# 1. Что такое PhotoMania?

# 2. Разработка

### Введение

Наш проект будет состоять из двух частей – API и клиент. API (Application Programming Interface или интерфейс программирования приложений) — это совокупность инструментов и функций в виде интерфейса для создания новых приложений, благодаря которому одна программа будет взаимодействовать с другой, что позволяет разработчикам расширять функциональность своего продукта и связывать его с другими. Например, у нас есть сайт, БД к нему и все такое. Мы хотим расширить проект и сделать еще например мобильное приложение. Тогда нам нужно для него откуда-то брать данные, и кроме как с API, больше никак не получится брать данные из одной БД на несколько разных приложений (связать приложения между собой). Или API понадобится обязательно, если мы используем приложения SPA (Single Page Application) - приложения, которые не используют сервер-сайт рендеринг, например некоторые фреймворки (Angular и т.д.). Поскольку мы таки будем в нашем проекте использовать Angular (клиентская часть), да и возможно, в дальнейшем проект будет расширяться, API (Asp.net Core) нам вполне подойдет.

## 1. API

### 1.1 Swagger

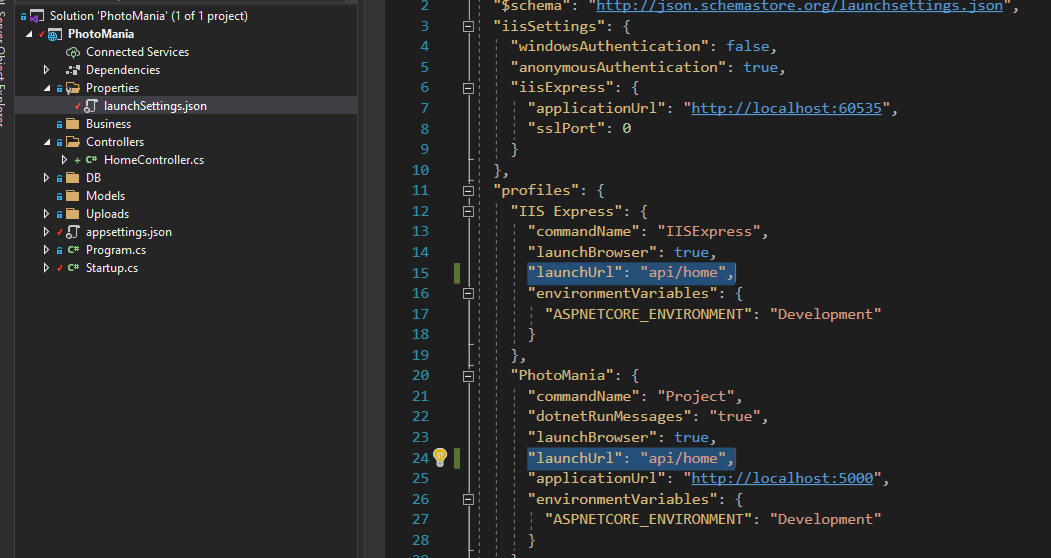
Первое с чего мы начнем это позаботимся о том, что бы у нас был Swagger и что бы он был по умолчанию, для чего при создании проекта выбираем последовательно: Asp.net Core Web Application, => выбираем версию Asp.net Core 5.0 и Asp.net Core Web API. Убираем галочку с https, но оставляем на "enable open api support". Со Swagger мы можем для нашего АПИ создать утилиту, которая позволит тестировать и писать описание к нашему АПИ (как постмен только лучше).

По ссылке, которую предоставляет Swagger (/swagger/v1/swagger.json) мы можем посмотреть документацию - полное описание наших запросов и в разделе "components" описан объект, который возвращается в результате запроса (это именно в swagger.json), а если мы подставим в ссылку "index.html" (у нас получится ссылка: "swagger/index.html" или полная ссылка http://localhost:57191/swagger/index.html) то мы увидим ту же документацию по нашим запросам, только в графическом интерфейсе. Здесь в 'Schemas' можно увидеть из каких полей состоит возвращаемое значение. Это нам заметно облегчит разработку.

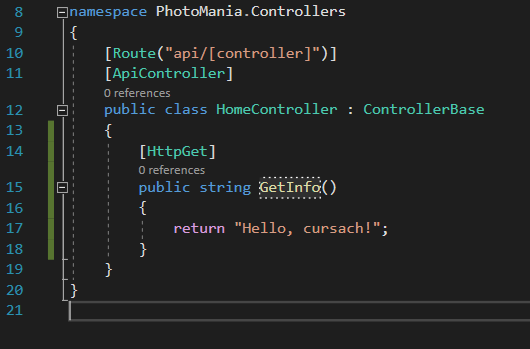
API автоматически преобразовывает ответ в json - это делает ASP.

Проверим как все работает:

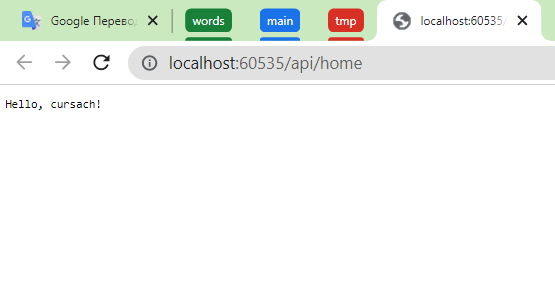
Для начала удалим в проекте класс WeatherForecast и контроллер по умолчанию, затем настроим, какой контроллер должен срабатывать при старте прогламмы в Properties/launchSettings.json в свойстве launchUrl:



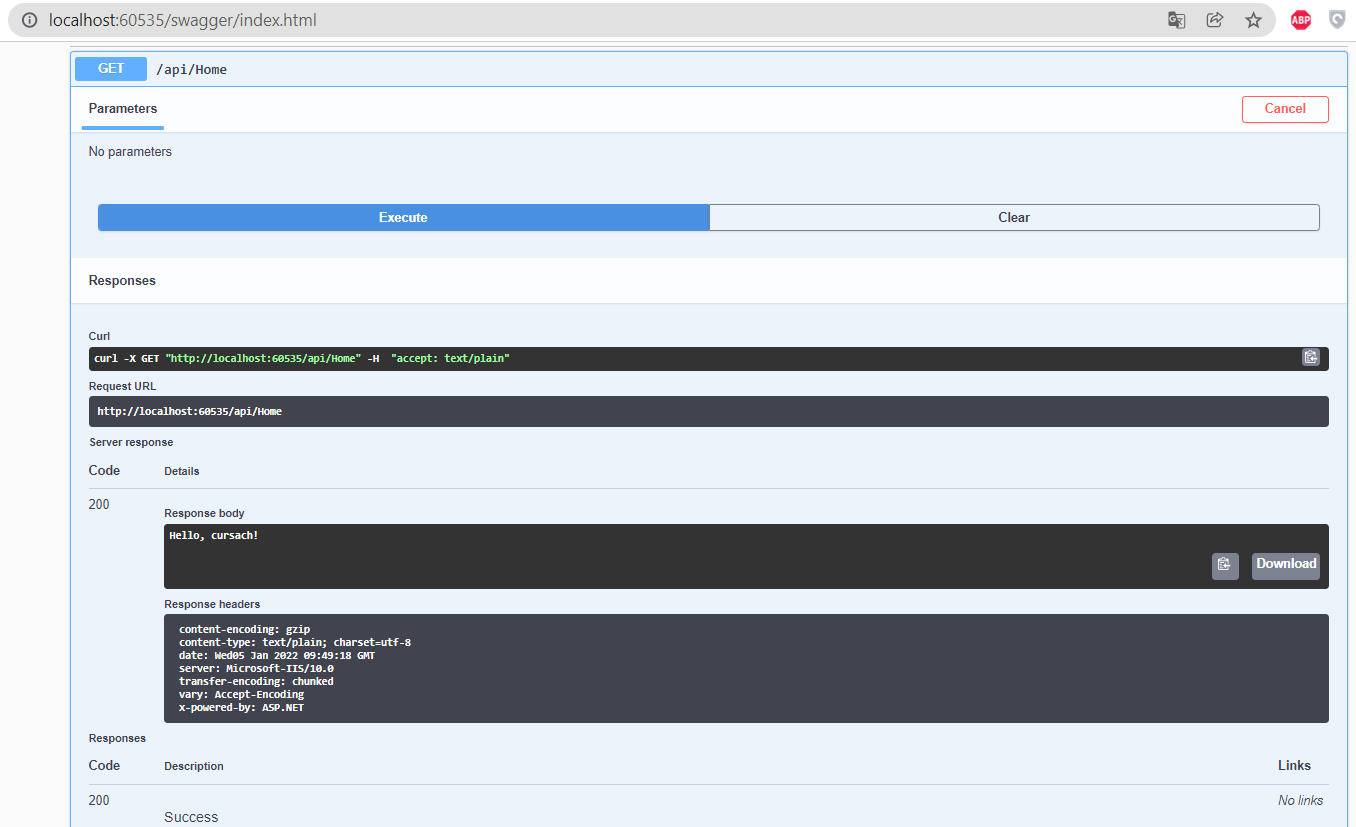
Далее создадим новый контроллер (когда мы создаем новый контроллер - выбираем "Контроллер API пустой"), в котором определим тестовый метод GetInfo()



В браузере:



Swagger:



### 1.2 Пакеты

Установим некоторые пакеты которые нам будут точно нужны на первых этапах работы:

1) для JWT авторизации устанавливаем пакет Microsoft.AspNetCore.Authentication.JwtBearer (v 5.0.13).

JWT - JSON Web Token https://ru.wikipedia.org/wiki/JSON\_Web\_Token

Используя даный вид авторизации защищаем API при помощи токена.

Токены предназначены для электронного удостоверения личности (например, клиента, получающего доступ к банковскому счёту), при этом они могут использоваться как вместо пароля, так и вместе с ним. В некотором смысле токен — это электронный ключ для доступа к чему-либо.

2) для работы с БД - Microsoft.EntityFrameworkCore.SqlServer (v 5.0.13)

3) для использования DbContext, запросов LINQ для выборки данных из БД и т.д. - Microsoft.EntityFrameworkCore (v 5.0.13)

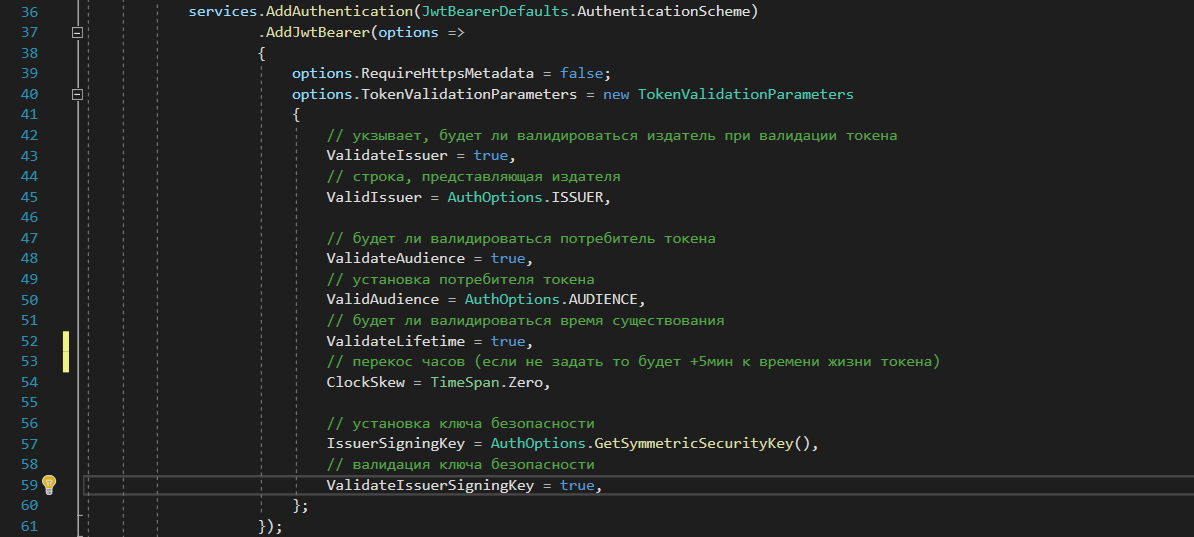
4) позволяет проецировать одну модель на другую – AutoMapper (v 10.1.1)

