenerator.GenerateToken(account.Id, role);  
  
 return TokenResponse.Success(token);  
 }  
 }  
  
 return TokenResponse.Unauthorized();  
 }

public async Task<AccountResponse> GetIfExists(Guid id)  
 {  
 if (await AccountRepository.ExistsWithIdAsync(id))  
 {  
 var account = await AccountRepository.GetAsync(id);  
  
 return AccountResponse.Success(account.FullName);  
 }  
  
 return AccountResponse.Unauthorized();  
 }  
}

Сервис зависит от интерфейсов репозитория аккаунтов, хэш-генетратора и jwt-генератора. При регистрации создаётся хэш пароля, а JWT возвращается и методом регистрации, и авторизации.

Пароль записывается в базу данных в виде хэша. Метод авторизации заново генерирует хэш и сравнивает полученный хэш с имеющимся в базе данных.

3.3 Генерация хэша и JWT

Интерфейс хэш-генератора:

public interface IHashGenerator  
{  
 public Task<string> GenerateSaltedHash(string inputText);  
}

Реализация, использующая MD5:

public class HashGenerator : IHashGenerator  
{  
 private HashConfiguration Configuration { get; }  
 public HashGenerator(HashConfiguration configuration)  
 {  
 Configuration = configuration;  
 }  
 public async Task<string> GenerateSaltedHash(string inputText)  
 {  
 using HashAlgorithm hashAlgorithm = new MD5CryptoServiceProvider();  
 await using var inputStream = new MemoryStream();  
 var streamWriter = new StreamWriter(inputStream);  
  
 string saltedText = inputText + Configuration.Salt;  
  
 await streamWriter.WriteAsync(saltedText);  
  
 var hash = await hashAlgorithm.ComputeHashAsync(inputStream);  
 return Encoding.Unicode.GetString(hash);  
 }  
}

Интерфейс jwt-генератора:

public interface IJwtGenerator  
{  
 string GenerateToken(Guid userId, UserRole userRole = UserRole.User);  
}

Класс JwtGenerator создаёт токен и записывает в него **идентификатор** и **роль** пользователя.

public class JwtGenerator : IJwtGenerator  
{  
 private IJwtConfigurator JwtConfigurator { get; }  
 public JwtGenerator(IJwtConfigurator jwtConfigurator)  
 {  
 JwtConfigurator = jwtConfigurator;  
 }  
 public string GenerateToken(Guid userId, UserRole userRole)  
 {  
 var jwtConfiguration = JwtConfigurator.Configuration;  
 var signingCredentials = JwtConfigurator.GetSigningCredentials();  
  
 var currentDateTime = DateTime.UtcNow;  
  
 var jwt = new JwtSecurityToken(  
 jwtConfiguration.Issuer,  
 jwtConfiguration.Audience,  
 new[]  
 {  
 new Claim(AppClaimTypes.UserId, userId.ToString()),  
 new Claim(AppClaimTypes.UserRole, userRole.ToString())  
 },  
 currentDateTime,  
 currentDateTime.AddDays(jwtConfiguration.TokenLifetime),  
 signingCredentials);  
  
 return new JwtSecurityTokenHandler().WriteToken(jwt);  
 }  
}

Доступные роли:

public enum UserRole  
{  
 User,  
 Admin  
}

Интерфейс IJwtConfigurator:

public interface IJwtConfigurator  
{  
 JwtConfiguration Configuration { get; }  
 TokenValidationParameters ValidationParameters { get; }  
 SigningCredentials GetSigningCredentials();  
}

Класс JWT конфигуратора:

public class JwtConfigurator : IJwtConfigurator  
{  
 public JwtConfiguration Configuration { get; }  
  
 public JwtConfigurator(JwtConfiguration jwtConfiguration)  
 {  
 Configuration = jwtConfiguration;  
 }  
  
 public TokenValidationParameters ValidationParameters => new()  
 {  
 ValidateIssuer = true,  
 ValidIssuer = Configuration.Issuer,  
 ValidateAudience = true,  
 ValidAudience = Configuration.Audience,  
 ValidateLifetime = true,  
 RequireExpirationTime = true,  
 IssuerSigningKey = GetSymmetricSecurityKey(),  
 ValidateIssuerSigningKey = true  
 };  
  
 public SigningCredentials GetSigningCredentials() =>  
 new (GetSymmetricSecurityKey(),

