# 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

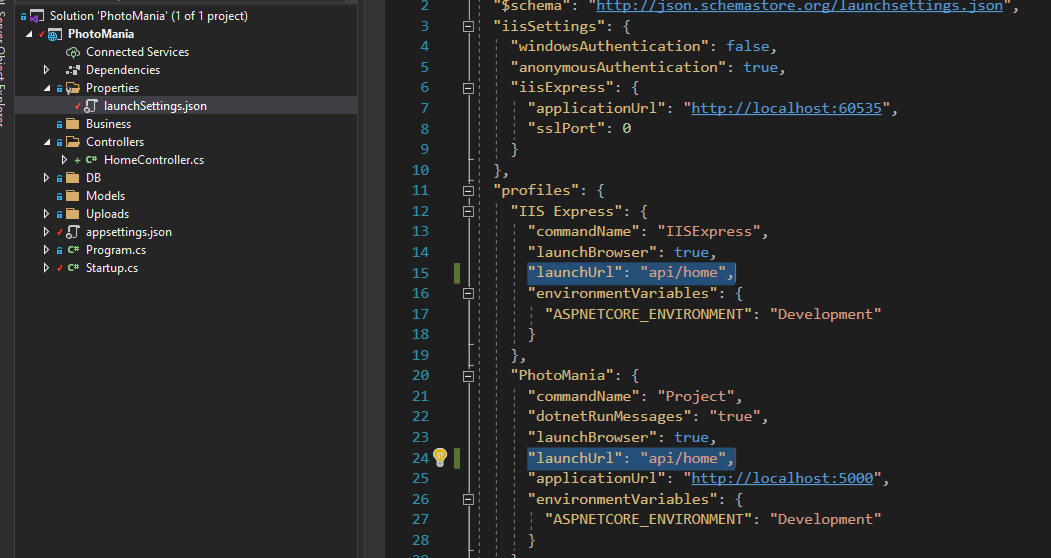
Первое с чего мы начнем это позаботимся о том, что бы у нас был Swagger и что бы он был по умолчанию, для чего при создании проекта выбираем последовательно: Asp.net Core Web Application, => выбираем версию Asp.net Core 5.0 и Asp.net Core Web API. Убираем галочку с https, но оставляем на "enable open 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.

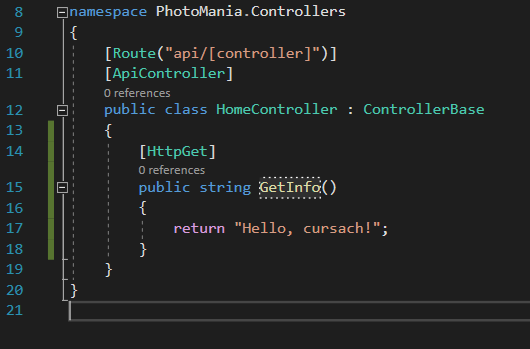
API автоматически преобразовывает ответ в json - это делает ASP.

Проверим как все работает:

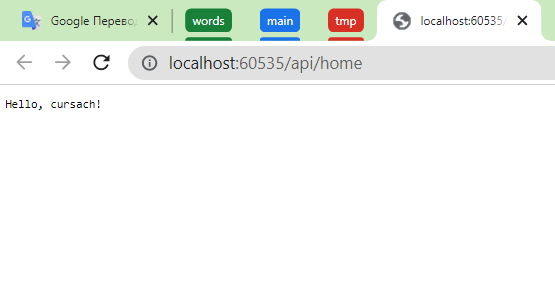
Для начала удалим в проекте класс WeatherForecast и контроллер по умолчанию, затем настроим, какой контроллер должен срабатывать при старте прогламмы в Properties/launchSettings.json в свойстве launchUrl:



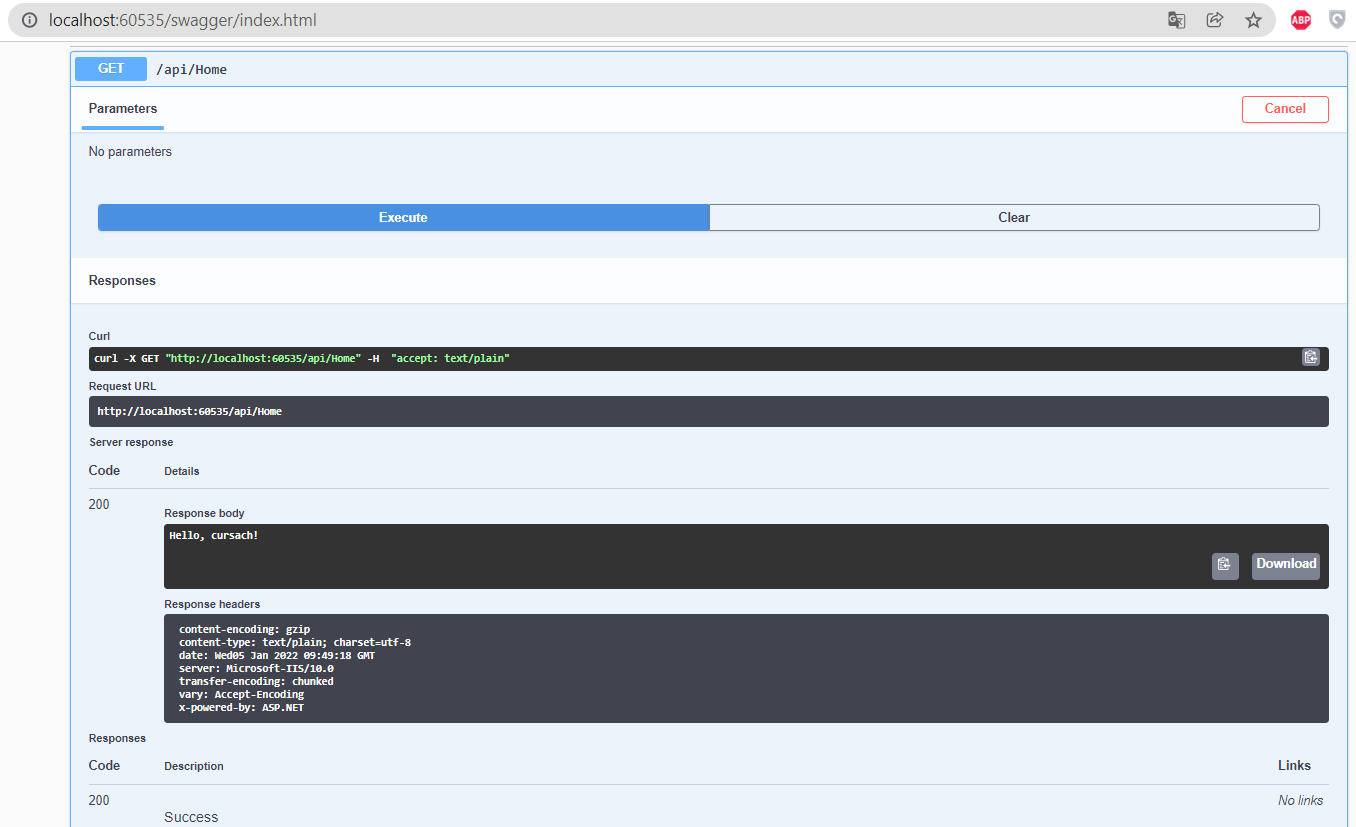
Далее создадим новый контроллер (когда мы создаем новый контроллер - выбираем "Контроллер API пустой"), в котором определим тестовый метод GetInfo()



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)