5) Для использования метода Post (например создания чего-то) нужно так же установить пакет Newtonsoft. Json (v 13.0.1) и Microsoft.AspNetCore.Mvc.NewtonsoftJson (v 3.1.17). Это поможет конвертировать типы данных автоматически, иначе пришлось для каждого типа данных создавать вручную отдельно конвертер (например у продукта цена типа данных десемал, но тип десемал это тип данных c#, а клиент у нас это js и он ничего не знает о типе данных десемал)

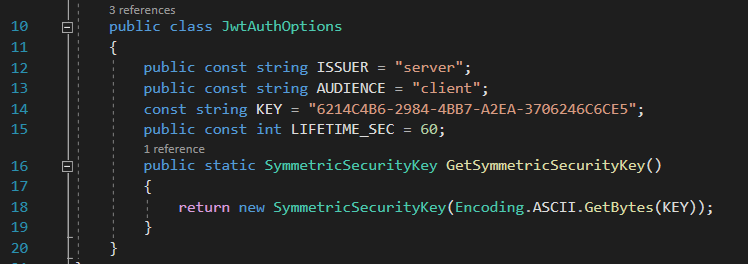
### 1.3 Startup

1. Метод ConfigureServices()

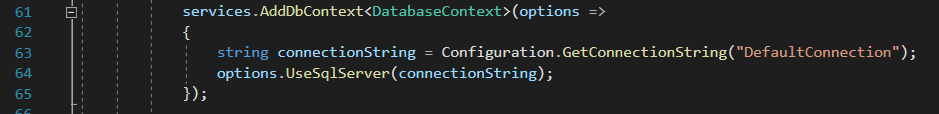
1) для JWT авторизации:



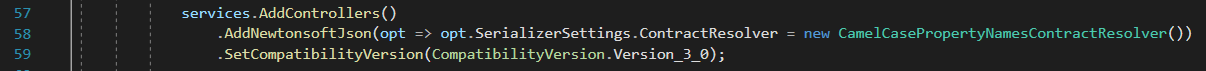
Так же создаем класс JwtAuthOptions, где указываем издателя токена, потребителя, ключ для шифрации и время жизни:



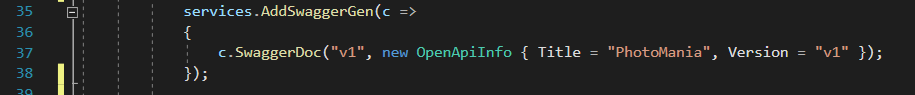
2) для работы с БД:



3) настраиваем конвертирование данных через Newtonsoft.Json:



4) настройки Swagger (были по умолчанию, после создания проекта)



5) настройка Сors

Сors - настройка доступа из других доменов к нашему ресурсу (АПИ). Добавляем возможность слать кросс-доменные запросы. по умолчанию наше приложение защищенно и мы не можем слать запрос на АПИ из другого домена (например сейчас у нашего АПИ домен такой: http://localhost:57191/ но если мы сделаем запрос с домена http://localhost:57222/ например, то браузер выдаст ошибку. а если мы хотим, например с нашей клиентской части в WebApplication2 сделать запрос на это апи (у клиентской части будет другой домен), то нужно разрешить кросс-доменные запросы таким образом):

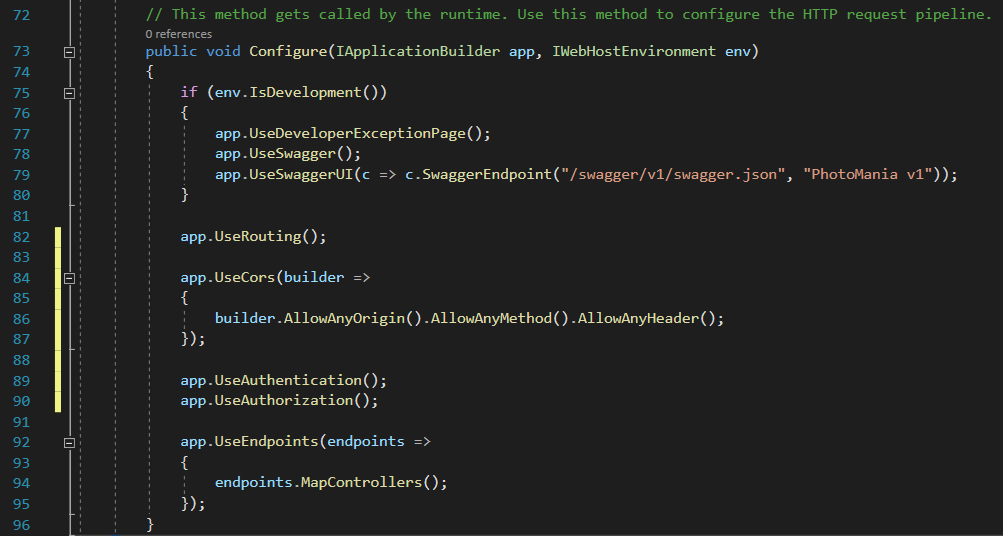


2. Метод Configure()

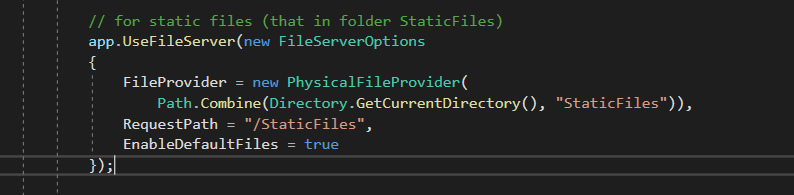
Чтобы задействовать CORS для обработки запроса в методе Configure вызывается метод UseCors(). помощью метода AllowAnyOrigin() мы указываем, что приложение может обрабатывать запросы от приложений по любым адресам. А так же:

AllowAnyHeader(): принимаются запросы с любыми заголовкам

AllowAnyMethod(): принимаются запросы любого типа (GET/POST)



Так же мы будем использовать статические файлы, прежде всего для фотографий пользователей



На нашем сервере будет общедоступная папка для аватаров и фотографий постов, к которым можно будет получить доступ на клиенте. Эта папка будет в корне проекта и назовем ее StaticFiles. Подробнее о том, как именно будем получать доступ к этой папке с клиента в разделе Client / Страница Home.

### 1.4 Основные дирректории

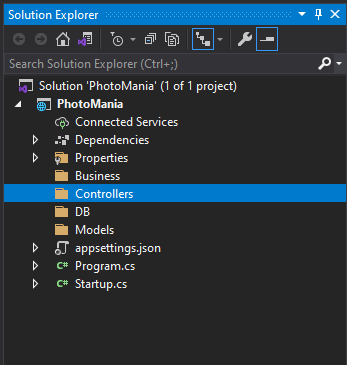
В апи, если нам нужно, можем добавлять отдельные слои по работе с БД, сервисы, потом просто получаем данные оттуда и возвращаем их клиенту, чем мы собственно и займемся.

3 основные папки:

1) DB – здесь будут храниться сущности, репозитории и контекст БД

2) Business – DTO объекты и сервисы которые будут переводить сущности в DTO объекты (и выполнять еще какие-то дополнительные действия, где понадобится). Объекты DTO - сущности, предназначенные для описания данных, которые мы будем возвращать из наших контроллеров - в них описываются данные, которые будет получать пользователь, который делает запрос

3) Models – дополнительные модели, которые мы будем отправлять на клиент (далее будет об этом подробнее)

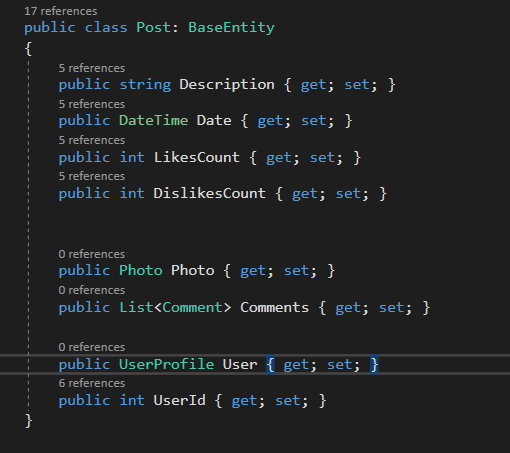


### 1.5 Дирректория DB

1) Создадим папку Entities в которую добавим основные сущности – BaseEntity, Post, Comment и т.д. От BaseEntity мы будем наследовать все остальные сущности.

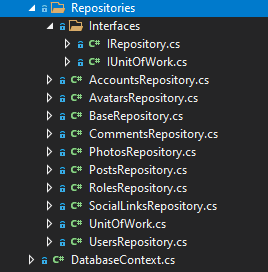


Настроим в каждой сужност нужные нам всязи, например так будет выглядеть сущность «Пост»:

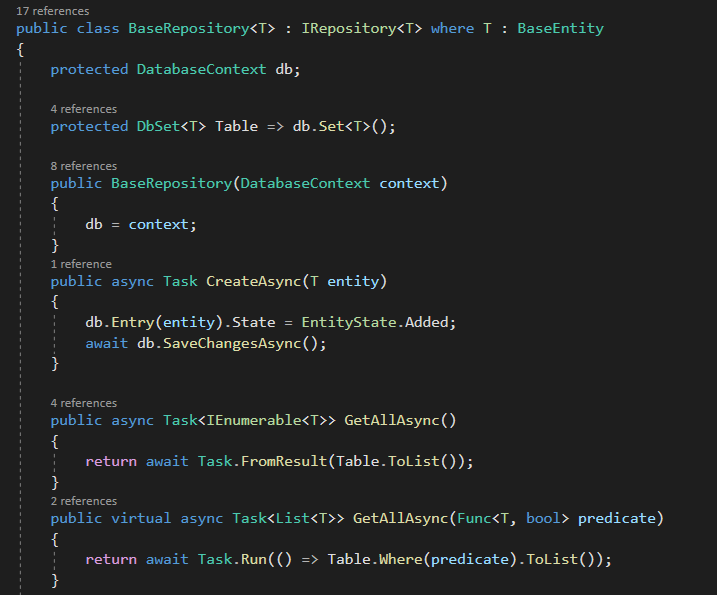


Здесь мы видим связь с таблицами photos, comments, userProfiles. И в дальнейшем добавятся еще какие-то связи, возможно.

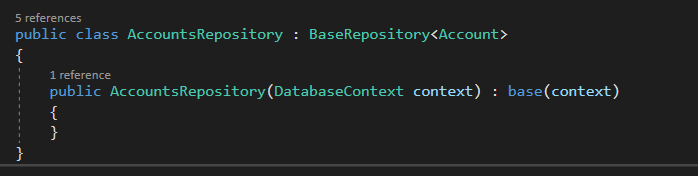
2) Далее создадим папку Repositories, в которой будут репозитории для каждой сущности, пременим пттерн UnitOfWork для удобной работы с этими репозиториями и создадим здесь DatabaseContext в котором определим таблицы по нашим сущностям



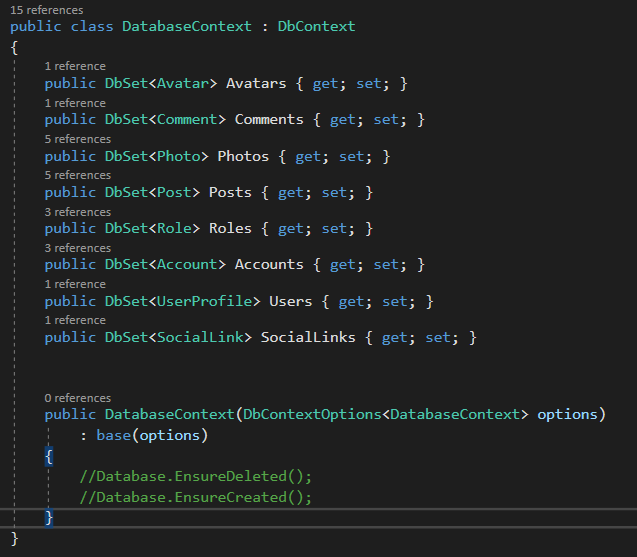
BaseRepository:



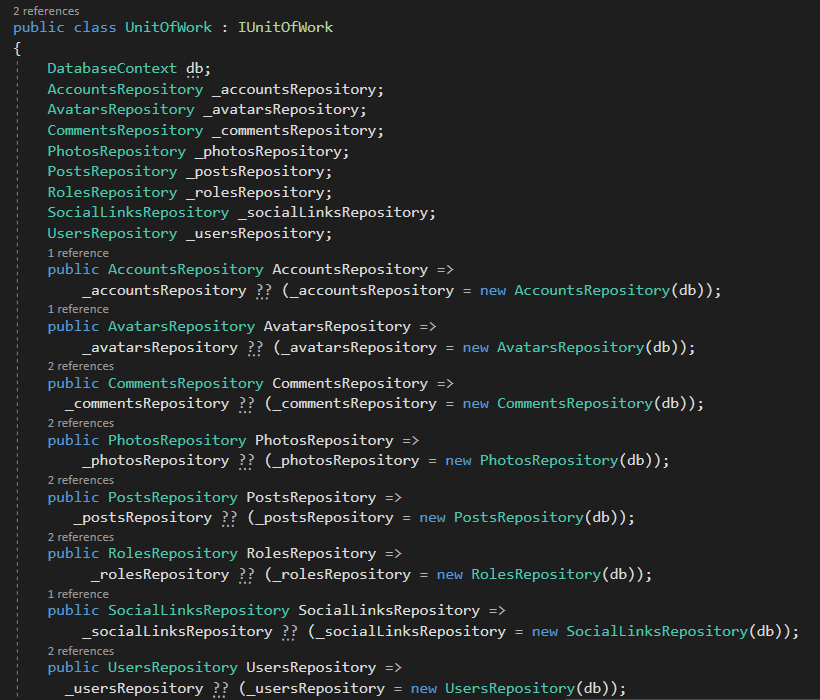
Один из репозиториев:



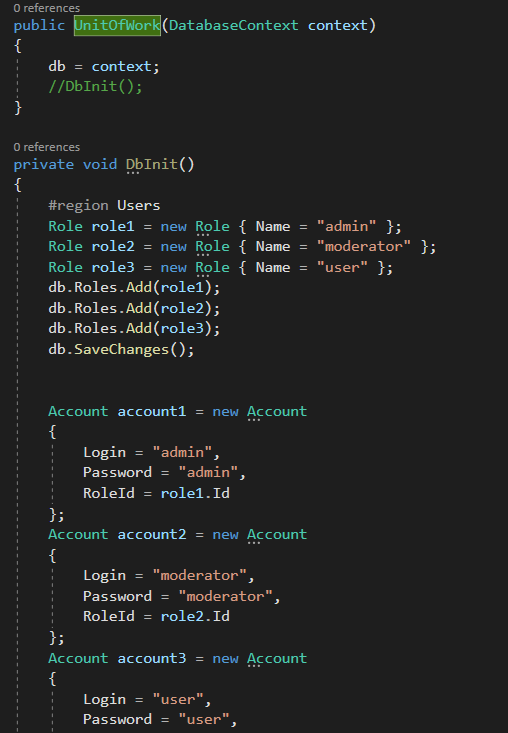
DatabaseContext:



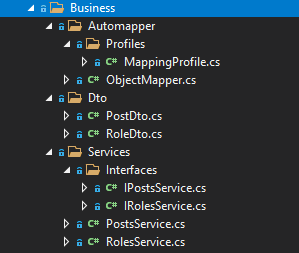
UnitOfWork:



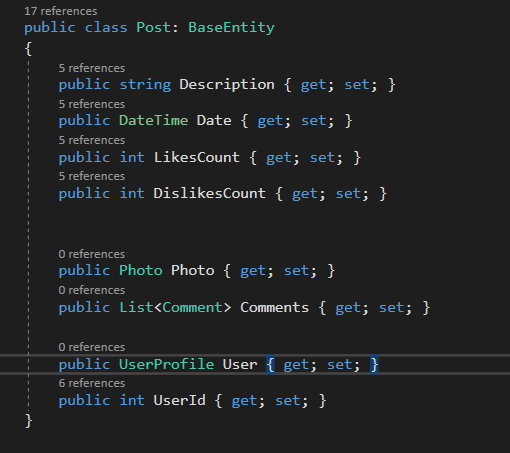
Здесь же, в UnitOfWork, будет инициализация базы данных значениями по умолчанию:



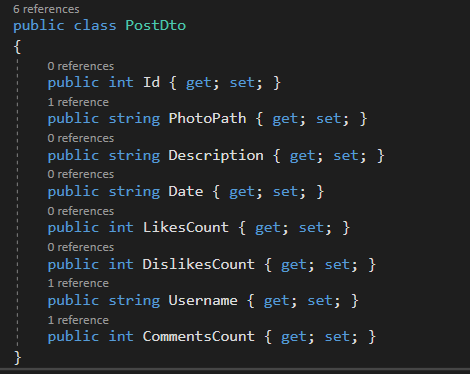
### 1.6 Дирректория Business



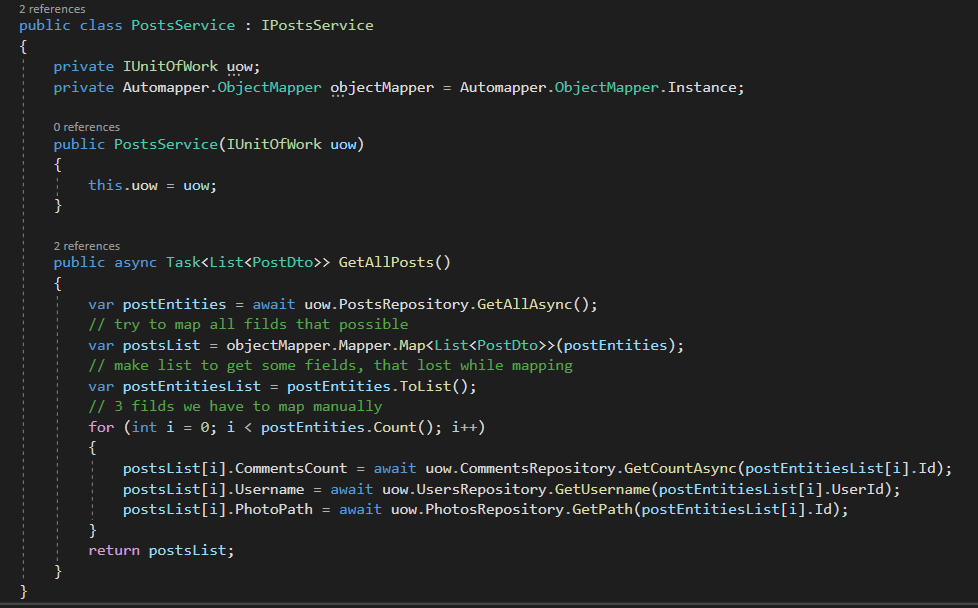
В папке Automapper мы будем маппить сущности в объекты DTO - сущности, предназначенные для описания данных, которые мы будем возвращать из наших контроллеров - в них описываются данные, которые будет получать пользователь, который делает запрос. То есть будем отдавать объекты в таком виде, в котором нам нужно отдать по запросу клиента. Например у нас есть сущность пост с такими полями:



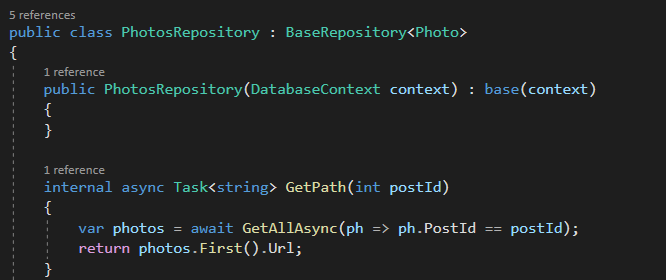
Но для отображения всех постов на главной странице, например, там нужны данные немного в другом виде, например нам нужно поле, которое содержит путь к фотографии поста (а тут мы видим только ссылку на объект фотографии) и т.д. Объект поста, которыйнам нужно отправить на клиент будет таким:



Для этого нам и нужны объекты DTO. Поля, который совпадают, за нас скопирует Automapper, а остальное мы можем получить за счет связей с другими таблицами. Рассмотрим как это делается. Для этого обратим внимание на папку Services – в которой будут сервисы, которые и помогут нам получать объекты которые нам нужны (DTO) из сущностей БД. Например в PostsService мы собственно и преобразуем нашу сущность таким образом:



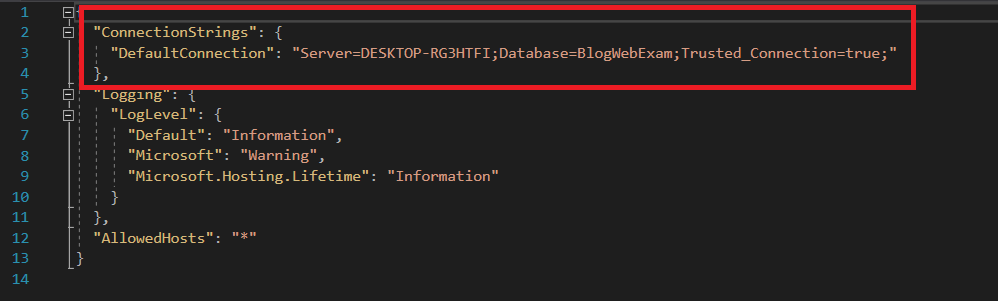
Что мы тут видим: мы наследуемся от интерфейса IpostsService, который описывает методы нашего сервиса, инжектим IunitOfWork с помощью которого сможем с нашего сервиса получить доступ к любому репозиторию. А нам это понадобится таки. Сначала получаем все сущности постов, потом с помощью Automapper копируем все совпадающие поля в каждый объект PostDto, а затем несовпадающие поля получаем вручную, проходясь циклом по всем обектам и для каждого преобразовываем информацию. Например путь к фото поста мы получим через метод GetPath() в PhotosRepository так:



Вот на примере мы и рассмотрели для чего нам нужны репозитории, сервисы, объекты DTO, Automapper и вобще вся эта структура.

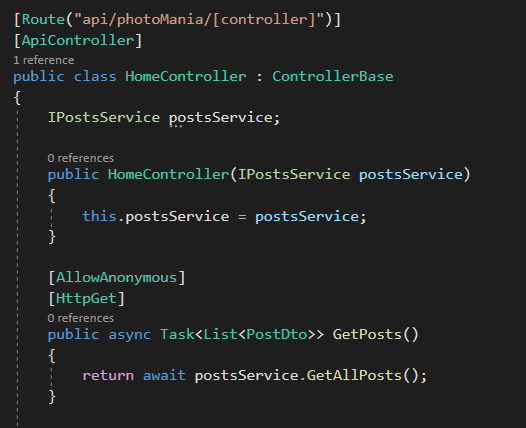
### 1.7 HomeController

Теперь для того что бы мы могли с клиента получить информацию о наших постах нам осталось сделать всего ничего:

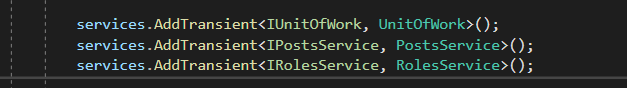
1) Пропишем для нашего АПИ ConnectionStrings в appsettings.json - это нужно для соединения с базой данный: 

Где BlogWebExam – название БД.