SecurityAlgorithms.HmacSha256Signature);  
  
 private byte[] GetByteKey() =>

Encoding.UTF8.GetBytes(Configuration.SecretKey);

private SymmetricSecurityKey GetSymmetricSecurityKey()

=> new(GetByteKey());  
}

Конфигурация хранится в ASP.NET веб-приложениях хранится в файле **appsettings.json**:

"JwtConfiguration": {  
 "Issuer": "FMIWebSiteMainAPI",  
 "Audience": "TestAudience",  
 "TokenLifetime": 7,  
 "SecretKey": "9z$C&F)J@NcRfTjWnZr4u7x!A%D\*G-Ka"  
},

Здесь же хранится и соль для хэша, и строка подключения к базе данных:

"HashConfiguration": {  
 "Salt": "TestSalt"  
},  
"ConnectionStrings": {  
 "DefaultConnectionString": "Data source=(LocalDB)\\MSSQLLocalDB;Initial catalog=NewsWebsite;Integrated Security=True;"  
}

3.4 Создание контроллеров

Для того, чтобы Сервис возвращал код состояния был создан перечисляемый тип **ResponseStatus**:

public enum ResponseStatus  
{  
 Success = 200,  
 Created = 201,  
 Accepted = 202,  
 NoContent = 204,  
 PartialContent = 206,  
 BadRequest = 400,  
 Conflict = 409,  
 NotFound = 404,  
 Unauthorized = 401,  
 Forbidden = 403,  
 TooManyRequests = 429,  
 InternalServerError = 500  
}

Базовый класс ответа имеет статус ответа:

public class BaseResponse   
{  
 public ResponseStatus Status { get; set; }  
}

Все ответы выглядят аналогично классу **GetPostsResponse**:

public class GetPostsResponse : BaseResponse  
{  
 public Post[] Posts { get; set; }  
 public static GetPostsResponse Success(Post[] posts) =>  
 new()  
 {  
 Posts = posts,  
 Status = ResponseStatus.Success  
 };  
 public static GetPostsResponse NoContent() =>  
 new()  
 {  
 Status = ResponseStatus.NoContent  
 };  
}

Пример использования:

return posts.Any() ?  
 GetPostsResponse.Success(posts) :  
 GetPostsResponse.NoContent();

Базовый класс контроллера, с методом **ExecuteAction** который получает **BaseResponse**, и возвращает действительный код состояния в зависимости от значения ResponseStatus:

public class BaseController : ControllerBase  
{  
 protected async Task<ActionResult<TResponse>>

ExecuteAction<TResult, TResponse>(  
 Func<Task<TResult>> action,  
 Func<TResult, TResponse> dataMethod)  
 where TResult : BaseResponse  
 {  
 var result = await action();  
 var response = dataMethod(result);  
  
 ActionResult<TResponse> actionResult = result.Status switch  
 {  
 ResponseStatus.InternalServerError =>

StatusCode(StatusCodes.Status500InternalServerError, response),  
 ResponseStatus.Success => Ok(response),  
 ResponseStatus.BadRequest => BadRequest(response),  
 ResponseStatus.Conflict => Conflict(response),  
 ResponseStatus.NoContent =>

StatusCode(StatusCodes.Status204NoContent, response),  
 ResponseStatus.NotFound => NotFound(response),  
 ResponseStatus.Unauthorized => Unauthorized(response),  
 ResponseStatus.Accepted => Accepted(response),  
 ResponseStatus.PartialContent =>

StatusCode(StatusCodes.Status206PartialContent, response),  
 ResponseStatus.Forbidden =>

StatusCode(StatusCodes.Status403Forbidden, response),  
 ResponseStatus.Created =>

StatusCode(StatusCodes.Status201Created, response),  
 ResponseStatus.TooManyRequests =>

StatusCode(StatusCodes.Status429TooManyRequests),  
 \_ => throw new ArgumentOutOfRangeException(  
 $"{result.Status}",  
 "Should be a valid HTTP Status Code")  
 };  
 return actionResult;  
 }