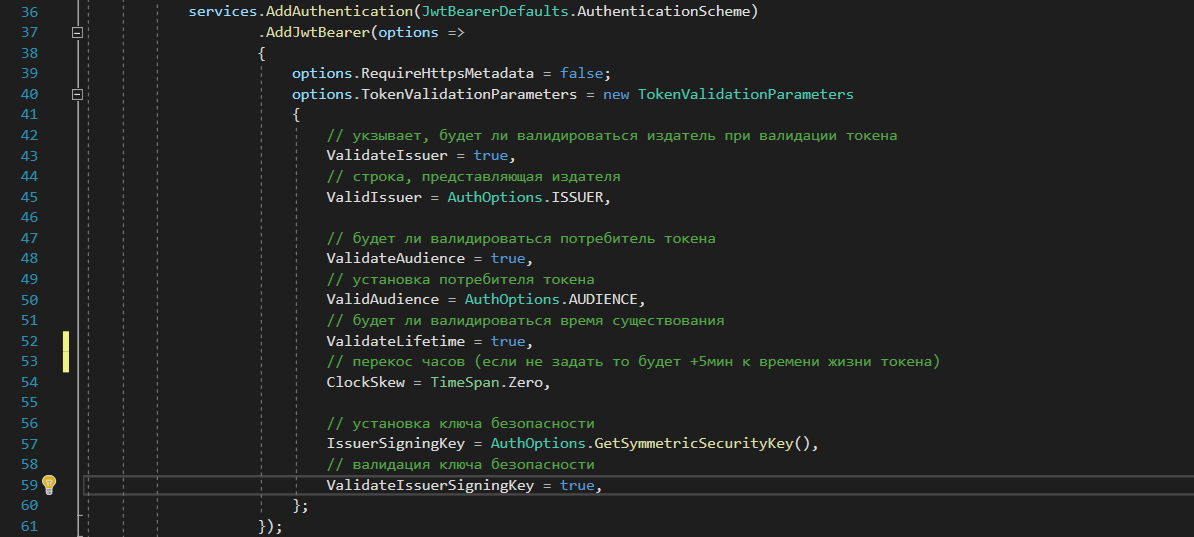
4) позволяет проецировать одну модель на другую – AutoMapper (v 10.1.1)

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)

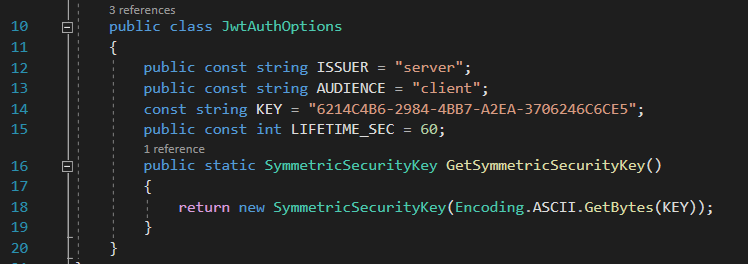
### 1.3 Startup

1. Метод ConfigureServices()

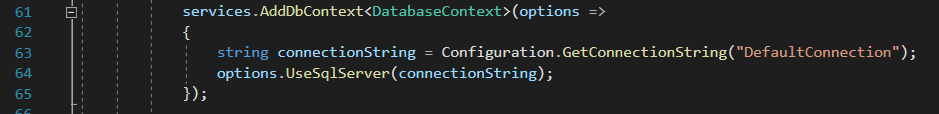
1) для JWT авторизации:



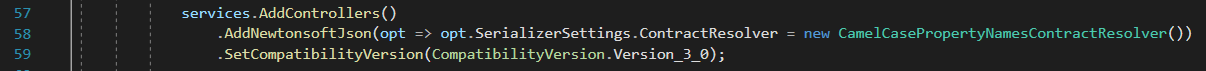
Так же создаем класс JwtAuthOptions, где указываем издателя токена, потребителя, ключ для шифрации и время жизни:



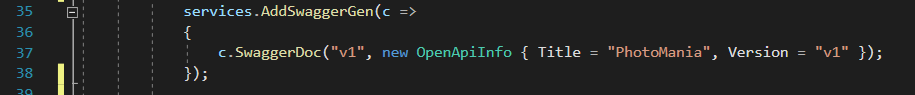
2) для работы с БД:



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):