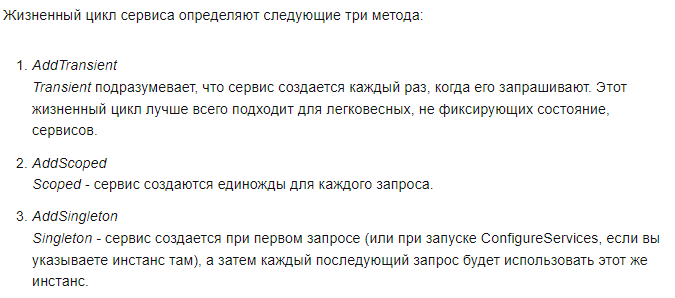
2) используем в HomeController все что мы рассмотривали все это время:



Здесь мы инжектим IpostsService что бы пользоваться в контроллере методами этого сервиса, кстати, инжектить мы можем потому что сделали так в Startup / ConfigureServices – зарегистрировали наши сервисы:



Что такое AddTransient и о жизненных циклах сервисов:

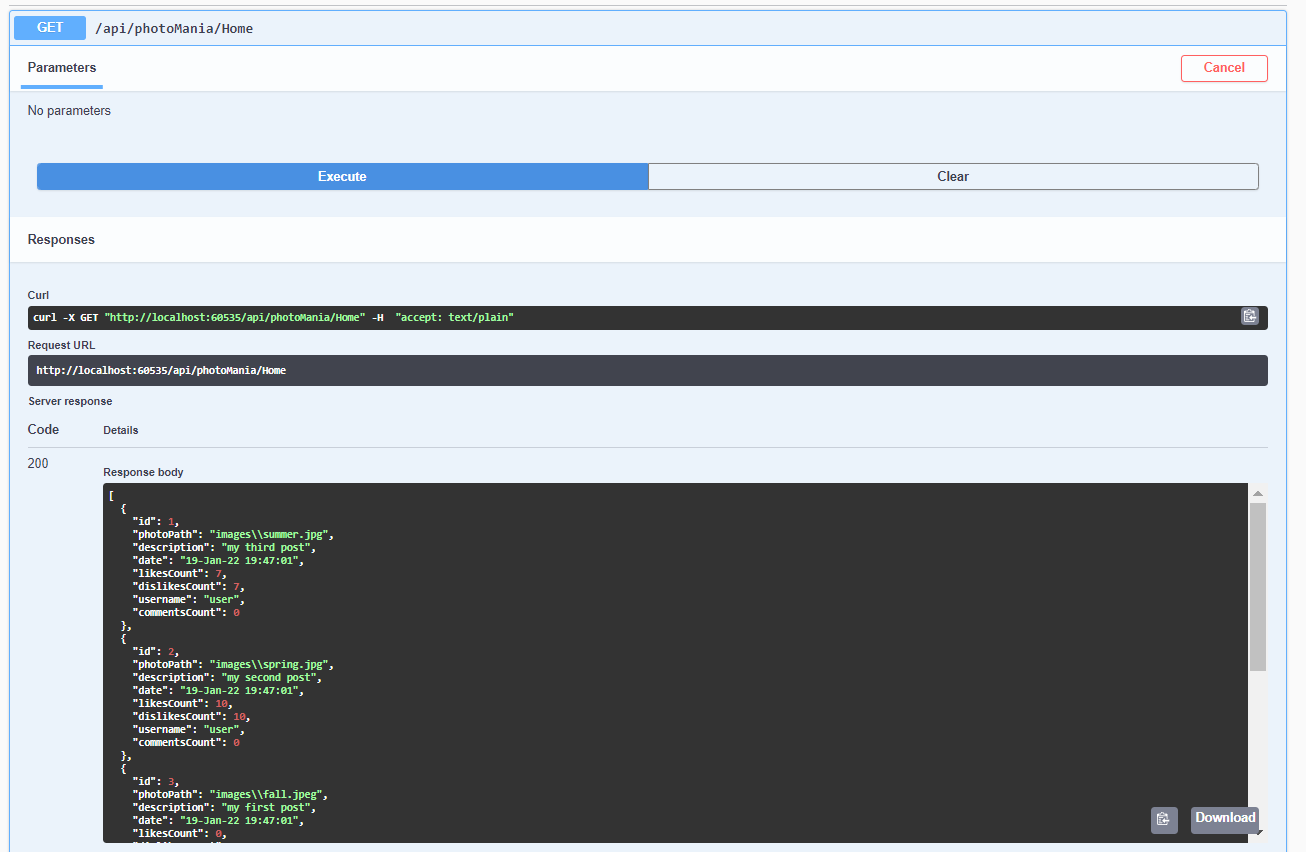


И вобще инжектить – значит применять механизм Dependency injection (DI).

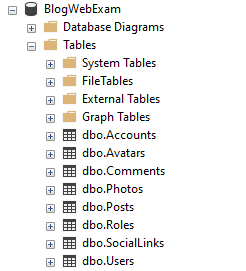
Dependency injection (DI) или внедрение зависимостей представляет механизм, который позволяет сделать взаимодействующие в приложении объекты слабосвязанными. Такие объекты связаны между собой через абстракции, например, через интерфейсы, что делает всю систему более гибкой, более адаптируемой и расширяемой.

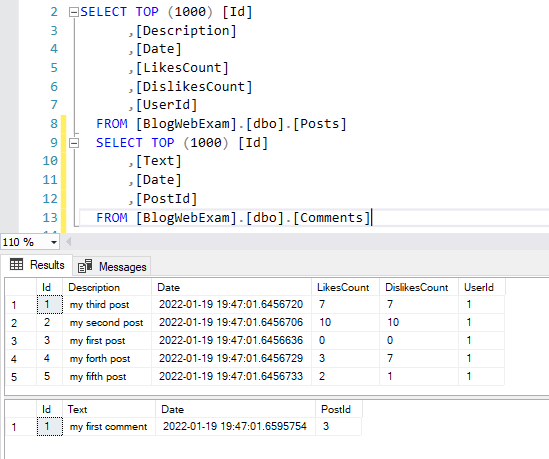
То есть если мы создадим сервис вместе с интерфейсом для него (как мы сделали с PostsService) и зарегистрируем это все в Startup, то сможем в любом конструкторе получить этот сервис и методы по работе с ним, как мы сделали в HomeController.

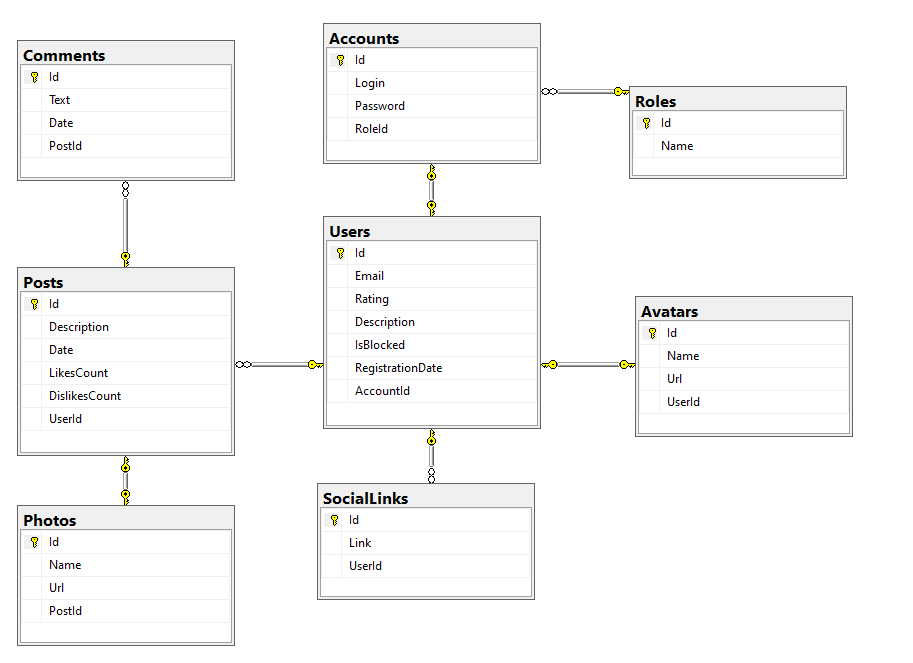
Теперь при запуске нашего приложения создастся база данных, заполнится нашими дефолтными данными (БД создается при первом обращении к ней а мы сделали это в HomeController, а он запускается автоматически потому что мы это прописали в launchSettings.json а над контроллером мы прописали атрибут [Route("api/photoMania/[controller]")] – с таким же роутом как и там) и теперь мы можем получить все наши посты с теми полями, которые нам нужны – проверим в Swagger:



И мы можем убедиться что наша БД создалась c Microsoft SQL Server Management Studio







Далее, возможно, мы будем вносить по ходу работы какие-то изменения в структуру БД, поскольку в этой работе нет каких-то строгих рамок, а значит работа приобретает более творческий характер и может захочется добавить что-то или переделать, но это уже другая история.

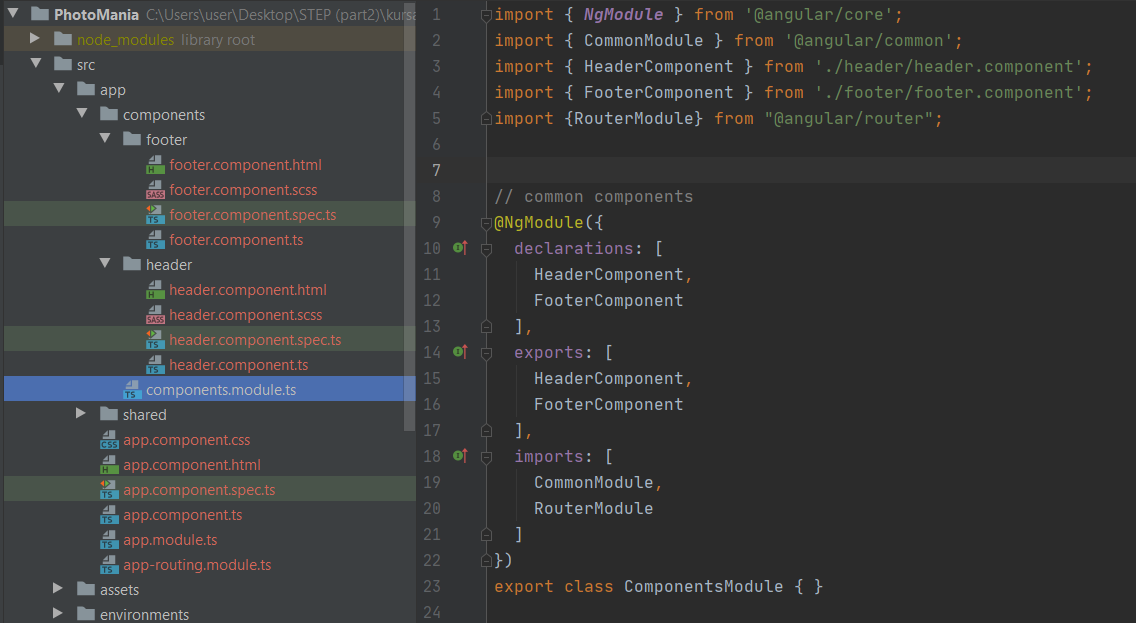
## 2. Client

Теперь все что мы сделали в апи получим на клиенте и покажем в браузере результат

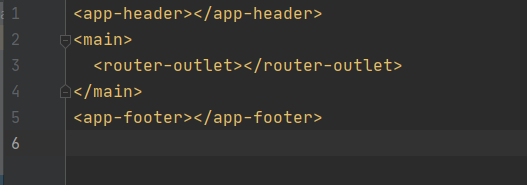
### 1.1 Создаем Angular проект и делаем первые шаги

1. при создании проекта применяем --routing (ng new project\_name --routing)

2. создаем папку components в которой определяем основные компоненты типа хедер и футер и модуль для них. создаем компоненты при помощи Angular Schematic. В этой папке так же будут и другие общие компоненты (которые используются в нескольких местах)

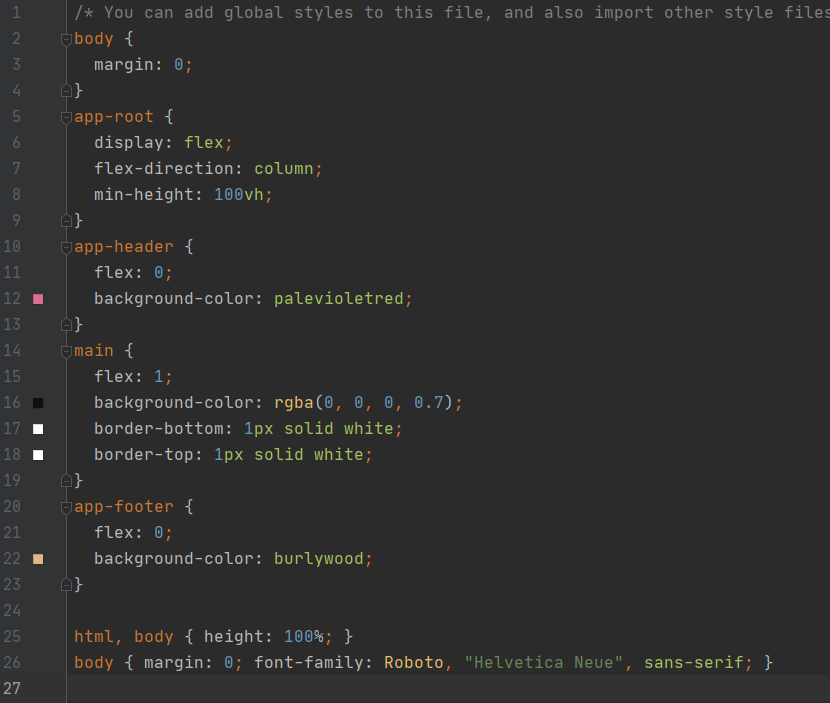


Подключим наш components.module.ts в app.module.ts, удалим содержимое файла app/app.component.html и вставим туда только что созданные хедер и футер