3.5 Внедрение зависимостей

Внедрение зависимостей происходит в методе ConfigureServices в классе Startup:

services.AddDbContextForConnectionString<AccountContext>(  
 Configuration.GetConnectionString("DefaultConnectionString"));  
services.AddDbContextForConnectionString<PostContext>(  
 Configuration.GetConnectionString("DefaultConnectionString"));  
  
services.AddScoped<IPostContext, PostContext>();  
services.AddScoped<IAccountContext, AccountContext>();  
  
services.AddGenerators(Configuration);  
services.AddJwt();  
services.AddRepositories();  
services.AddServices();

Большая часть зависимостей внедряется в методах расширения:

public static void AddRepositories(this IServiceCollection services)  
{  
 services.AddScoped<IAccountRepository, AccountRepository>();  
 services.AddScoped<IPostRepository, PostRepository>();  
}  
public static void AddServices(this IServiceCollection services)  
{  
 services.AddScoped<IAccountService, AccountService>();  
}  
public static void AddJwt(this IServiceCollection services)  
{  
 services.AddScoped<IJwtGenerator, JwtGenerator>();  
 services.AddScoped<IJwtReader, JwtReader>();  
}

Получение конфигурации из appsettings.json:

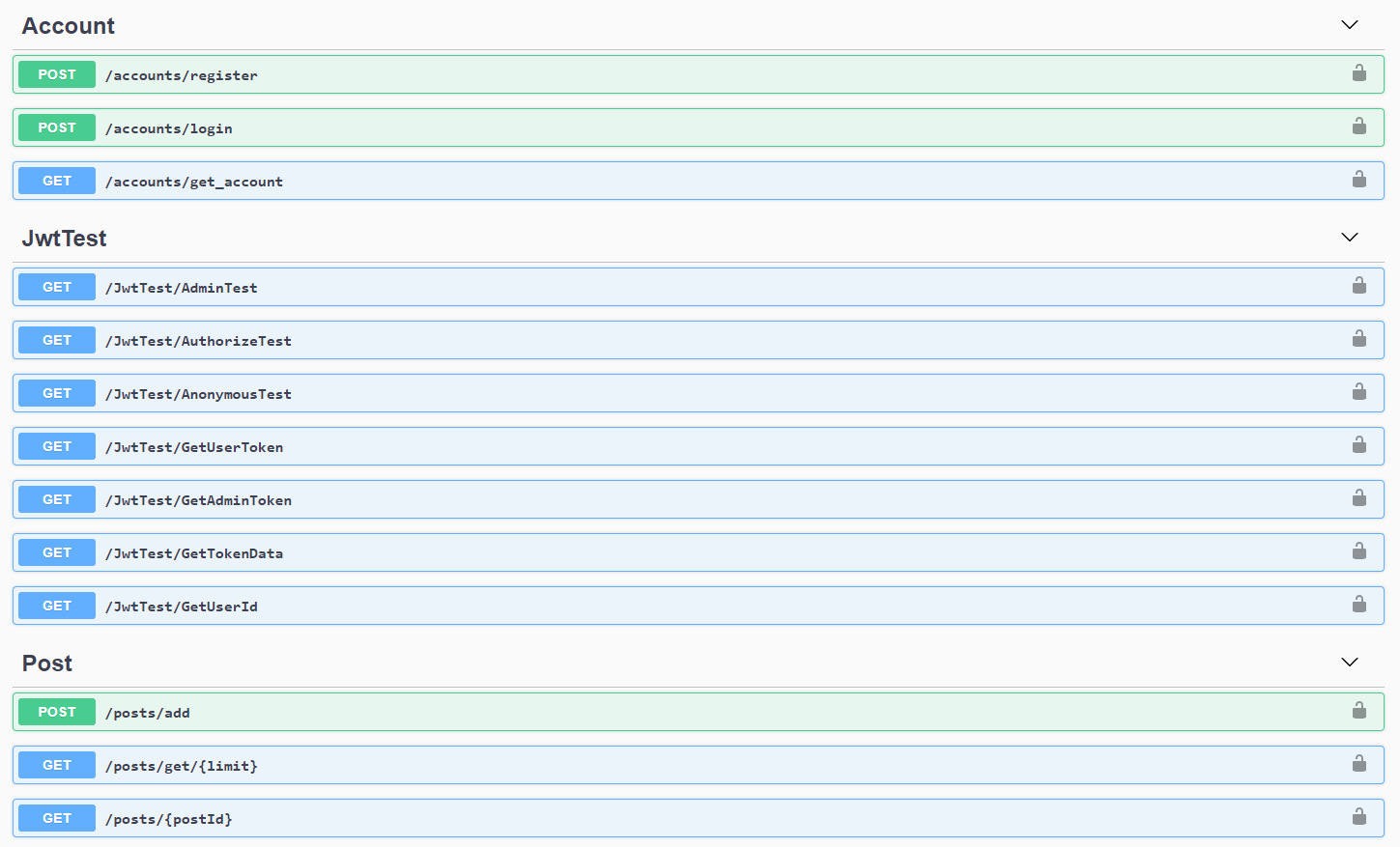
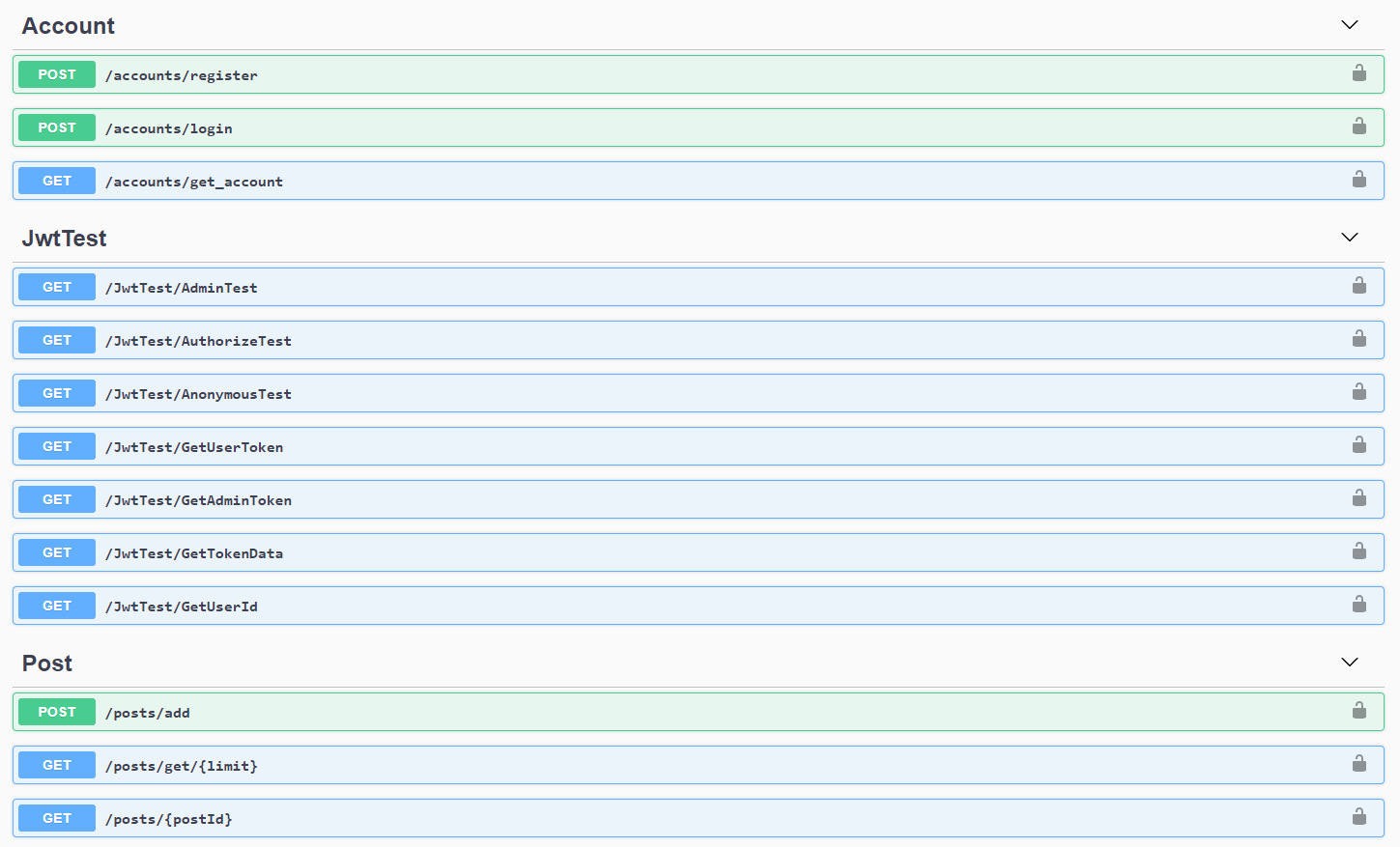
services.AddSingleton<IJwtConfigurator, JwtConfigurator>(\_ =>  
{  
 var jwtConfiguration = configuration  
 .GetSection(SettingsSections.JwtConfiguration)