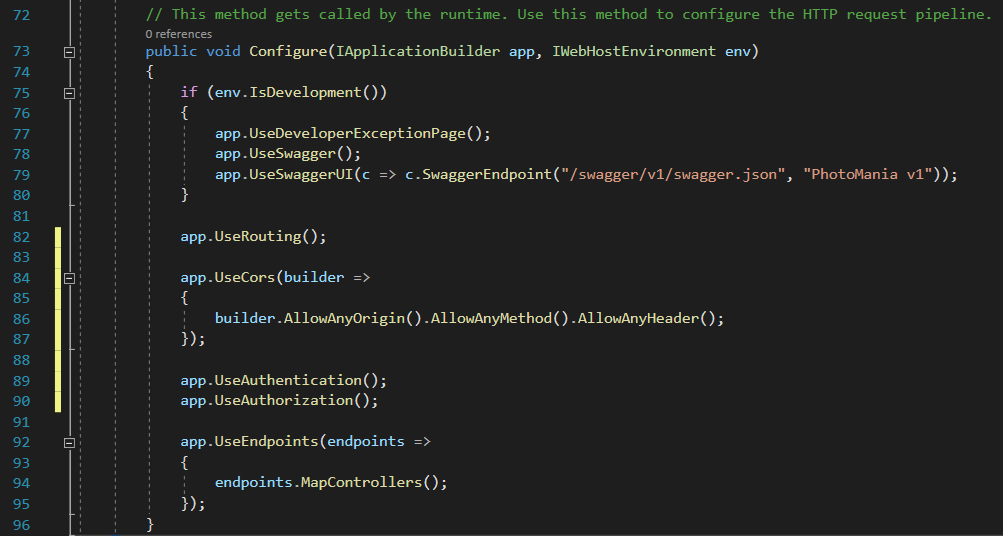


2. Метод Configure()

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)



### 1.4 Основные дирректории

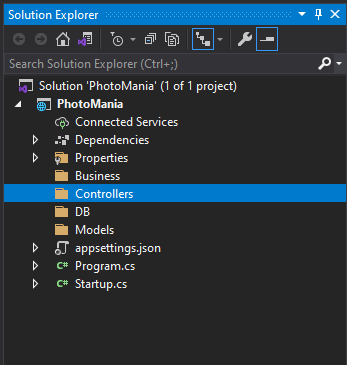
В апи, если нам нужно, можем добавлять отдельные слои по работе с БД, сервисы, потом просто получаем данные оттуда и возвращаем их клиенту, чем мы собственно и займемся.

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. От BaseEntity мы будем наследовать все остальные сущности.

Add ScreenShots

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

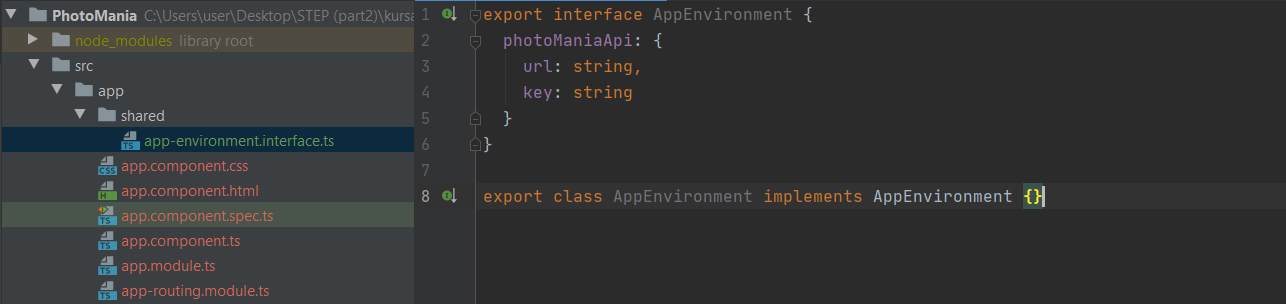
Пропишем так же для АПИ ConnectionStrings в appsettings.json