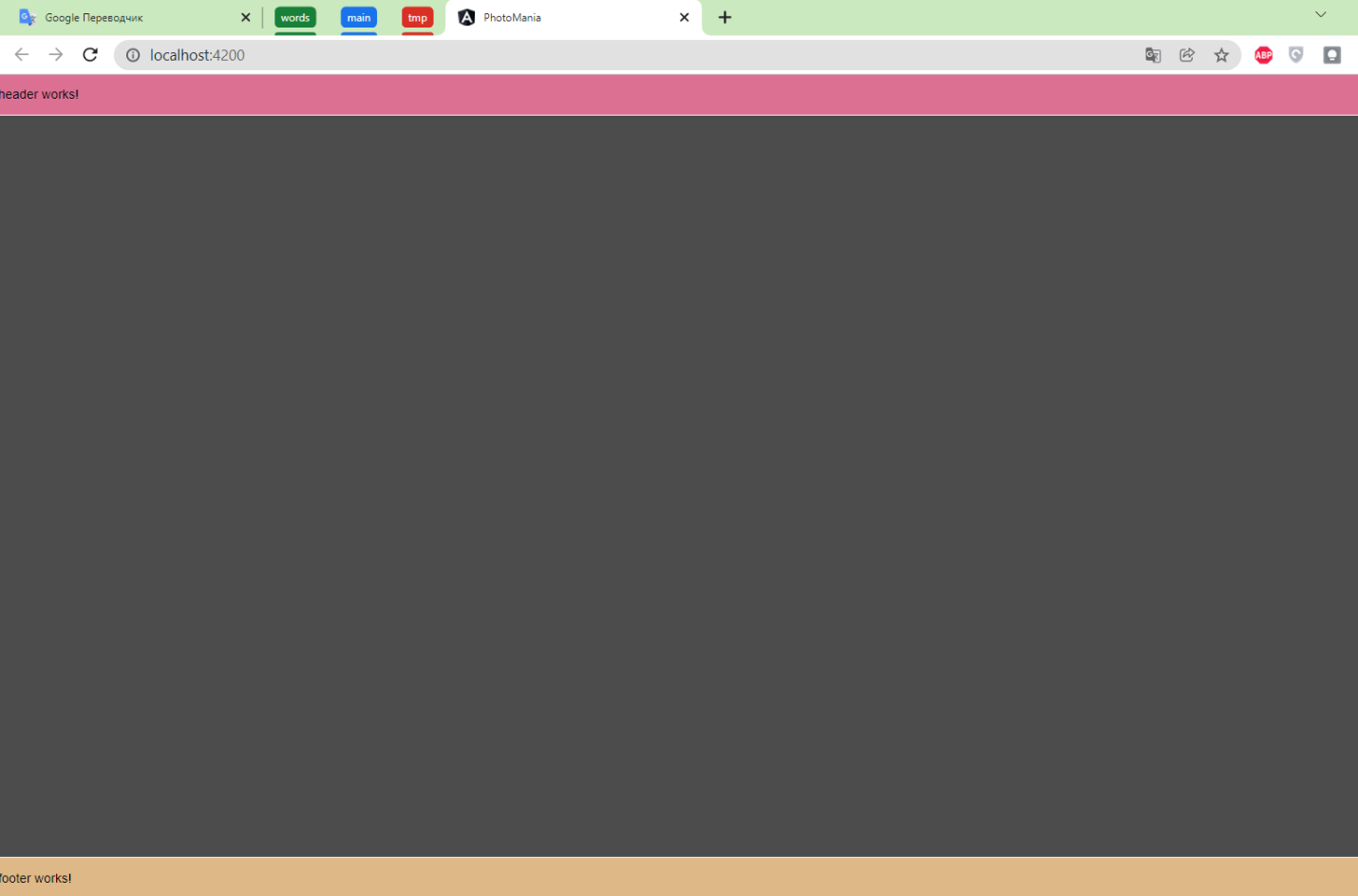


Таким образом хедер и футер изменяться не будет при переходе на другие страници, а страници, на которые будет переходить пользователь мы будем подставлять в тег main.

Немного глобальных стилей в src/styles.css



И у нас уже есть на что посмотреть в браузуре!



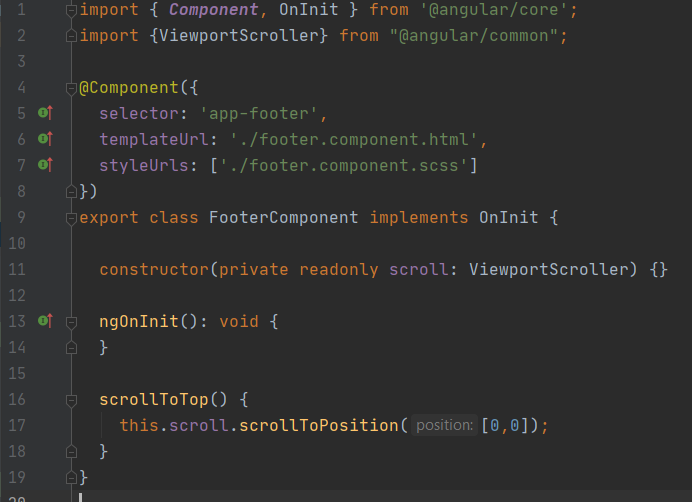
### 1.2 Установка Вootstrap, стилизация header/footer

Пишем в терминале npm i bootstrap, заходим в файл angular.json - там есть статические файлы (стили, скрипты) и подключаем туда bootstrap (к стилям и скриптам соответственно):

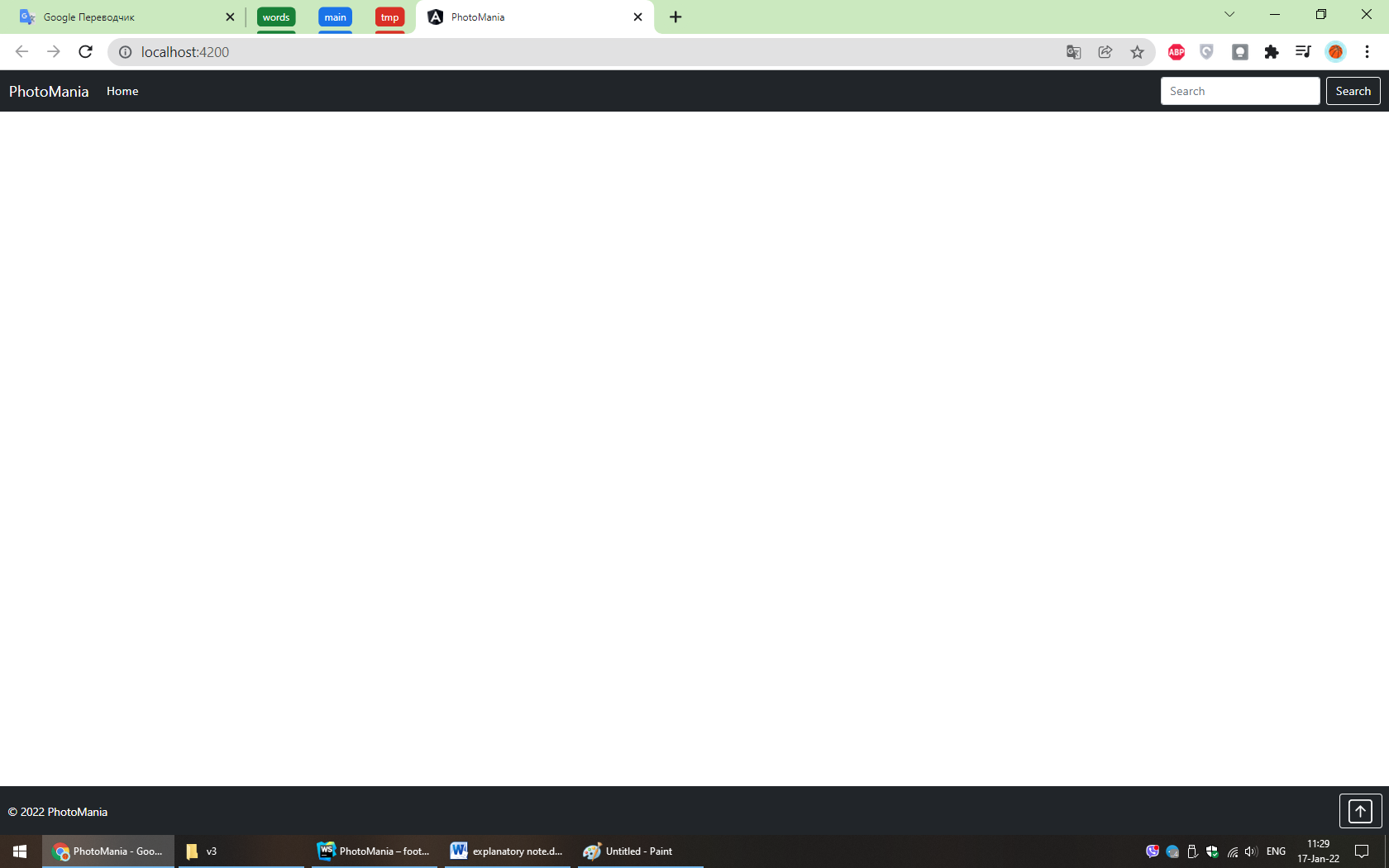


После чего перезапустим проект (ng serve). Возможно придется закрыть и открыть проект заново.

Стилизируем header и footer с помощью bootstrap (для header берем navbar у bootstrap). Логику работы кнопки «вверх» опишем в footer.component.ts



Получим header и footer в том виде, в котором он нам нужен:



### 1.3 Создание страниц

создаем папку pages в папке app, в ней будут страницы, каждая со своим маршрутом. например создание страницы home:

1) нажимаем ПКМ по папке pages и Open in terminal

2) пишем в терминале: ng g m home --routing (генерировать модуль home с роутингом)

3) там же в терминале по тому же адресу пишем: ng g c home (сгенерировать компонент

home - название компонента и модуля должны совпадать что бы создалось все в одной папке)

4) в файле home-routing.module.ts прописываем роут для этой страници:

const routes: Routes = [

{

path: '',

component: HomeComponent

}

];

5) в файле app-routing.module.ts регистрируем этом маршрут с ленивой загрузкой:

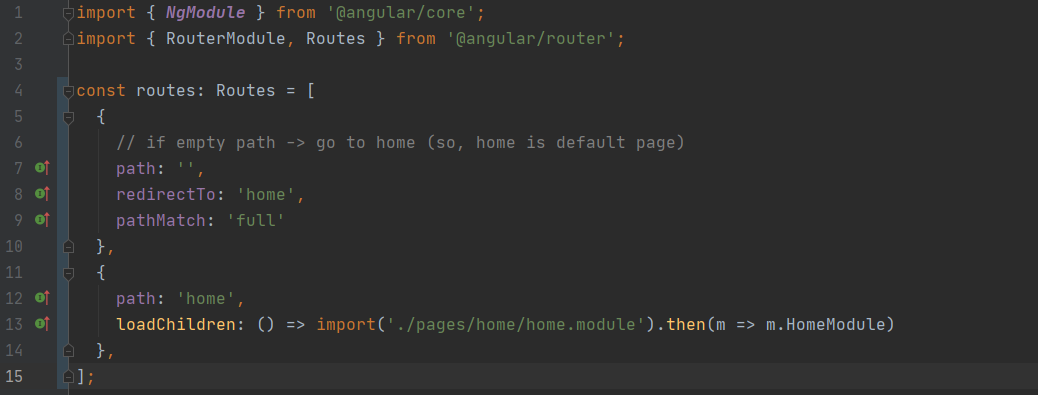
{

path: 'home',

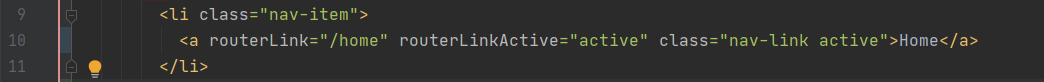
loadChildren: () => import('./pages/home/home.module').then(m => m.HomeModule)