.Get<JwtConfiguration>();  
  
 return new JwtConfigurator(jwtConfiguration);  
});

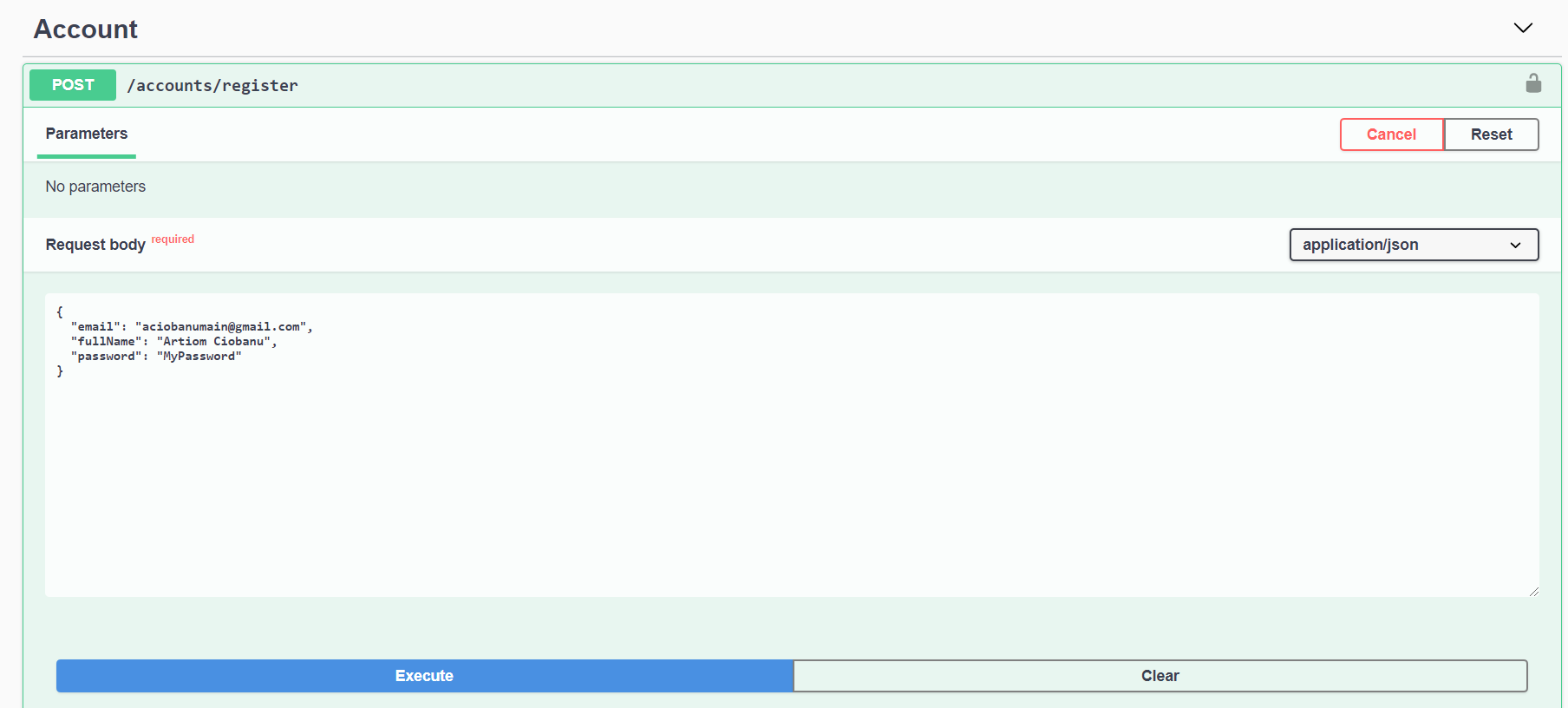
3.6 Демонстрация работы Веб-сервиса

Для тестирования я использовал **Swagger UI** – инструмент, автоматически генерирующий визуальную API спецификацию проекта. Он позволяет отправлять запросы, создаёт примеры.

Все API узлы проекта:



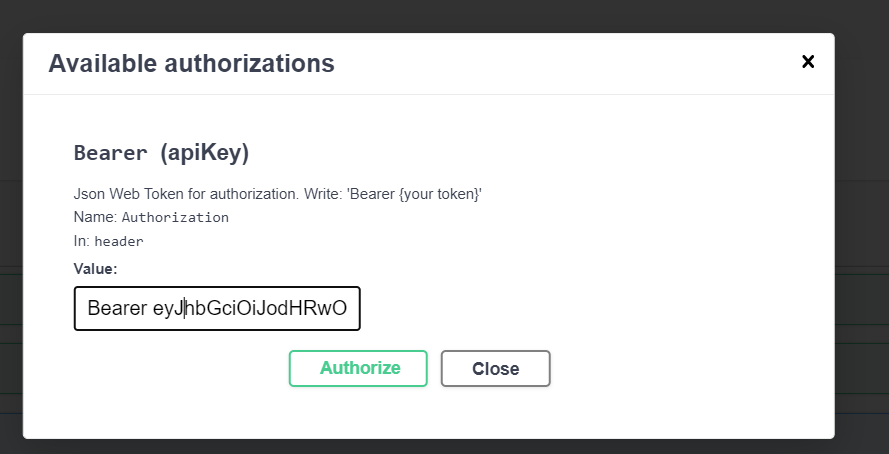
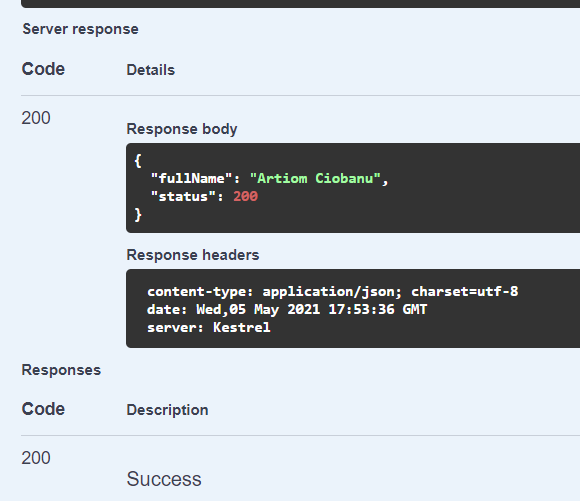
Регистрация:



Тело ответа содержит JWT:



Получим данные аккаунта, передав токен в заголовке:

# Заключение

Во время работы были изучены следующие темы:

* Архитектура REST
* Шаблоны проектирования:

1. Репозиторй
2. Сервис
3. Посредник
4. Model-View-Controller (MVC)

* Принцип CQS и паттерн CQRS
* Внедрение зависимостей
* Авторизация с помощью JWT
* Object Relational Mapping, Entity Framework
* Middleware
* Swagger UI
* Инверсия Управления

Реализованный функционал:

* Регистрация
* Авторизация
* Получение данных аккаунта
* Добавление и получение новостей

# Литература и документация:

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

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

<https://jwt.io/>

<https://github.com/jbogard/MediatR>

<https://docs.microsoft.com/en-us/ef/>

<https://docs.microsoft.com/en-us/aspnet/core/web-api>

<https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-5.0>

**Чистая архитектура** - Роберт К. Мартин

**Design Patterns via C# -** Александр Шевчук, Дмитрий Охрименко, Андрей Касьянов

**Паттерны проектирования на платформе .NET –** Сергей Тепляков