## 2. Client

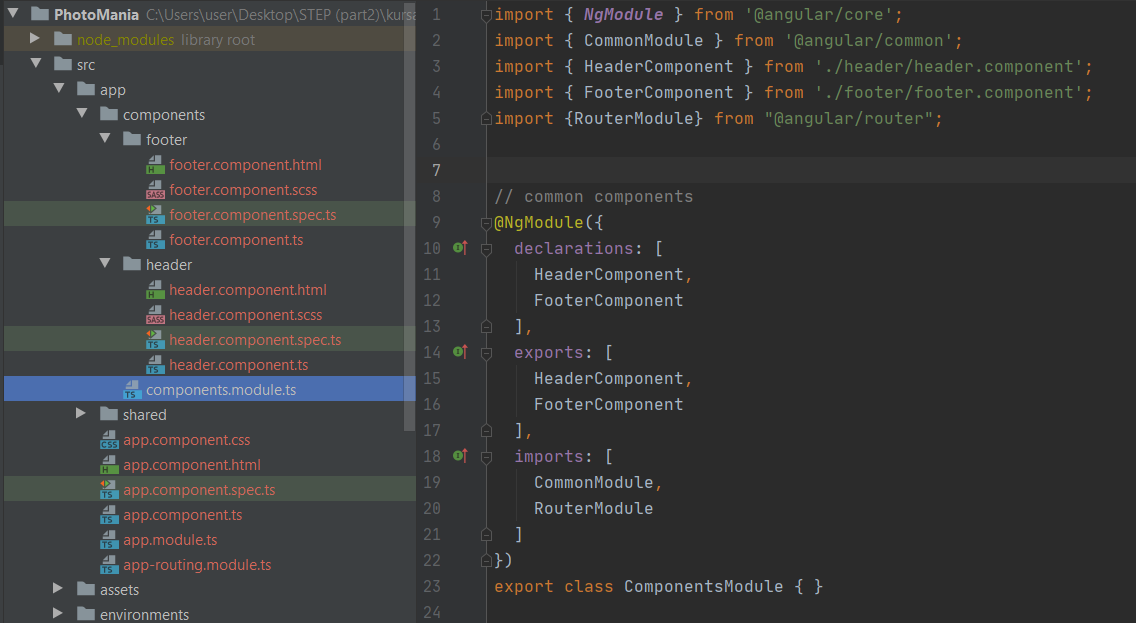
### 1.1 Создаем Angular проект и делаем первые шаги

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

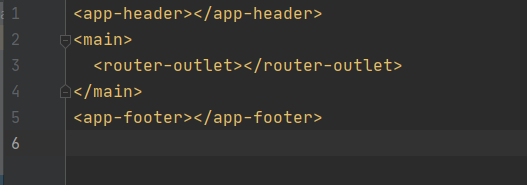
2. в папке shared создаем app-environment.interface.ts для удобного доступа к environment где ключ и ссылка на апи



3. 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)

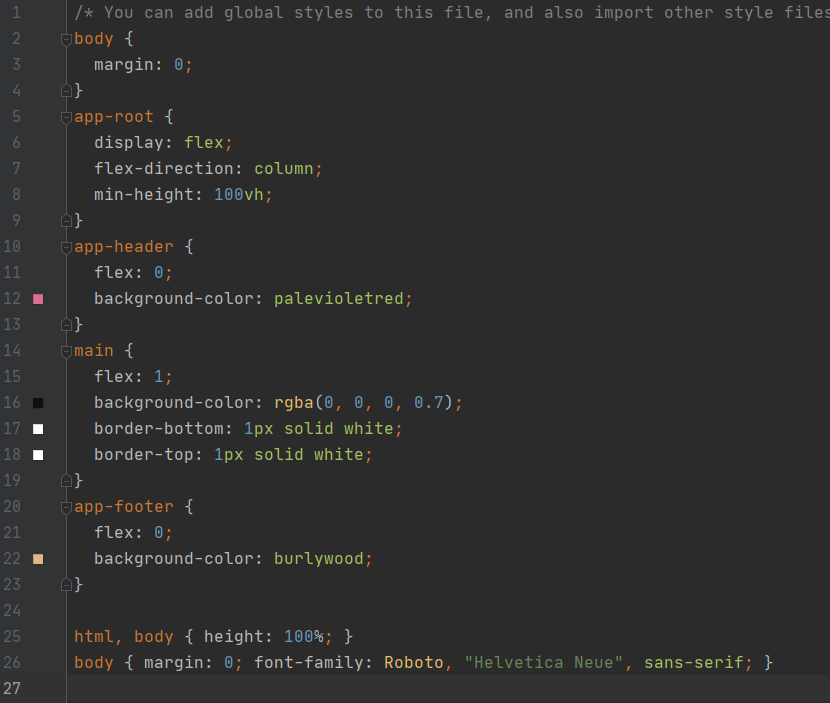


Подключим наш components.module.ts в app.module.ts, удалим содержимое файла app/app.component.html и вставим туда только что созданные хедер и футер