}

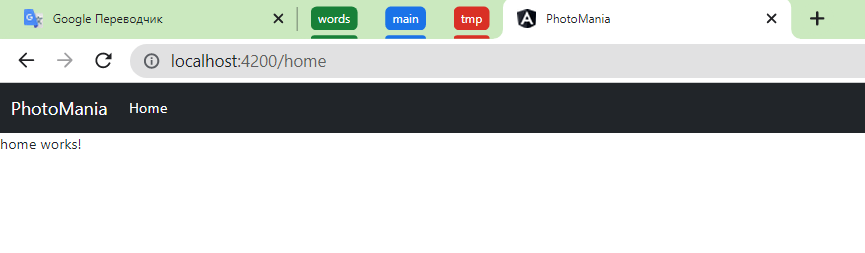
Так же сделаем переадресацию на эту страницу, если путь будет пустой. Важно что бы эта сапись была самой первой. В итоге получим в app-routing.module.ts таким:



В header подвязываем эту страницу к ссылке Home:



Теперь при загрузке сайта мы попадем сразу на главную страницу и при переходе по ссылке на главную страницу мы так же попадем на главную страницу:

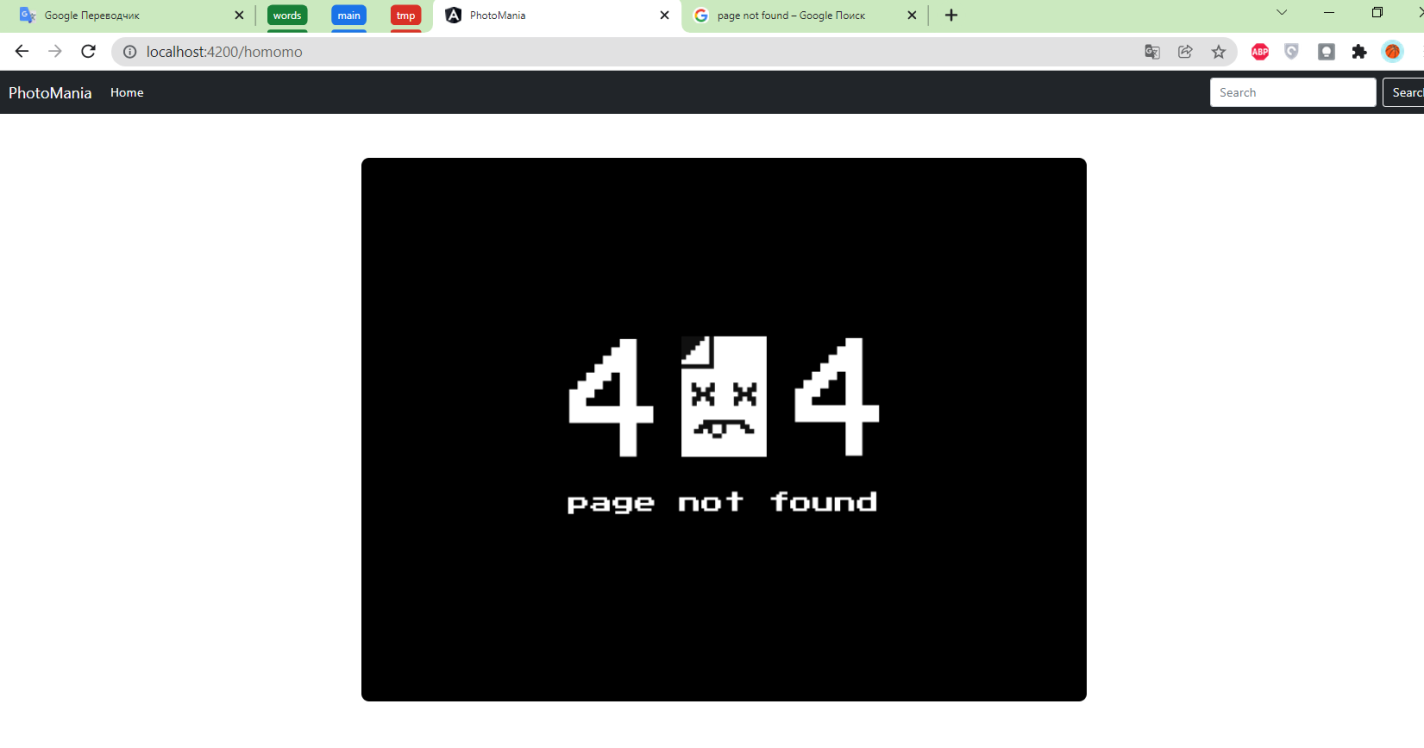


При этом, заметьте что header и footer не будут меняться при переходах по ссылкам, а меняется только середина – наши страницы, поскольку все они будут подставляться в тег <router-outlet></router-outlet>. Последующие страницы (разделы сайта) будут создаваться по такому же принципу как и Home.

И сразу создадим страницу not-found что бы при ошибке в маршруте мы видели что такой страници нет. После создания компонента и модуля в файле app-routing.module.ts регистрируем этом маршрут самым последним (обязательно) таким образом:

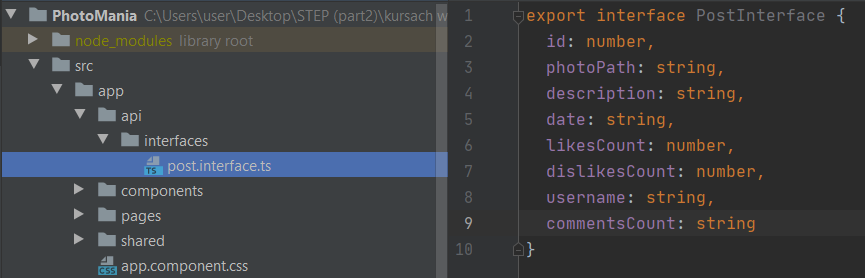


Теперь если мы напишим какую-то ерунду в адресной строки то увидим:

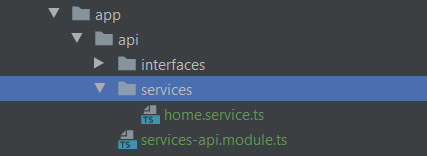


### 1.4 Работа с api

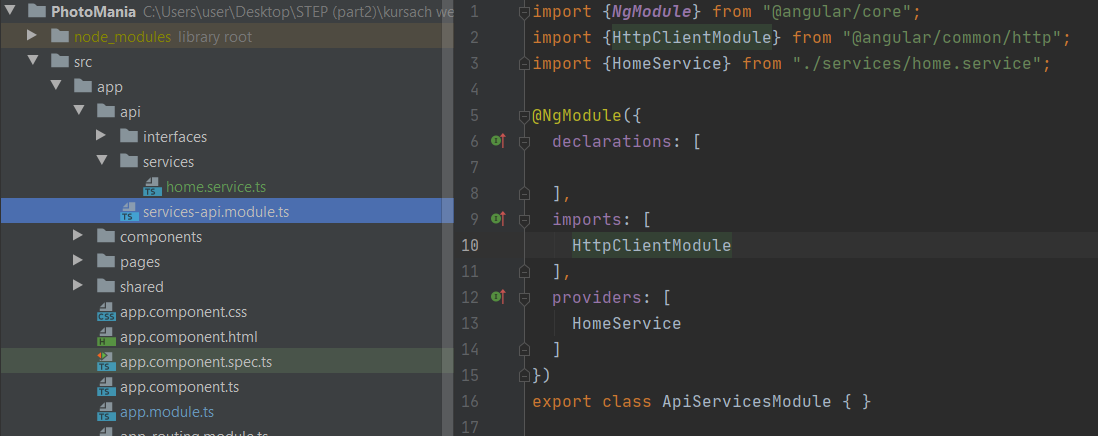
Создадим папку api в app и в папке api создадим папку interfaces - в которой будут описаны сущности, получаемые с апи:



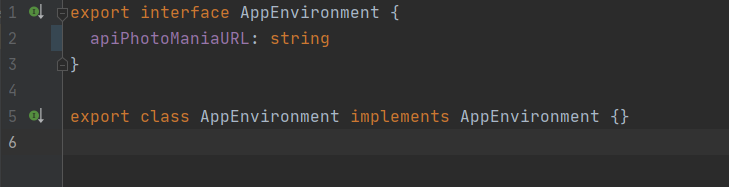
Так же создадим папку services в которой будут сервисы по работе с апи (каждый сервис для отдельной страници в приложении) и файл services-api.module.ts в котором мы будем регистрировать эти сервисы:



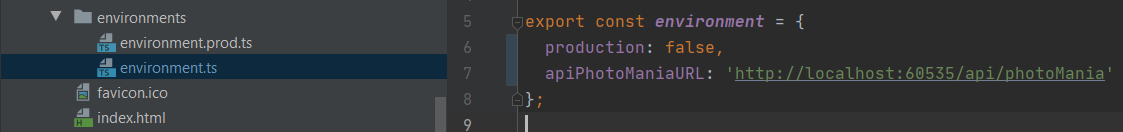
Зарегистрируем в модуле наш первый сервис и импортируем HttpClientModule для работы с Http, а так же подключим над модуль и HttpClientModule в app.module.ts (если выдает ошибку что HttpClientModule не найден то нужно будет перезапустить проект)



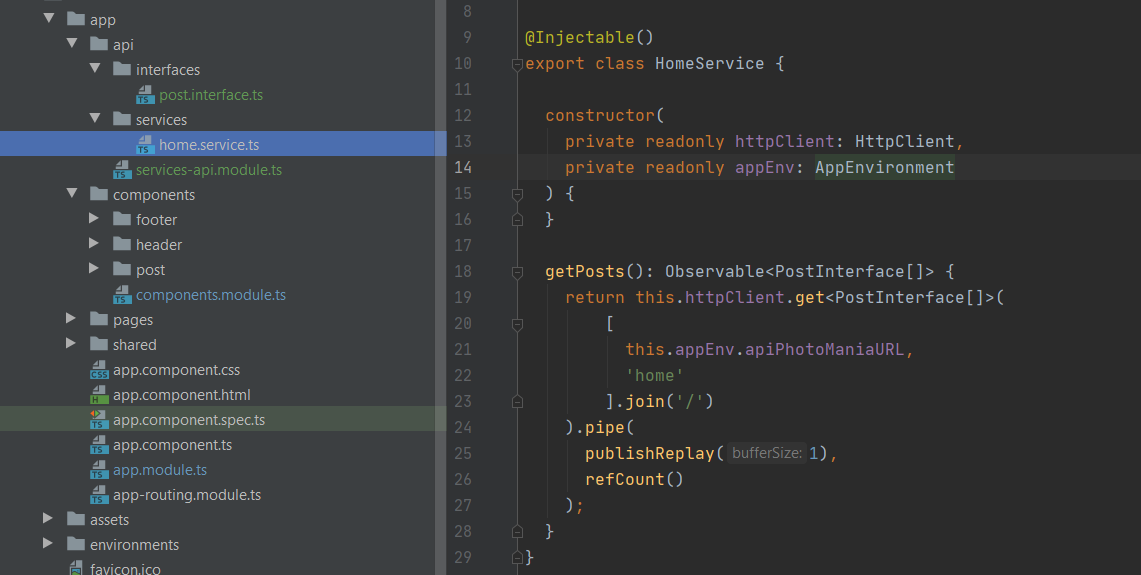
Далее займемся сервисом для главной страници – HomeService:

В папке shared создаем app-environment.interface.ts для удобного доступа к environment 

А в самом environment установим ссылку на апи:

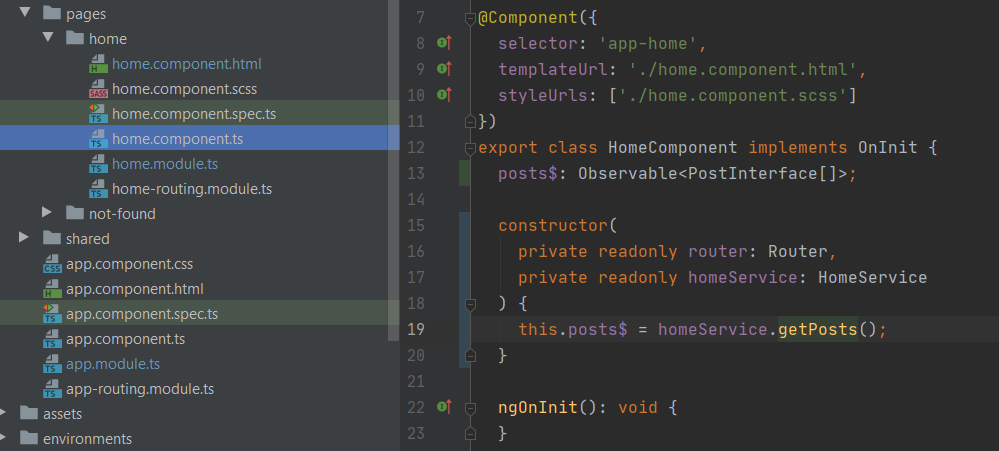


Теперь можем писать гет запрос, который позволит нам получить все посты с апи:

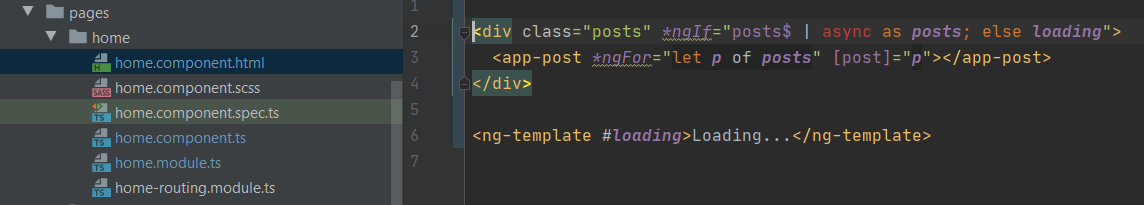


### 1.5 Страница Home

Теперь из сервиса мы будем получать посты и отображать их на главной странице в определенном виде. Получаем посты:



Отображаем:

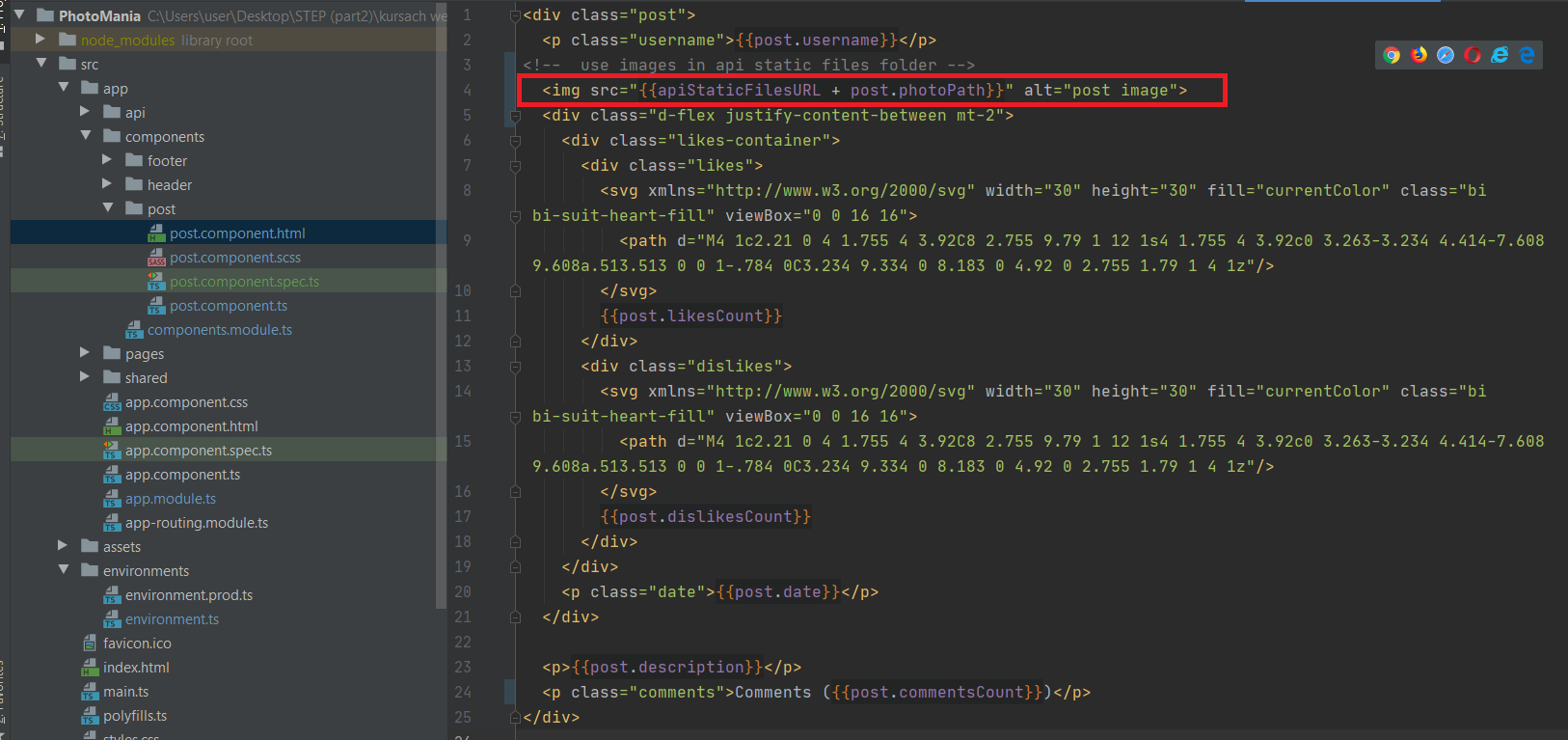


Таким образом пока будут грузиться посты – будем отображать «Loading…»

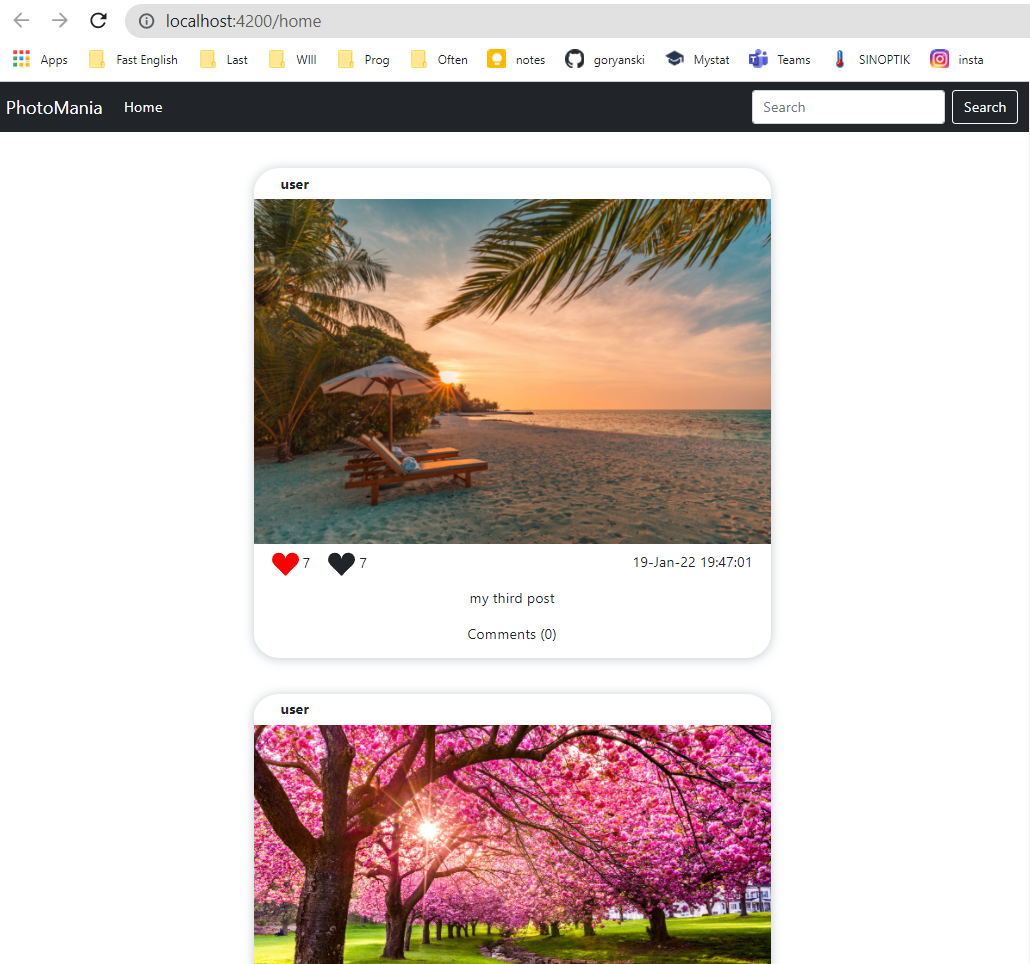
Каждый пост сделаем отдельным компонентном в папке components, поскольку мы может будем его использовать еще где-то, например, при поиске постов. ngFor будет перебирать все посты и в post.component.ts опишем интерфейс получаемогого объекта что бы удобно использовать его в шаблоне, так же получим ссылку на статические файлы сервера (для вывода фотографий на клиенте)



И того путь к фотографии поста будет apiStaticFilesURL (<http://localhost:60535/StaticFiles/> - описано в environment.ts) + post.photoPath (например images/fall.jpeg). Применим это в шаблоне поста:



Немного стилизуем это все дело и получим нашу ленту постов:



## 3. А теперь все вместе

В предыдущих разделах мы продемонстрировали структуру апи и клиентской части, которую мы будем использовать, рассмотрели основные моменты общее направление дальнейшей работы. Далее будем одновременно разрабатывать и апи и клиентскую часть по каждой задаче, которую предстоит выполнить, для того что бы пояснение работы было более логичным и последовательным

### 1.1 Пагинация

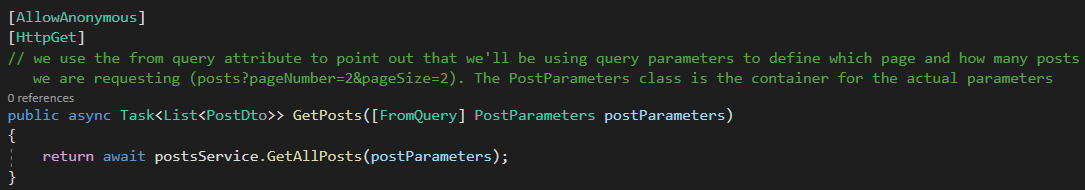
Сейчас мы получаем на главную страницу все посты из БД от всех пользователей. Но ведь в реальном приложении этих постов может быть очень много (тысячи и миллионы), к тому же они будут постоянно добавляться и если мы попробуем получить сразу все, ничего хорошего от этой затеи ожидать не стоит. А мы таки хотим что бы на главной странице – в «новостной ленте» отображались последние посты всех пользователей приложения. Для реализации этой задачи мы сделаем пагинацию.



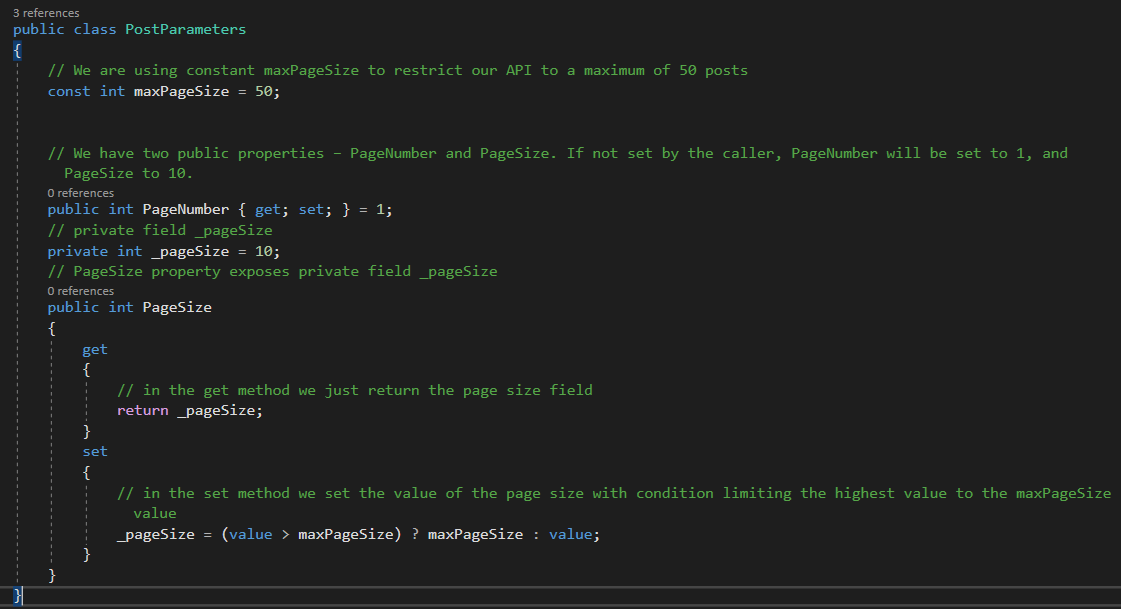
Пагинация используется на разных типах сайтов, а в особенности на больших интернет-магазинах. Благодаря ей огромные пласты информации можно разделить на части, за счёт этого увеличивается скорость загрузки страниц, а также становится удобно просматривать и находить нужные элементы, и это хорошо повлияет на поведенческие факторы и поисковое продвижение ну и как минимум наш сервер не ляжет.

#### Api

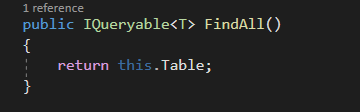
И так, начнем со стороны апи – с контроллера HomeController:



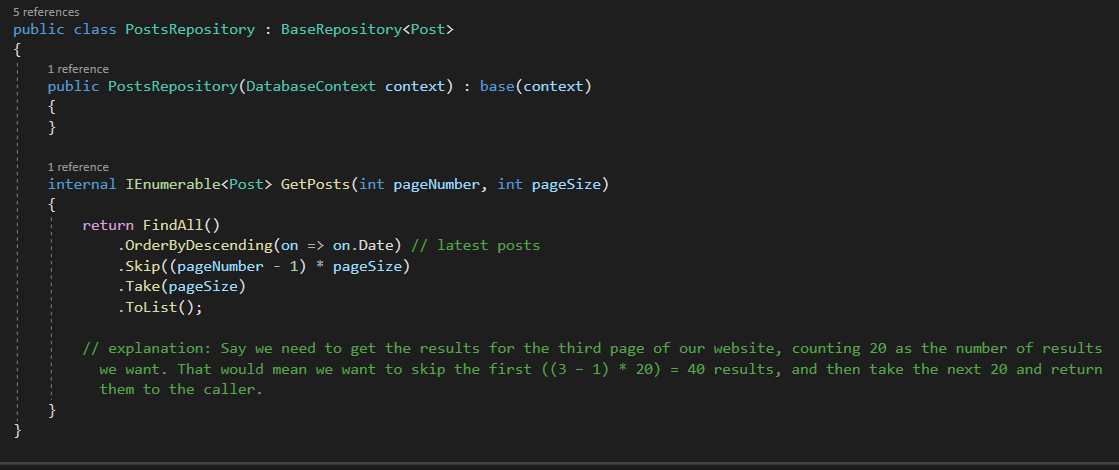
То есть теперь мы не будем получать просто все посты с БД а будем передавать параметры – какие именно посты мы хотим получать (posts?pageNumber=2&pageSize=2 – какую страницу и как много постов мы хотим получить с БД). Параметры оформим отдельным классом в папке Business / PaginationModels / PostParameters. Именно там, потому что работать с этим классом мы будем в PostsService, который так же в разделе Business. PostParameters будет выгдядеть так:



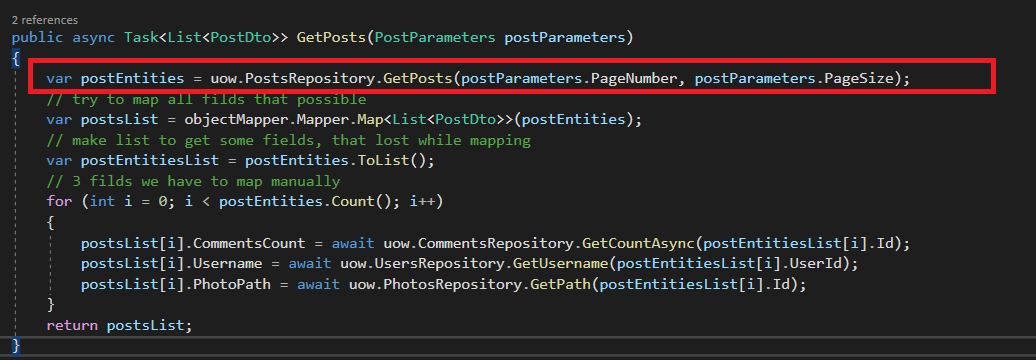
Добавим в BaseRepository метод, который будет возвращать все записи как IQueryable



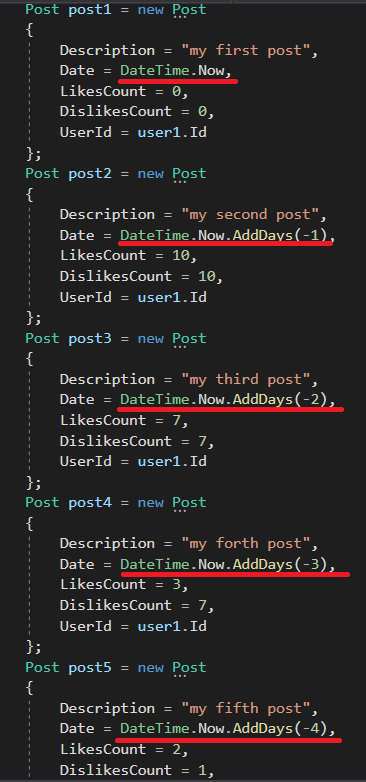
Где Table это: DbSet<T> Table => db.Set<T>();

Теперь перепишем метод получения постов в PostsRepository:

И в PostsService так же изменим чуть метод получения постов:

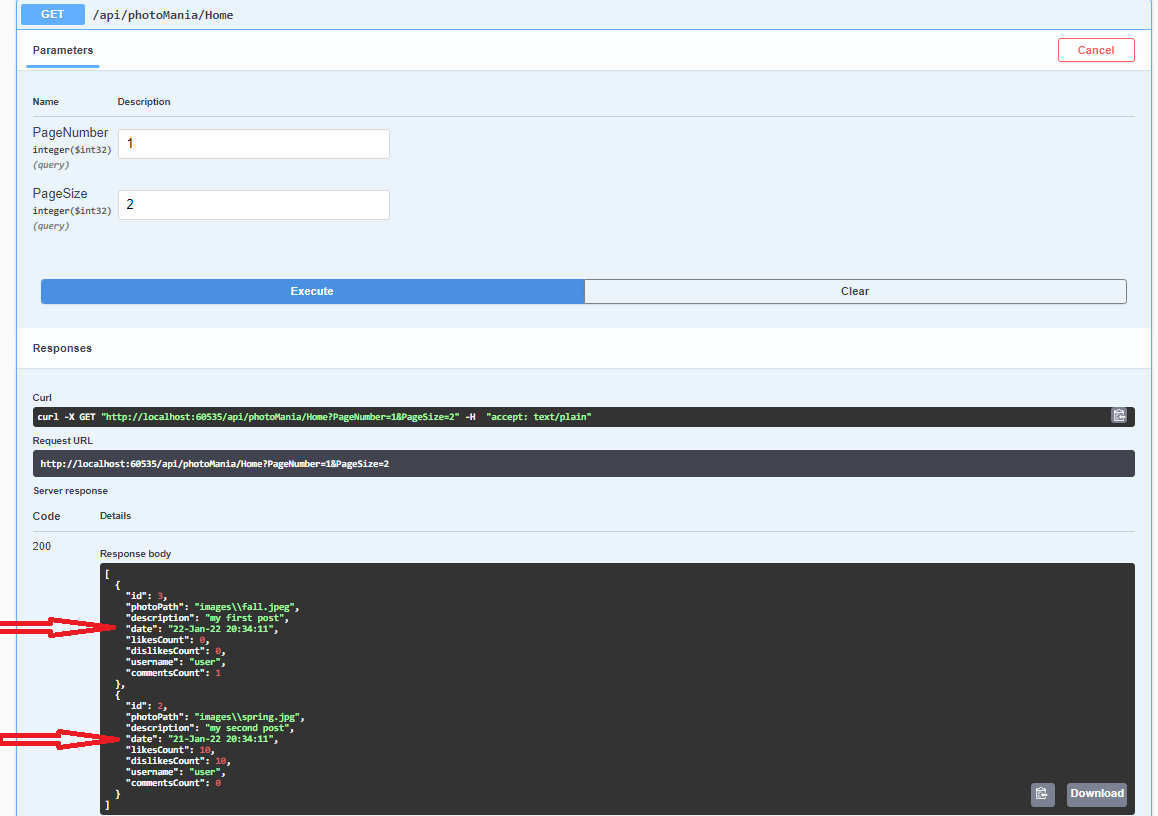


Теперь для того, что бы легко заметить работает ли все корректно, мы немного изменим данные в БД:

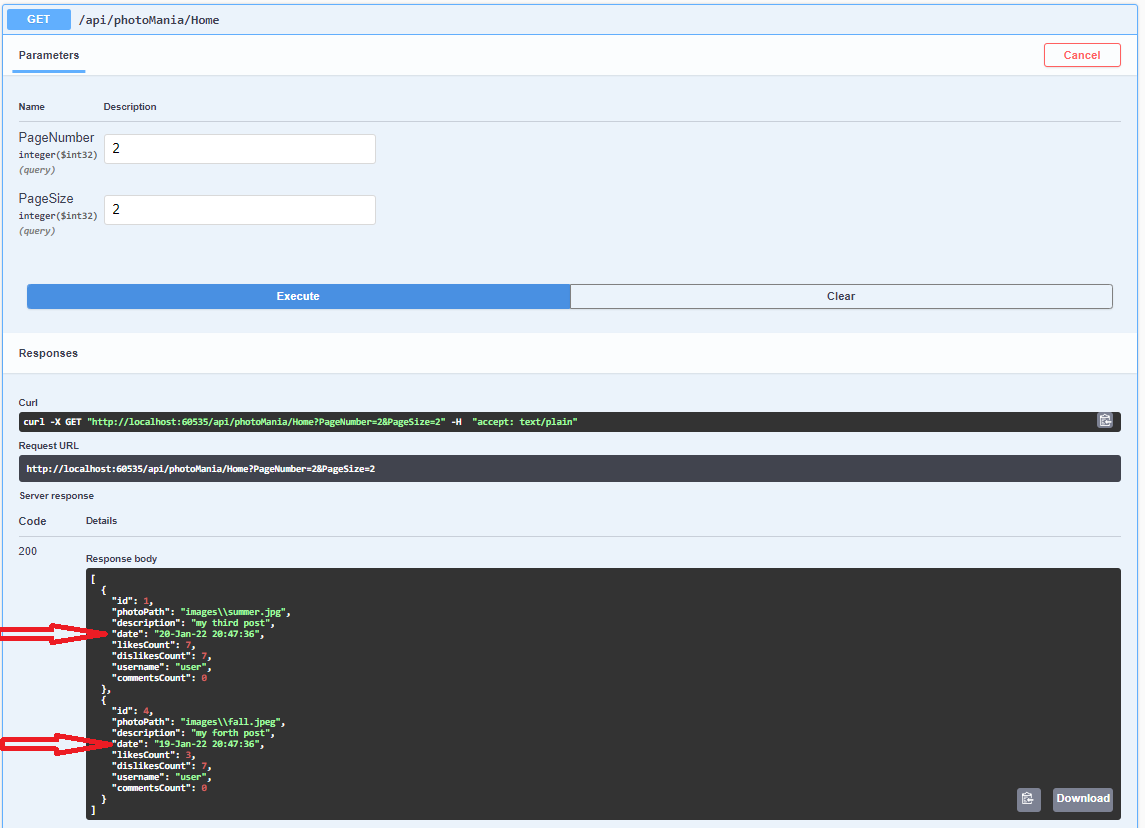


Поскольку мы сортируем посты по дате то нам важно удостовериться, что все выводится правильно, проследив за датами постов.

Берем 2 поста с первой страницы:



Видим что первый пост – дата за сегодня, второй – за вчера. Выведем теперь вторую страницу:



Если на первой странице было 22 и 21 число, то на второй мы видим 20 и 19, что означет правильную работу пагинации.

#### Client

Теперь займемся отображением постов с пагинацией на главной странице нашего сайта. Важно отметить что у нас будет не классическая пагинация, как в интернет-магазинах, потому что это не логично делать в приложениях такого рода, нам не обязательно знать сколько всего постов и нам не нужно возвращаться на определенную страницу, как это нужно делать в интернет-магазине что бы сравнить цены, например. И наша пагинация будет больше напоминать ленту в инстаграме – в процессе прокрутки страници мы будем подгружать сдедующие страницы (следующую группу постов),

// только сделаем не по прокрутке, а по нажатию на кнопку «Показать еще» для того что бы это было более нагладно. Это ведь тестовое задание, а в реальном рабочем проекте конечно было бы лучше незаметно для пользователя подгружать информацию, но если нужно будет – это очень легко переделать. //