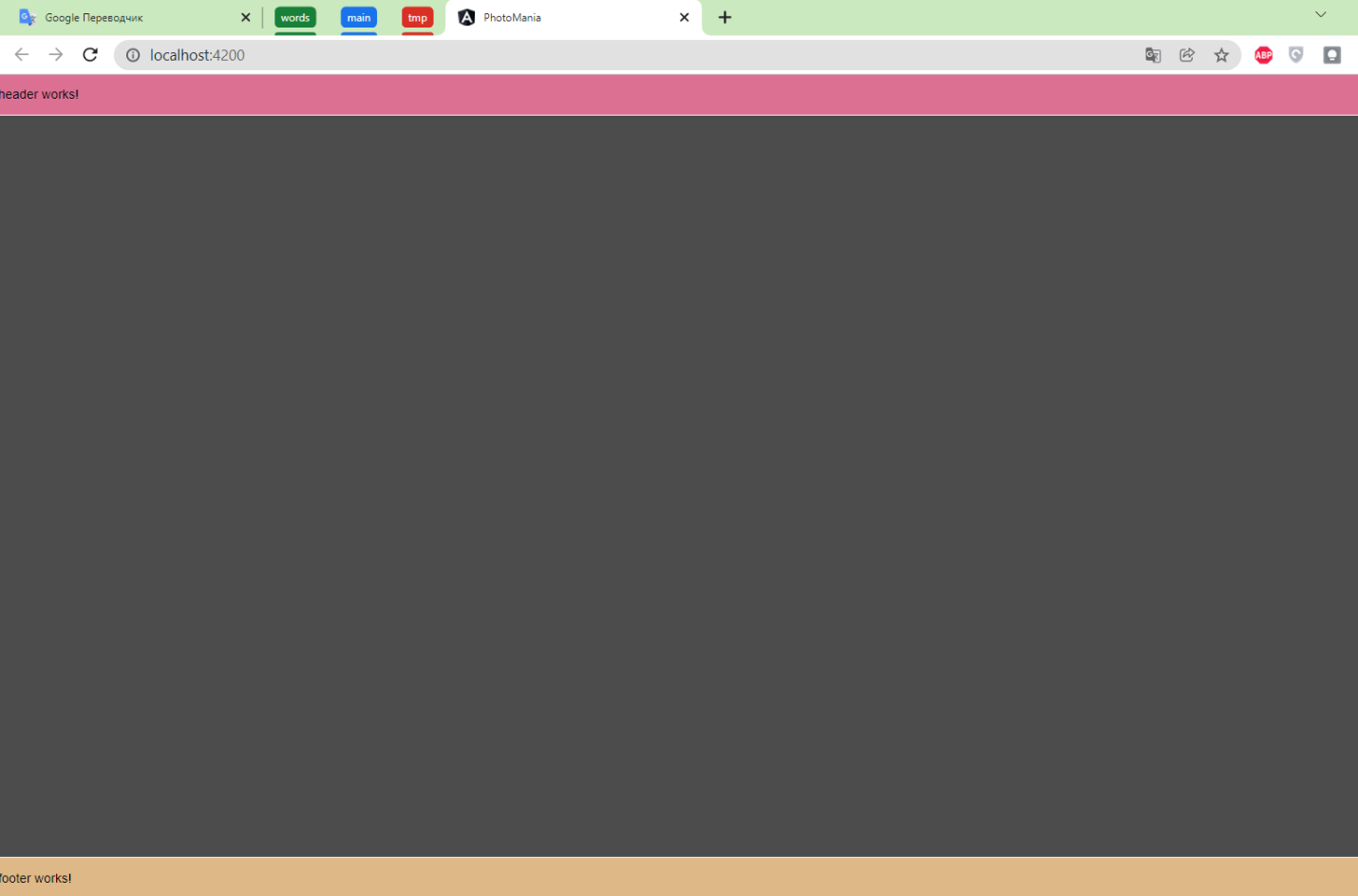


Таким образом хедер и футер изменяться не будет при переходе на другие страници, а страници, на которые будет переходить пользователь мы будем подставлять в тег main.

Немного глобальных стилей в src/styles.css

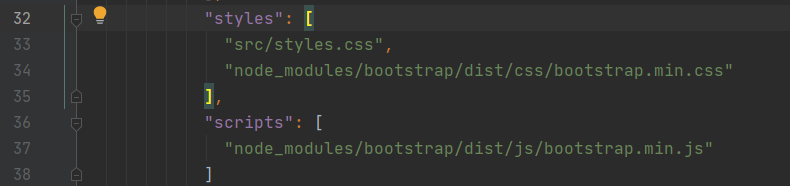


И у нас уже есть на что посмотреть в браузуре!



### 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):

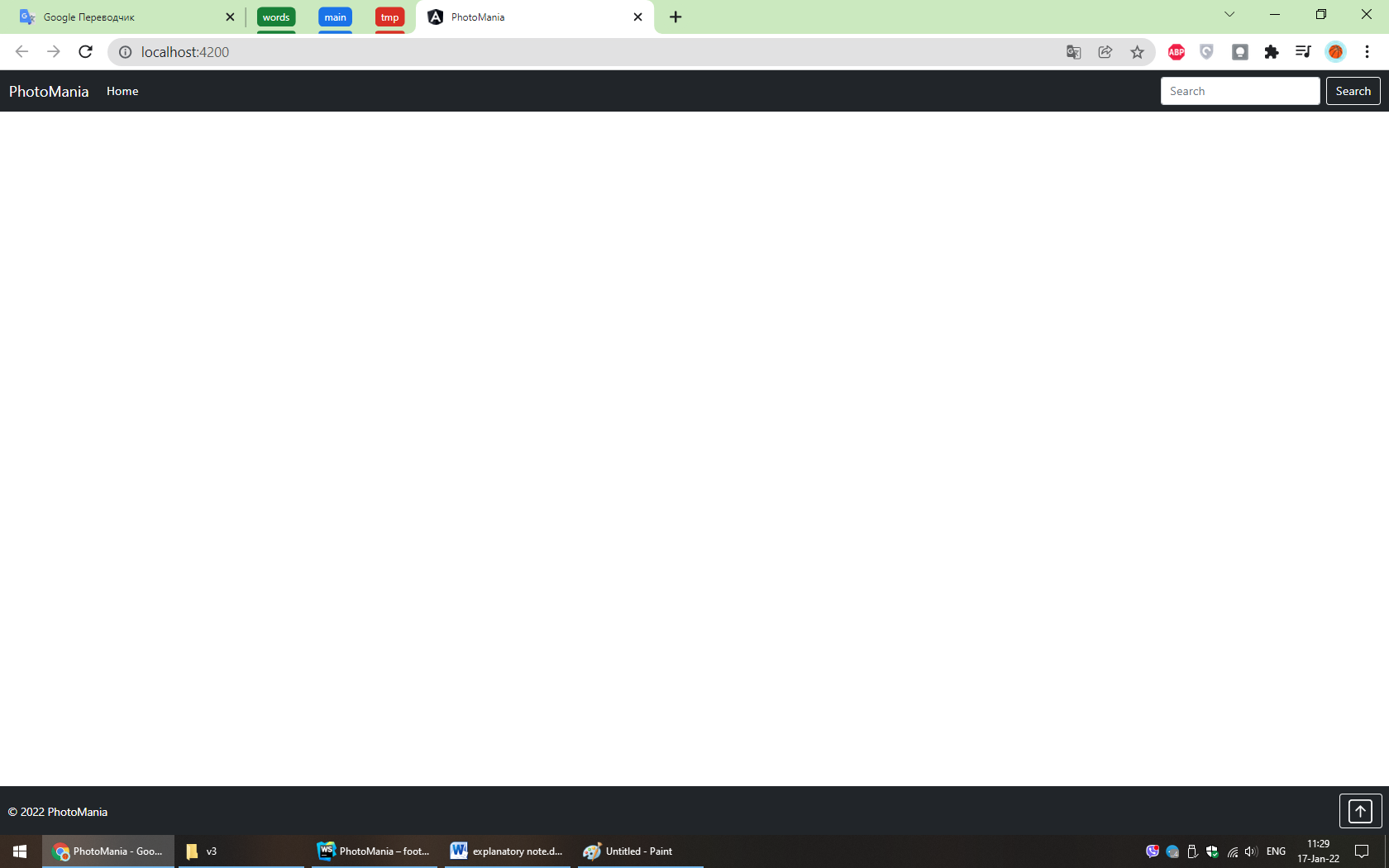


После чего перезапустим проект (ng serve). Возможно придется закрыть и открыть проект заново.

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:

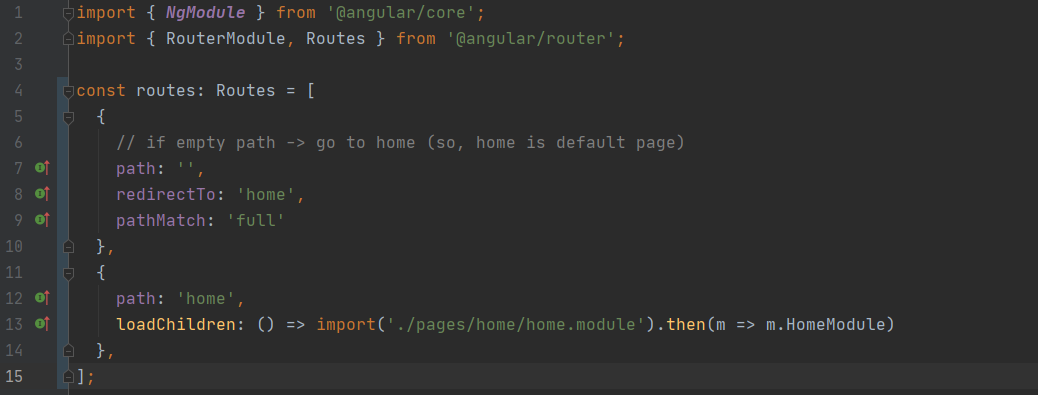
{

path: 'home',

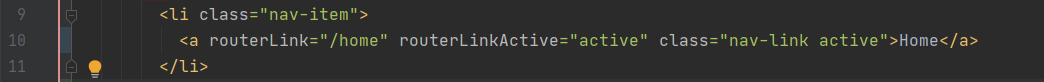
loadChildren: () => import('./pages/home/home.module').then(m => m.HomeModule)

}

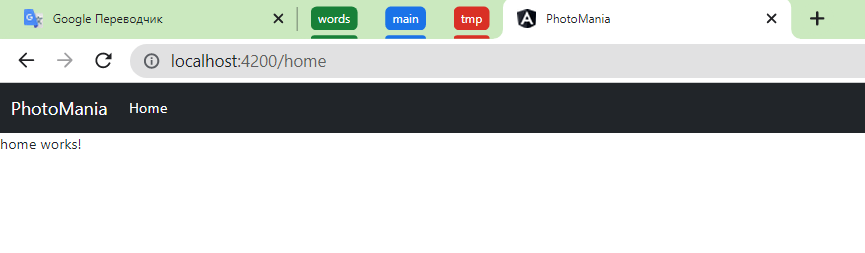
Так же сделаем переадресацию на эту страницу, если путь будет пустой. Важно что бы эта сапись была самой первой. В итоге получим в app-routing.module.ts таким:



В header подвязываем эту страницу к ссылке Home:

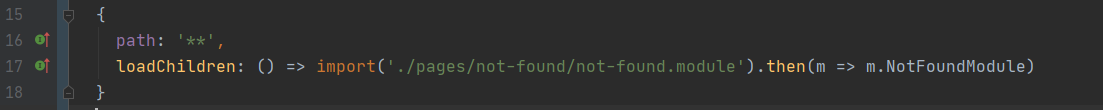


Теперь при загрузке сайта мы попадем сразу на главную страницу и при переходе по ссылке на главную страницу мы так же попадем на главную страницу:



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:

