

# Создание проекта ASP.Net Core MVC, используя Docker

## Установка необходимого ПО

Фреймворк ASP.NET Core MVC является частью платформы ASP.NET Core, его отличительная особенность - применение паттерна MVC. Преимуществом использования фреймворка ASP.NET Core MVC по сравнению с "чистым" ASP.NET Core является то, что он упрощает в ряде ситуаций и сценариев организацию и создание приложений, особенно это относится к большим приложениям.

Стоит отметить, сам паттерн MVC не является исключительной особенностью ASP.NET Core MVC, данный паттерн появился еще в конце 1970-х годов в компании Xerox как способ организации компонентов в графическом приложении на языке Smalltalk и в настоящее время применяется во многих платформах и для различных языков программирования. Особенно популярен паттерн MVC в веб-приложениях.

Концепция паттерна MVC предполагает разделение приложения на три компонента:

- **Модель (model):** описывает используемые в приложении данные, а также логику, которая связана непосредственно с данными, например, логику валидации данных. Как правило, объекты моделей хранятся в базе данных.

В MVC модели представлены двумя основными типами: модели представлений, которые используются представлениями для отображения и передачи данных, и модели домена, которые описывают логику управления данными.

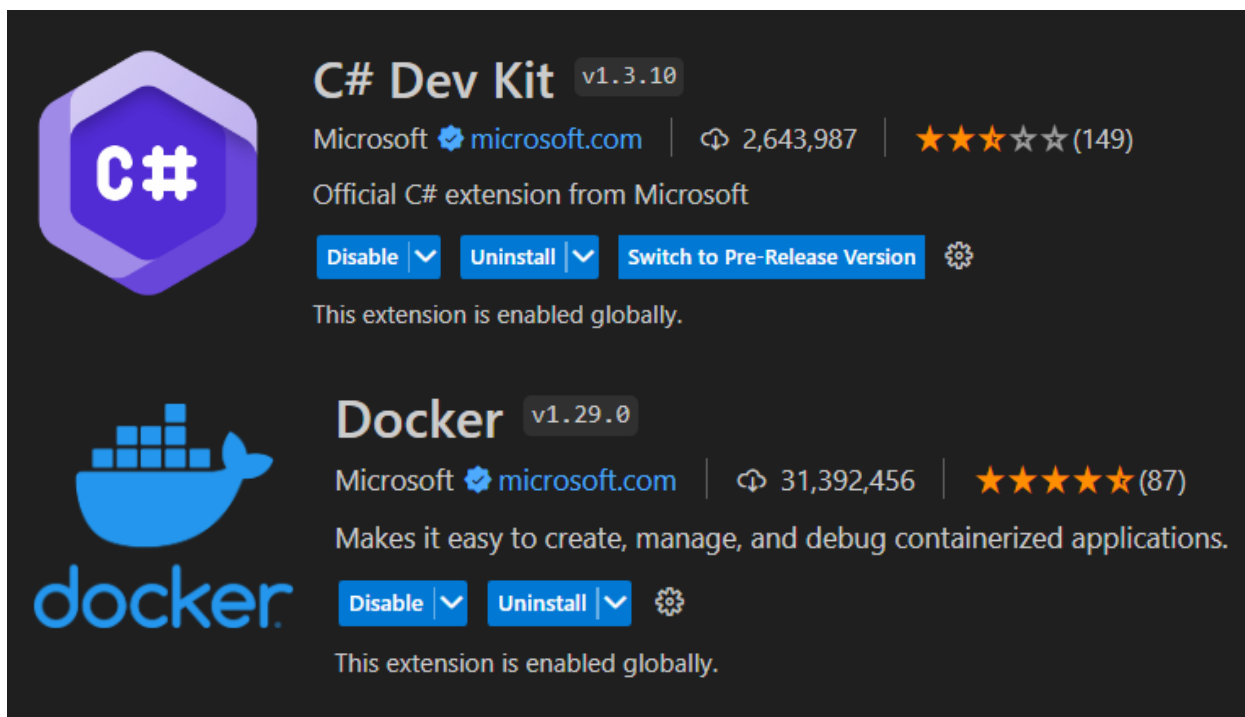
Модель может содержать данные, хранить логику управления этими данными. В то же время модель не должна содержать логику взаимодействия с пользователем и не должна определять механизм обработки запроса. Кроме того, модель не должна содержать логику отображения данных в представлении.

- **Представление (view):** отвечают за визуальную часть или пользовательский интерфейс, нередко html-страница, через который пользователь взаимодействует с приложением. Также представление может содержать логику, связанную с отображением данных. В то же время представление не должно содержать логику обработки запроса пользователя или управления данными.

- **Контроллер (controller):** представляет центральный компонент MVC, который обеспечивает связь между пользователем и приложением, представлением и хранилищем данных. Он содержит логику обработки запроса пользователя. Контроллер получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления, наполненного данными моделей.

Перед созданием проекта необходимо установить дополнительное ПО. Нас интересует Visual Studio Code (VS Code) — это редактор исходного кода. Его разработал Microsoft для всех популярных операционных систем: Windows, Linux и macOS. (можно скачать по ссылке <https://code.visualstudio.com/download>)

В VS Code можно скачивать дополнения для помощи в разработке. В случае с этим проектом нам потребуются C# Dev Kit (помощь в разработке на ЯП C#) и Docker (помощь в написании файлов настройки Docker контейнеров).



Также для создания и работы над проектом потребуется .Net (скачать можно по [ссылке](#)) — это бесплатная кроссплатформенная платформа разработчика с открытым кодом для создания множества приложений. Он может запускать программы, написанные на нескольких языках, при этом C# является наиболее популярным. Она использует высокопроизводительную среду выполнения, которая используется в рабочей среде многими высокомасштабируемыми приложениями.

Как все работает:

Вспомним, что такое процесс компиляции — это перевод вашего кода, понятного человеку, в бинарный код, понятный компьютеру. В программировании на .net компиляция и запуск приложений происходит следующим образом: Код из любого языка преобразовывается в код, написанный на общем языке (Common intermediate language или CIL). Этот язык является языком низшего уровня, похожего по синтаксису на язык ассемблер. После, этот код передаётся так называемой исполняющей среде (Common language runtime или CLR), которая берёт функции и методы из .net Framework. После этого конечный результат передаётся на процессор и выполняется программа.

Так как приложение будет работать в Docker, необходимо его скачать ([ссылка для скачивания](#)). Docker — это проект с открытым исходным кодом для автоматизации развертывания приложений в виде переносимых автономных контейнеров, выполняемых в облаке или локальной среде. Одновременно с этим, Docker — это компания, которая разрабатывает и продвигает эту технологию в сотрудничестве с поставщиками облачных служб, а также решений Linux и Windows, включая корпорацию Microsoft.

Контейнеры Docker могут работать в любой среде, например в локальном центре обработки данных, в службе стороннего поставщика или в облаке Azure. Контейнеры образов Docker работают в исходном формате в Linux и Windows. Но образы Windows будут выполняться только на узлах Windows, тогда как образы Linux — на узлах Linux или Windows (на данный момент с помощью виртуальной машины Linux Hyper-V). Термин "узлы" здесь означает физические серверы и виртуальные машины.

Разработчики могут использовать среды разработки на базе Windows, Linux или macOS. На компьютере разработчика выполняется узел Docker, где развернуты образы Docker с создаваемым приложением и всеми его зависимостями. Разработчики, работающие в Linux или macOS, могут использовать узел Docker на базе Linux и создавать

образы только для контейнеров Linux. (В macOS разработчики могут изменять код приложения и запускать Docker CLI в macOS, но на момент написания этой статьи они не могут запускать контейнеры непосредственно в macOS.) В Windows разработчики могут создавать образы для контейнеров Linux или Windows.

Docker предоставляет Docker Desktop для Windows и macOS, позволяя размещать контейнеры в среде разработки и использовать дополнительные средства разработки. Оба продукта устанавливают необходимую виртуальную машину (узел Docker) для размещения контейнеров.

В приложении планируется внедрение СУБД PostgreSQL. Для простоты пользования база данных будет также храниться в контейнере Docker, в виду того, что сами контейнеры легковесные, что позволяет использовать БД, не занимая много места на самом ПК.

Для связи контейнеров используется docker compose. Docker Compose — инструмент, позволяющий запускать среды приложений с несколькими контейнерами на основе определений, задаваемых в файле YAML. Он использует определения служб для построения полностью настраиваемых сред с несколькими контейнерами, которые могут использовать общие сети и тома хранения данных.

После установки .Net становится доступной команда *dotnet*, с помощью данной команды можно управлять проектом, добавлять библиотеки, собирать, запускать проект и многое другое.

Сам шаблон команды выглядит так:

```
dotnet [runtime-options] [path-to-application] [arguments]
```

Вот основные команды dotnet:

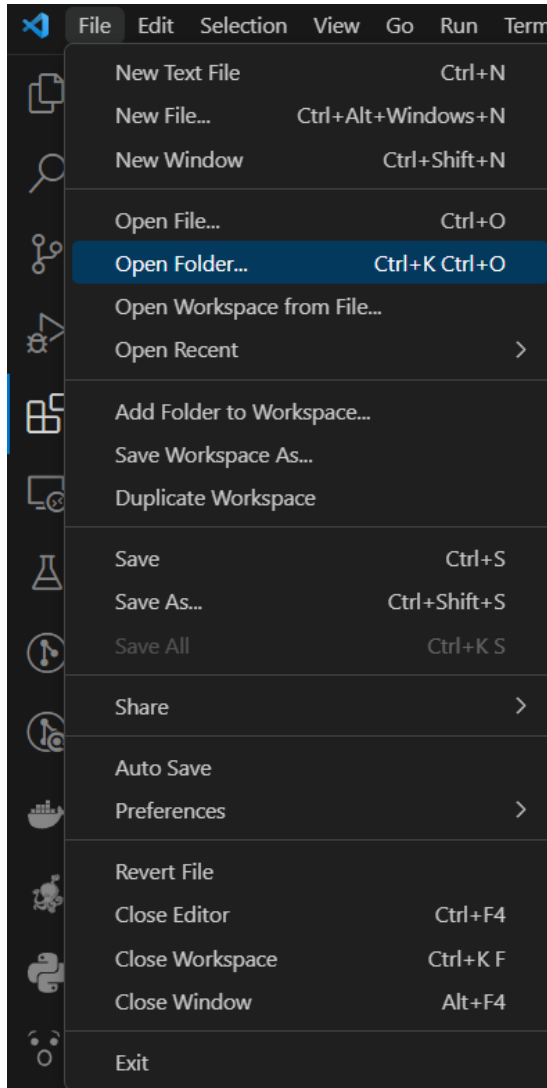
add	Добавление пакета или ссылки в проект .NET.
build	Сборка проекта .NET.
build-server	Взаимодействие с серверами, запущенными в ходе сборки.
clean	Очистка выходных данных сборки проекта .NET.
format	Применение настроек стилей к проекту или решению.
help	Показать справку командной строки.
list	Вывод списка ссылок на проекты в проекте .NET.
msbuild	Выполнение команд Microsoft Build Engine (MSBuild).
new	Создание нового файла или проекта .NET.
nuget	Предоставление дополнительных команд NuGet.
pack	Создание пакета NuGet.
publish	Публикация проекта .NET для развертывания.
remove	Удаление пакета или ссылки из проекта .NET.
restore	Восстановление зависимостей, указанных в проекте .NET.
run	Сборка и запуск проекта .NET.
sdk	Управление установкой пакета SDK .NET.
sln	Изменение файлов решения Visual Studio.
store	Сохранение указанных сборок в хранилище пакетов среды выполнения.
test	Запуск модульных тестов с помощью средства, указанного в проекте .NET.
tool	Установка и настройка инструментов, расширяющих возможности .NET.
vstest	Выполнение команд Microsoft Test Engine (VSTest).
workload	Управление необязательными рабочими нагрузками.

При работе с Docker управлять контейнерами можно с помощью команд *docker* и *docker-compose* ([параметры команды docker](#) , [параметры команды docker-compose](#))

## Создание проекта

Для создания проекта необходимо.

1. Открыть Visual Studio Code, выбрать File -> Open Folder, далее выбрать необходимую директорию.



2. Нажать комбинацию клавиш *CTRL+SHIFT* + ~ для открытия командой строки внутри VS Code.

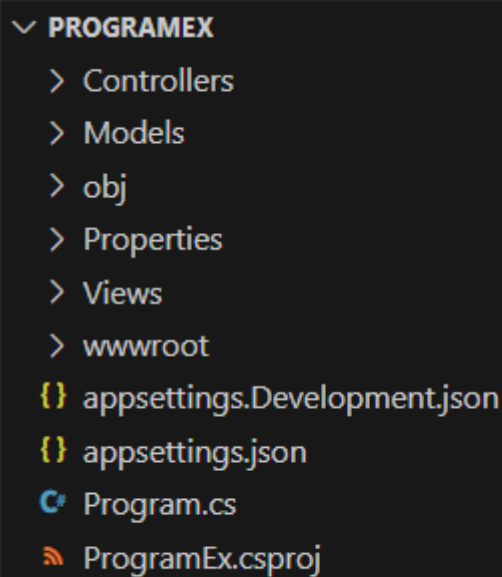
3. Ввести команду *dotnet new mvc* для создания проекта ASP.Net Core MVC.

После выполненных операций мы имеем созданный проект с необходимым расположением файлов.

```
PS D:\for_work\teacher\C_sharp_programming\ProgramEx> dotnet new mvc
Шаблон "Веб-приложение ASP.NET Core (модель-представление-контроллер)" успешно создан
.
Этот шаблон содержит технологии сторонних производителей, кроме Майкрософт. Дополнительные сведения см. в разделе https://aka.ms/aspnetcore/8.0-third-party-notices.

Идет обработка действий после создания...
Восстановление D:\for_work\teacher\C_sharp_programming\ProgramEx\ProgramEx.csproj:
  Определение проектов для восстановления...
  Восстановлен D:\for_work\teacher\C_sharp_programming\ProgramEx\ProgramEx.csproj (за 526 ms).
Восстановление выполнено.

PS D:\for_work\teacher\C_sharp_programming\ProgramEx> 
```



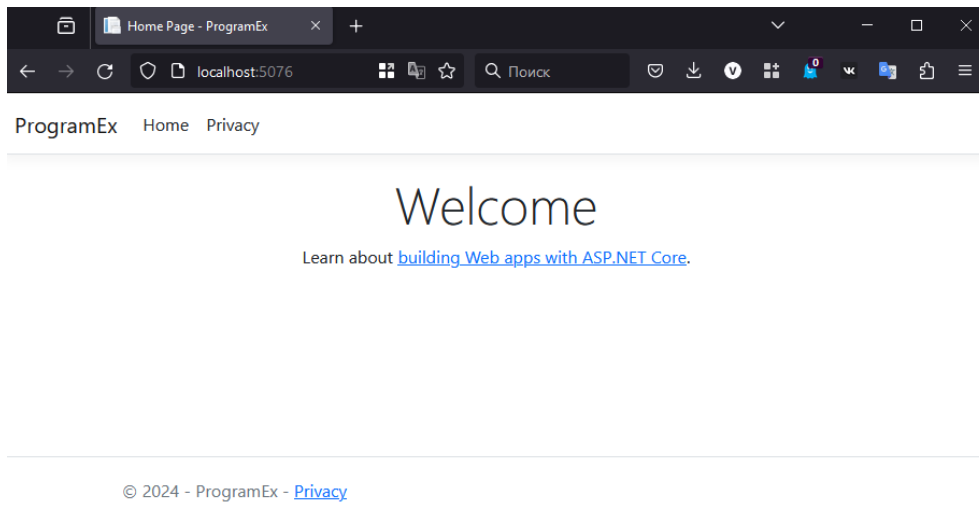
```
▼ PROGRAMEX
  > Controllers
  > Models
  > obj
  > Properties
  > Views
  > wwwroot
  {} appsettings.Development.json
  {} appsettings.json
  C# Program.cs
  ProgramEx.csproj
```

- **wwwroot:** этот узел (на жестком диске ему соответствует одноименная папка) предназначен для хранения статических файлов - изображений, скриптов javascript, файлов css и т.д., которые используются приложением.
- **Controllers:** папка для хранения контроллеров, используемых приложением. По умолчанию здесь уже есть один контроллер - HomeController
- **Models:** каталог для хранения моделей. По умолчанию здесь создается модель ErrorViewModel
- **Views:** каталог для хранения представлений. Здесь также по умолчанию добавляются ряд файлов - представлений
- **appsettings.json:** хранит конфигурацию приложения
- **Program.cs:** файл, который определяет входную точку в приложение ASP.NET Core.

Сам проект создан, можно ввести команду `dotnet run`

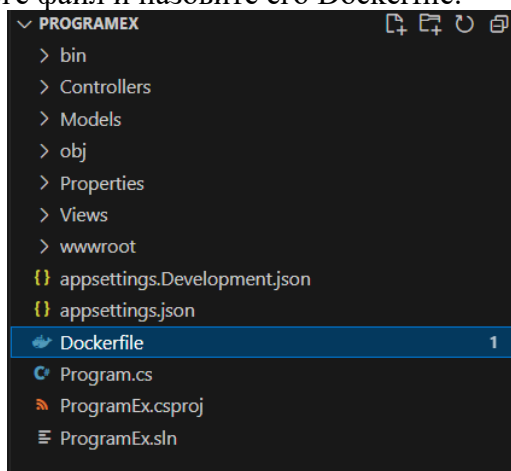
```
PS D:\for_work\teacher\C_sharp_programming\ProgramEx> dotnet run
Сборка...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5076
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\for_work\teacher\C_sharp_programming\ProgramEx
```

При запуске консоль отобразит адрес, по которому доступен проект. В моем случае это `http://localhost:5076`. И если мы обратимся по адресу запущенного приложения, то сработает запрос к контроллеру по умолчанию - классу `HomeController`, который выберет для генерации ответа нужное представление. И в итоге из представления будет создана html-страница, которую мы увидим в своем веб-браузере:



## Размещение проекта в контейнере Docker

Далее нам необходимо поместить проект в контейнер, для этого создайте в корневом каталоге файл и назовите его `Dockerfile`.



В Dockerfile добавьте следующий код

```
Dockerfile X launchSettings.json
Dockerfile > ...
1 FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
2 WORKDIR /app
3
4 FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
5 WORKDIR /src
6
7 COPY ["ProgramEx.csproj", "ProgramEx.csproj"]
8 RUN dotnet restore "ProgramEx.csproj"
9 COPY . .
10 WORKDIR "/src/"
11 RUN dotnet build "ProgramEx.csproj" -c Release -o /app
12
13 FROM build AS publish
14 RUN dotnet publish "ProgramEx.csproj" -c Release -o /app
15
16 FROM base AS final
17 WORKDIR /app
18 COPY --from=publish /app .
19 ENTRYPOINT ["dotnet", "ProgramEx.dll"]
20
21
```

Замените *ProgramEx* на название вашего проекта.

Для сборки контейнера введите команду

*“docker build -t название\_контейнера\_в\_нижнем\_регистре .”*.

```
PS D:\for_work\teacher\C_sharp_programming\ProgramEx> docker build -t program_ex:testing .
[+] Building 0.4s (18/18) FINISHED
=> [internal] load build definition from Dockerfile 0.05s
=> => transferring dockerfile: 32B 0.05s
=> => transferring context: 2B 0.05s
=> [internal] load metadata for mcr.microsoft.com/dotnet/sdk:8.0 0.05s
=> [internal] load metadata for mcr.microsoft.com/dotnet/aspnet:8.0 0.35s
=> [build 1/7] FROM mcr.microsoft.com/dotnet/sdk:8.0 0.05s
=> => transferring context: 9.08kB 0.05s
=> [base 1/2] FROM mcr.microsoft.com/dotnet/aspnet:8.0@sha256:789045ecae51d62d07877994d567eff4442b7bbd4121867898ee7bf00b7241ea 0.05s
=> CACHED [base 2/2] WORKDIR /app 0.05s
=> CACHED [final 1/2] WORKDIR /app 0.05s
=> CACHED [build 2/7] WORKDIR /src 0.05s
=> CACHED [build 3/7] COPY [ProgramEx.csproj, ProgramEx.csproj] 0.05s
=> CACHED [build 4/7] RUN dotnet restore "ProgramEx.csproj" 0.05s
=> CACHED [build 5/7] COPY . . 0.05s
=> CACHED [build 6/7] WORKDIR /src/ 0.05s
=> CACHED [build 7/7] RUN dotnet build "ProgramEx.csproj" -c Release -o /app 0.05s
=> CACHED [publish 1/1] RUN dotnet publish "ProgramEx.csproj" -c Release -o /app 0.05s
=> CACHED [final 2/2] COPY --from=publish /app . 0.05s
=> exporting to image 0.05s
=> => exporting layers 0.05s
=> => writing image sha256:83e173405ae0e6ff8f5abede4f906e2f54f3e3cd123e3751b1faf42dafb64d1d 0.05s
=> => naming to docker.io/library/program_ex:testing 0.05s
```

Для запуска контейнера используйте следующую команду:

*docker run -it -p 5076:8080 название\_контейнера*



```
PS D:\for_work\teacher\C_sharp_programming\ProgramEx> docker run -it -p 5076:8080 program_ex:testing
warn: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[60]
      Storing keys in a directory '/root/.aspnet/DataProtection-Keys' that may not be persisted outside
      of the container. Protected data will be unavailable when container is destroyed. For more informati
      on go to https://aka.ms/aspnet/dataprotectionwarning
warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
      No XML encryptor configured. Key {9d07ccff-8f68-415b-9583-e742d0e2427c} may be persisted to stor
      age in unencrypted form.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://[::]:8080
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /app
```

Теперь, при переходе по адресу <http://localhost:5076> вас будет приветствовать то же самое окно, что и было при запуске вне контейнера.

## Создание docker-compose

Но ввиду того, что планируется использование базы данных из контейнера, то для объединения их нам потребуется docker-compose.

Первым делом нам потребуется создать файл *docker-compose.yaml* на директорию выше, чем расположен наш проект.

После мы будем добавлять код в файл.

Первым делом укажем версию docker-compose, в нашем случае 3.7

```
1  version: '3.7'
```

Так как у нас будут 2 контейнера Docker работать, то их над связать между собой, для этого пропишите следующий код.

```
3  networks:
4    web-release:
5      driver: bridge
```

То есть мы создали соединение «мост» и назвали его *web-release*.

Далее будем прописывать контейнеры, которые имеют название «сервисы». Сначала пропишем контейнер, на котором будет храниться БД. Использовать мы будет в качестве СУБД PostgreSQL.



```

services:
  db_postgres:
    image: postgres
    volumes:
      - ./db:/var/lib/postgres/data/
    environment:
      - POSTGRES_PASSWORD=root
      - POSTGRES_USER=root
      - POSTGRES_DB=db
    ports:
      - 5432:5432
    networks:
      - web-release

```

*db* – название контейнера.

*volumes* – указывает где будут храниться данные БД.

*environment* – переменные окружения, в данном случае мы задаем пароль root, имя пользователя root и база данных будет называться db.

*ports* – указывает какой порт необходимо читать в самом контейнере (он прописан слева от двоеточия) и через какой порт это будет читаться на самом компьютере.

*networks* – указывает на то, через какую сеть будет проведен контейнер.

Далее зададим правила для контейнера, на котором будет крутиться сам проект MVC.

```

21   mvc-proj:
22     image: mvc
23     build:
24       context: .
25       dockerfile: ProgramEx/Dockerfile
26     ports:
27       - 5199:8080
28     networks:
29       - web-release
30

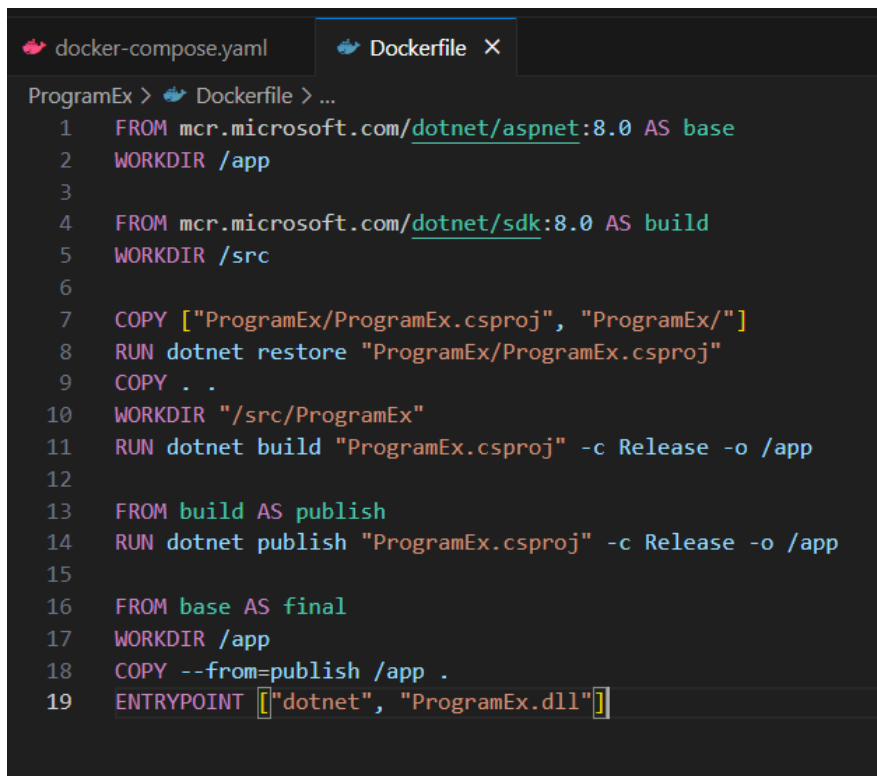
```

*image* – название образа.

*build* – указывает какой Dockerfile будет читаться. (*context* – какую директорию будем использовать в качестве начальной точки, *dockerfile* – путь к самому Dockerfile относительно контекста).

Инструкции *ports* и *networks* нам уже знакомы.

Перед запуском docker-compose нам необходимо немного отредактировать наш Dockerfile.



```
ProgramEx > Dockerfile > ...
1 FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
2 WORKDIR /app
3
4 FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
5 WORKDIR /src
6
7 COPY ["ProgramEx/ProgramEx.csproj", "ProgramEx/"]
8 RUN dotnet restore "ProgramEx/ProgramEx.csproj"
9 COPY . .
10 WORKDIR "/src/ProgramEx"
11 RUN dotnet build "ProgramEx.csproj" -c Release -o /app
12
13 FROM build AS publish
14 RUN dotnet publish "ProgramEx.csproj" -c Release -o /app
15
16 FROM base AS final
17 WORKDIR /app
18 COPY --from=publish /app .
19 ENTRYPOINT ["dotnet", "ProgramEx.dll"]
```

Так как docker-compose файл расположен над директорией самого проекта, то и сам Dockerfile тоже, по сути, будет читаться в контексте расположения docker-compose, следовательно нам необходимо дописать путь к файлу *ProgramEx.csproj*, что, собственно, и сделано на изображении выше.

После внесения всех необходимых изменений мы можем запустить проект через команду *docker-compose up --build*. Проект запущен и можно также в браузере перейти по адресу <http://localhost:5076> и все будет работать.

## Подключение к базе данных

Для работы с БД воспользуемся технологией Entity Framework Core.

Entity Framework (EF) Core — это простая, кроссплатформенная и расширяемая версия популярной технологии доступа к данным Entity Framework с открытым исходным кодом.

EF Core может использоваться в качестве объектно-реляционного модуля сопоставления (ORM), который:

- Позволяет разработчикам .NET работать с базой данных с помощью объектов .NET.
- Устраняет необходимость в большей части кода для доступа к данным, который обычно приходится писать.

EF Core поддерживает множество систем баз данных.

В EF Core доступ к данным осуществляется с помощью модели. Модель состоит из классов сущностей и объекта контекста, который представляет сеанс взаимодействия с базой данных. Объект контекста позволяет выполнять запросы и сохранять данные.

EF поддерживает следующие подходы к разработке моделей:

- Создание модели на основе существующей базы данных.
- Написание кода модели вручную в соответствии с базой данных.

Для того, чтобы добавить библиотеку в проект, вставьте следующий код в файл *ProgramEx.csproj*.

```

1 <Project Sdk="Microsoft.NET.Sdk.Web">
2
3   <PropertyGroup>
4     <TargetFramework>net8.0</TargetFramework>
5     <Nullable>enable</Nullable>
6     <ImplicitUsings>enable</ImplicitUsings>
7   </PropertyGroup>
8
9   <ItemGroup>
10    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="8.0.2" />
11    <PackageReference Include="Npgsql.EntityFrameworkCore.PostgreSQL" Version="8.0.0" />
12    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.5.0" />
13    <PackageReference Include="System.Configuration.ConfigurationManager" Version="8.0.0" />
14    <PackageReference Include="System.ComponentModel.Annotations" Version="5.0.0" />
15  </ItemGroup>
16
17 </Project>

```

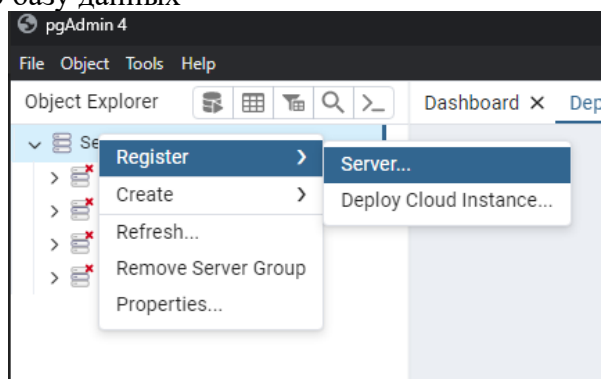
Через тег `PackageReference` можно добавлять различные библиотеки в проект .Net

После вставки необходимых ресурсов просто соберите проект, в свою очередь подгрузятся все библиотеки (для запуска проекта через docker предварительная сборка не требуется, так как в самом контейнере прописана инструкция по сборке решения).

В нашей базе будут использоваться 1 сущность под названием *todoelement*.

Запустим docker-compose и зайдем в pgAdmin.

Создадим новую базу данных



Назовем бд *leaning\_mvc*

**Register - Server**

General Connection Parameters SSH Tunnel Advanced

Name: learning\_mvc

Server group: Servers

Background: ☐

Foreground: ☐

Connect now?: ☒

Comments:

Either Host name or Service must be specified.

Close Reset Save

Во вкладке *Connection* введем следующие параметры

**Register - Server**

General Connection Parameters SSH Tunnel Advanced

Host name/address: 127.0.0.1

Port: 5432

Maintenance database: db

Username: root

Kerberos authentication?: ☐

Password: ....

Save password?: ☐

Role:

Service:

Close Reset Save

Пропишем те данные, которые введены в переменных окружения в файле docker-compose

```
db_postgres:
  image: postgres
  volumes:
    - ./db:/var/lib/postgres/data/
  environment:
    - POSTGRES_PASSWORD=root
    - POSTGRES_USER=root
    - POSTGRES_DB=db
  ports:
    - 5432:5432
  networks:
    - web-release
```

Создадим таблицу в БД. Таблица будет называться *to\_do\_elements*. Таблица состоит из 3 полей:

- id – первичный ключ, тип данных serial;
- text – текст элемента todo, тип данных character varying;
- completed – элемент, который хранит состояние задачи, тип данных boolean.

Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
id	serial			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
text	character varying			<input checked="" type="checkbox"/>	<input type="checkbox"/>	
completed	boolean			<input checked="" type="checkbox"/>	<input type="checkbox"/>	

После создания БД необходимо подключиться к ней в самом проекте. Строка подключения обычно хранится в конфигурационном файле *appsettings.json*. Добавим в этот файл элемент *ConnectionStrings*, в который, в свою очередь, добавим элемент *DefaultConnection*.

Синтаксис строки подключения к БД отличается в зависимости от СУБД. В PostgreSQL она выглядит следующим образом:

*User ID= username;Password=password;Host=localhost;Database=postgres;*

Конечно же, значения переменных могут отличаться.

Пропишем все то, что ранее упоминалось:

```
ProgramEx > {} appsettings.json > ...
You, 10 seconds ago | 1 author (You)
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft.AspNetCore": "Warning"
6      }
7    },
8    "ConnectionStrings": {
9      "DefaultConnection": "User ID=root;Password=root;Host=db_postgres;Port=5432;Database=db;"
10   },
11   "AllowedHosts": "*"
12 }
13
```

Теперь нужно настроить подключение к БД через EntityFramework Core, используя паттерн репозиторий.

Первым делом в папке *Models* создадим 2 класса. Первый класс называется *Entity*, в нем будет только одно поле *Id*.

```
ProgramEx > Models > Entity.cs > ...
1  using System.ComponentModel.DataAnnotations;
2  using System.ComponentModel.DataAnnotations.Schema;
3
4  namespace ProgramEx.Models
5  {
6      0 references
7      public class Entity
8      {
9          [Key]
10         [Column("id")]
11         0 references
12         public int Id { get; set; }
13     }
14 }
```

В C# существует аннотация данных, ее атрибуты определяют различные правила для отображения свойств модели. По большей части аннотация действительно необходима, когда в Postgres используются типы данных, которые сложно соотнести с типами данных/классами в C#. Аннотации также будут полезны, когда имена объектов БД отличается от имен в проекте (ввиду разных стандартов наименования).

В нашем случае используются атрибуты *[Key]*, который явно указывает на то, что свойство *Id* является первичным ключом, и атрибут *[Column("id")]*, благодаря ему мы явно указывается с каким атрибутом таблицы соотносить данное поле класса.

Далее создадим класс *ToDoElement*, который как раз и будет соответствовать таблице *to\_do\_elements* в базе данных.

```

1  using System.ComponentModel.DataAnnotations.Schema;
2
3  namespace ProgramEx.Models
4  {
5
6      [Table("to_do_elements")]
7      1 reference
8      public class ToDoElement : Entity
9      {
10         [Column("text")]
11         1 reference
12         public string Text {get; set;}
13
14         [Column("completed")]
15         1 reference
16         public bool Completed {get; set;}
17
18         0 references
19         public ToDoElement(string text)
20         {
21             Id = 0;
22             Text = text;
23             Completed = false;
24         }
25     }
26 }

```

Как можно заметить, этот класс наследует свойства класса *Entity*, то есть у него есть свойство *Id*. В классе прописан конструктор, сделано это по той причине, что при создании экземпляра класса необходимо прописать все not null поля для корректной записи в базу (при создании новой задачи в поле *completed* сразу прописывается значение false, в добавок поле *Id* в БД у нас является счетчиком, из-за чего в программе не будет продуман алгоритм генерирования этого поля).

Для описывания набора сущностей нам необходимо создать еще один класс. Для начала создайте папку *Data* в проекте, далее создайте класс *ToDoElementContext*. Код этого класса будет выглядеть следующим образом.

```

ProgramEx > Data > TodoElementsContext.cs > TodoElementsContext > OnModelCreating
1  using Microsoft.EntityFrameworkCore;
2  using ProgramEx.Models;
3
4  namespace ProgramEx.Data
5  {
6      1 reference
7      public class TodoElementsContext(DbContextOptions<TodoElementsContext> options) : DbContext(options)
8      {
9          0 references
10         public DbSet<ToDoElement> TodoElements { get; set; }
11
12         0 references
13         protected override void OnModelCreating(ModelBuilder modelBuilder)
14         {
15             modelBuilder.Entity<ToDoElement>(entity =>
16             {
17                 entity.Property(e => e.Id).UseIdentityColumn();
18             });
19         }
20     }
21 }

```



В строке 8 мы создаем объект класса `DbSet`. `DbSet` представляет собой список (набор) сущностей, хранящихся в базе данных. Через этот класс разработчик выполняет основные запросы в базе данных, что обеспечивает эффективное и безопасное взаимодействие.

В методе `OnModelCreating` мы прописываем инструкцию, которая по сути своей выполняется примерно ту же роль, что и аннотация данных. В данном случае мы указываем, что поле `Id` является счетчиком.

Далее необходимо подключить `DbContext` к проекту, это делается в файле `Program.cs`. Но перед этим немного разберем код, который представлен в этом файле.

```
0 references | You, 7 minutes ago | 1 author (You)
public class Program
{
    0 references
    public static void Main(string[] args)
    {

        var builder = WebApplication.CreateBuilder(args);
        // Add services to the container.
        builder.Services.AddControllersWithViews();

        var app = builder.Build();
    }
}
```

Приложение ASP.Net Core обычно начинается с инициализации переменной `builder`, который является экземпляром класса `WebApplicationBuilder`.

По сути своей `builder` выполняется ряд задач:

- Установка конфигурации приложения
- Добавление сервисов
- Настройка логгирования в приложении
- Установка окружения приложения
- Конфигурация объектов `IHostBuilder` и `IWebHostBuilder`, которые применяются для создания хоста приложения

`Builder.Services.AddContollersWithViews()` – прописывая эту строку, вы позволяете вашему приложению обрабатывать HTTP запросы, маршрутизировать их к соответствующим контроллерам и возвращать соответствующие представления в ответ на эти запросы.

`var app = builder.Build()` – в этой строке по сути создается экземпляр класса `IWebHost`, в это экземпляр добавляются все те настройки, которые были прописаны в `builder`.

```

19         // Configure the HTTP request pipeline.
20         if (!app.Environment.IsDevelopment())
21         {
22             app.UseExceptionHandler("/Home/Error");
23             // The default HSTS value is 30 days. You may want to change this for
24             app.UseHsts();
25         }
26
27         app.UseHttpsRedirection();
28         app.UseStaticFiles();
29
30         app.UseRouting();
31
32         app.UseAuthorization();
33
34         app.MapControllerRoute(
35             name: "default",
36             pattern: "{controller=Home}/{action=Index}/{id?}");
37
38         app.Run();
39     }
40 }
41 }
42

```

Далее у нас идет проверка, если приложение запущено в режиме разработки (Development), то оно использует адрес */Home/Error* для вывода ошибок. С помощью метода *UseHsts()* объекта *ApplicationBuilder* мы можем отправить браузеру заголовок Strict-Transport-Security, благодаря которому браузер всегда подключается по протоколу HTTPS.

Метод *UseHttpsRedirection* перенаправляет запросы HTTP на страницы HTTPS.

Компонент статических файлов (*UseStaticFiles*) возвращает статические файлы и прекращает последующую обработку запросов.

*UseRouting* – это метод в ASP.NET Core, который добавляет сопоставление маршрутов в конвейер промежуточного программного обеспечения. Этот метод просматривает набор конечных точек, определённых в приложении, и выбирает наилучшее соответствие на основе запроса. Перед вызовом *UseRouting* конечная точка всегда имеет значение null. Если найдено совпадение, конечная точка между *UseRouting* и *UseEndpoints* становится ненулевой.

Метод *UseAuthorization* подключает авторизацию в приложение.

Метод *MapControllerRoute()* используется для настройки маршрутизации запросов к контроллерам в приложении MVC. Этот метод позволяет определить шаблон маршрута для запросов, которые будут обрабатываться определенными контроллерами в вашем приложении.

Метод *Run* представляет собой один из способов настройки обработки запросов в вашем приложении. Этот метод используется для определения функции обработки запросов, которая будет вызываться, когда поступит HTTP запрос и не будет найдено ни одного совпадения с предыдущими обработчиками маршрутов.

После того, как мы разобрались с классом *Program*, продолжим работу над проектом. Добавим следующий код в ранее рассмотренный класс.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var connectionString = builder.Configuration["ConnectionStrings:DefaultConnection"];

builder.Services.AddDbContext<ToDoElementsContext>(options => options.UseNpgsql(connectionString));
```

Мы создаем строковую переменную, которая берет строку подключения к БД из нашего appsettings.json.

*builder.Services.AddDbContext* используется для регистрации контекста базы данных Entity Framework Core как службы зависимостей во время настройки конфигурации приложения.

*Services* - это коллекция служб, которая представляет собой контейнер для хранения и управления службами в приложении.

*AddDbContext<ToDoElementsContext>* - регистрирует контекст базы данных типа *ToDoElementsContext* в качестве службы в коллекции сервисов.

*Options* - это конфигурация для контекста базы данных, которая позволяет определить параметры, такие как строка подключения и настройки.

*UseNpgsql* - это метод, который определяет, что контекст базы данных будет использовать PostgreSQL в качестве провайдера.

Таким образом мы подключились к нашей базе данных через Entity Framework Core.

## Паттерн Repository

При работе с БД часто используется паттерн Repository (Репозиторий). Репозиторий представляет собой концепцию хранения коллекции для сущностей определенного типа.

Репозиторий позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой.

Допустим, у нас есть одно подключение к базе данных MS SQL Server. Однако, что, если в какой-то момент времени мы захотим сменить подключение с MS SQL на другое - например, к бд MySQL или MongoDB. При стандартном подходе даже в небольшом приложении, осуществляющем выборку, добавление, изменение и удаление данных, нам бы пришлось сделать большое количество изменений. Либо в процессе работы программы в зависимости от разных условий мы хотим использовать два разных подключения. Таким образом, репозиторий добавляет программе гибкость при работе с разными типами подключений. (более подробная информация по этому паттерну есть в различных источниках сети Интернет).

Но в данном проекте будет использоваться немного иной паттерн – Generic Repository (Обобщенного репозитория). Он имеет ряд преимуществ перед обычным Repository (Репозитория):

- Гибкость и повторное использование кода: Паттерн Generic Repository позволяет создать обобщенный интерфейс и реализацию для работы с различными типами сущностей. Это обеспечивает гибкость и повторное использование кода, так как один и тот же репозиторий может быть использован для доступа к данным разных типов объектов.

- Упрощение тестирования: Использование обобщенного репозитория делает тестирование кода более простым. Поскольку методы репозитория обобщены и не зависят

от конкретных типов данных, их легче тестировать с помощью мок-объектов или фейковых реализаций.

- Снижение дублирования кода: Generic Repository позволяет избежать дублирования кода для каждой сущности в вашем приложении. Вместо создания отдельного репозитория для каждой сущности, можно создать единый обобщенный репозиторий, который может обрабатывать запросы к данным для всех сущностей.

- Улучшенная абстракция слоя доступа к данным: Обобщенный репозиторий помогает абстрагировать слой доступа к данным от конкретной реализации хранилища данных (например, базы данных или памяти). Это обеспечивает более гибкую архитектуру и упрощает замену или обновление источника данных без изменения кода высокоуровневых компонентов приложения.

- Сокращение объема кода: Паттерн Generic Repository позволяет сократить количество кода, необходимого для доступа к данным, за счет использования обобщенных методов для выполнения основных операций CRUD (Create, Read, Update, Delete) над сущностями.

В целом, использование паттерна Generic Repository помогает улучшить структуру и гибкость приложения, сократить объем дублированного кода и упростить тестирование и обслуживание.

Основные компоненты Generic Repository:

- Интерфейс репозитория: Обобщенный интерфейс, который определяет базовые методы для доступа к данным, например, Add, Get, Update, Delete.

- Реализация репозитория: Класс, который реализует интерфейс репозитория и предоставляет конкретную реализацию методов доступа к данным для конкретного источника данных (например, базы данных).

Создайте в папке проекта директорию *Repositories*, в ней создайте еще 2 подпапки *Concrete* и *Interfaces*.

В папке *Interfaces* создайте интерфейс *IGenericRepository.cs* и *ITodoElementRepository.cs*.

В файле *IGenericRepository.cs* пропишите следующий код

```
ProgramEx > Repositories > Interfaces > IGenericRepository.cs > ...
You, 1 second ago | 1 author (You)
1 using System.Linq.Expressions;
2
3 namespace ProgramEx.Repositories.Interfaces
4 {
5     3 references | You, 1 second ago | 1 author (You)
6     public interface IGenericRepository<T> where T : class
7     {
8         1 reference
9         T GetById(int id);
10        1 reference
11        List<T> GetAll();
12        1 reference
13        IEnumerable<T> Find(Expression<Func<T, bool>> expression);
14        1 reference
15        void Add(T entity);
16        1 reference
17        void AddRange(IEnumerable<T> entities);
18        1 reference
19        void Remove(T entity);
20        1 reference
21        void RemoveRange(IEnumerable<T> entities);
22        1 reference
23        void Update(T entity);
24        1 reference
25        void SaveChanges();
26    }
27 }
```

В созданном интерфейсе *IToDoElementsRepository* укажите, что он наследуется от интерфейса *IGenericRepository*, при этом пропишите, что вместо *T* будет использоваться класс *ToDoElement*.

```
ProgramEx > Repositories > Interfaces > IToDoElementRepository.cs > ...
You, 7 days ago | 1 author (You)
1 using ProgramEx.Models;
2
3 namespace ProgramEx.Repositories.Interfaces
4 {
5     2 references | You, 7 days ago | 1 author (You)
6     public interface IToDoElementRepository : IGenericRepository<ToDoElement>
7     {
8     }
9 }
```

Перейдем в папку *Concrete* и создадим там 2 класса: *GenericRepository* и *ToDoElementRepository*.

В классе *GenericRepository* укажем, что он наследуется от одноименного интерфейса, также пропишем реализацию методов, которые мы указали в интерфейсе.

```
You, 7 days ago | 1 author (You)
1 using System.Linq.Expressions;
2 using ProgramEx.Repositories.Interfaces;
3 using ProgramEx.Data;
4
5 namespace ProgramEx.Repositories.Concrete
6 {
7     2 references | You, 7 days ago | 1 author (You)
8     public class GenericRepository<T>(ToDoElementsContext _context, IConfiguration _configuration) : IGenericRepository<T> where T : class
9     {
10         9 references
11         protected readonly ToDoElementsContext Context = _context;
12
13         0 references
14         protected readonly IConfiguration configuration = _configuration;
15
16         1 reference
17         public void Add(T entity)
18         {
19             Context.Set<T>().Add(entity);
20         }
21
22         1 reference
23         public void AddRange(IEnumerable<T> entities)
24         {
25             Context.Set<T>().AddRange(entities);
26         }
27
28         1 reference
29         public IEnumerable<T> Find(Expression<Func<T, bool>> expression)
30         {
31             return Context.Set<T>().Where(expression);
32         }
33     }
```

```

27 |         1 reference
    |         public List<T> GetAll()
28 |         {
29 |             return [... Context.Set<T>()];
30 |         }
31 |
32 |         1 reference
    |         public T GetById(int id)
33 |         {
34 |             return Context.Set<T>().Find(id);
35 |         }
36 |         1 reference
    |         public void Remove(T entity)
37 |         {
38 |             Context.Set<T>().Remove(entity);
39 |         }
40 |
41 |         1 reference
    |         public void RemoveRange(IEnumerable<T> entities)
42 |         {
43 |             Context.Set<T>().RemoveRange(entities);
44 |         }
45 |
46 |         1 reference
    |         public void Update(T entity)
47 |         {
48 |             Context.Set<T>().Update(entity);
49 |         }
50 |
51 |         1 reference
    |         public void SaveChanges()
52 |         {
53 |             Context.SaveChanges();
54 |         }
55 |     }
56 | }

```

You, 7 days ago • The stage of implementation of the Generi

Вот тут и раскрывается основной плюс данного паттерна. Мы создаем методы для работы с данными лишь 1 раз, а далее, путем наследования, можем применять эти методы для всех классов.

В ранее созданном классе *ToDoElementRepository* пропишем следующий код.

```

ProgramEx > Repositories > Concrete > ToDoElementRepository.cs > ...
You, 1 second ago | 1 author (You)
1  using ProgramEx.Data;
2  using ProgramEx.Models;
3  using ProgramEx.Repositories.Interfaces;
4
5  namespace ProgramEx.Repositories.Concrete
6  {
7      1 reference | You, 1 second ago | 1 author (You)
    |         public class ToDoElementRepository(ToDoElementsContext _context, IConfiguration _configuration) :
8      |         GenericRepository<ToDoElement>(_context, _configuration), IToDoElementRepository
9      |         {
10 |         }
11 |     }
12 | }

```

You, 7 days ago • The stage of implementation of the Generic Repo...

Мы прописали, что этот класс наследуется от *GenericRepository*, а также от интерфейса *IToDoElementRepository*.

Для того, чтобы репозитории наши функционировали, мы их добавим зависимость в наше приложение. Перейдите к класс *Program*, и пропишите следующий код после того, как мы добавили *DbContext*.

```

21         builder.Services.AddDbContext<ToDoElementsContext>(options => options.UseNpgsql(connectionString));
22
23
24         #region Repositories
25         builder.Services.AddScoped(typeof(IGenericRepository<>), typeof(GenericRepository<>));
26         builder.Services.AddScoped<IToDoElementRepository, ToDoElementRepository>();
27         #endregion
28
29         var app = builder.Build();
30
31         // Configure the HTTP request pipeline.
32         if (!app.Environment.IsDevelopment())
33         {
34             app.UseExceptionHandler("/Home/Error");
35             // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
36             app.UseHsts();
37         }

```

Немного о зависимостях. 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.

Используя различные методы внедрения зависимостей, можно управлять жизненным циклом создаваемых сервисов. Сервисы, которые создаются механизмом Dependency Injection, могут представлять один из следующих типов:

- Transient: при каждом обращении к сервису создается новый объект сервиса. В течение одного запроса может быть несколько обращений к сервису, соответственно при каждом обращении будет создаваться новый объект. Подобная модель жизненного цикла наиболее подходит для легковесных сервисов, которые не хранят данных о состоянии
- Scoped: для каждого запроса создается свой объект сервиса. То есть если в течение одного запроса есть несколько обращений к одному сервису, то при всех этих обращениях будет использоваться один и тот же объект сервиса.
- Singleton: объект сервиса создается при первом обращении к нему, все последующие запросы используют один и тот же ранее созданный объект сервиса

Для создания каждого типа сервиса предназначен соответствующий метод AddTransient(), AddScoped() и AddSingleton().

Таким образом, мы реализовали паттерн Generic Repository в нашем проекте, далее будем работать уже непосредственно с контроллерами и представлениями.

## Контроллеры и представления

Первым делом поработаем с представлениями. Все представления имеют расширение *cshtml*. Файл *cshtml* - это динамическая веб-страница, написанная в синтаксисе Razor, используемом ASP.NET динамическая платформа веб-разработки. Файл, по сути, представляет собой шаблон представления (layout) в архитектуре model-view-controller. Синтаксис Razor основан на C# и языке программирования Visual Basic.

Перейдем в папку *Views*. В этой папке уже есть некоторая подструктура. Во-первых, как правило, для каждого контроллера в проекте создается подкаталог в папке *Views*, который называется по имени контроллера и который хранит представления, используемые методами данного контроллера. Так, по умолчанию имеется контроллер *HomeController* и для него в папке *Views* есть подкаталог *Home* с представлениями для методов контроллера *HomeController*.

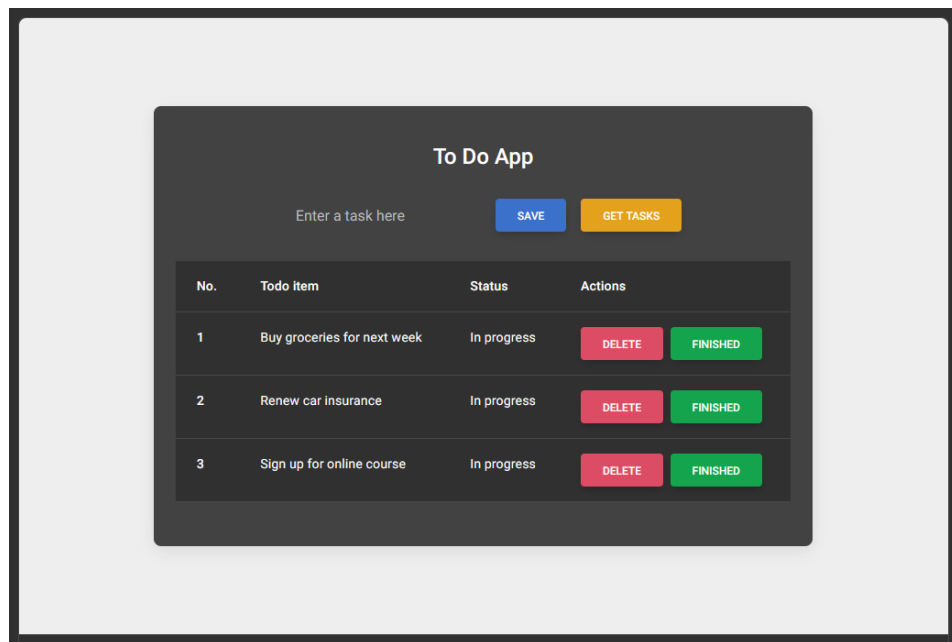
Также здесь есть папка *Shared*, которая хранит общие представления для всех контроллеров. По умолчанию это файлы *\_ViewImports.cshtml* и *\_ViewStart.cshtml*. Эти файлы содержат код, который автоматически добавляется ко всем представлениям.



`_ViewImports.cshtml` устанавливает некоторые общие для всех представлений пространства имен, а `_ViewStart.cshtml` устанавливает общую мастер-страницу.

При необходимости мы можем добавлять в каталог *Views* какие-то свои представления, каталоги для представлений. И они необязательно должны быть связаны с контроллерами и их методами.

Для начала начнем с верстки. Так как верстка интересует нас в меньшей степени, то будет применяться bootstrap для более быстрого формирования внешнего вида сайта, плюс будет взят с Интернета шаблон To-Do List. Перейдем в файл *Home.cshtml* и вставим шаблон в него шаблон.



Шаблон можно использовать какой угодно, но чтобы там была только та информация, которую мы можем извлечь из базы.

Сначала возьмем тот блок кода HTML, в котором происходит вывод списка задач. Его мы перенесем в новый файл (*ToDoElementList.cshtml*), который будет создан в директории *Home*, этот файл будет являться *Partial View*, то есть он будет по сути своей частью представления *Index*. Отобразить в самом представлении его можно при помощи следующего кода:

```
<div id = "itemList">
|   <partial name="ToDoElementList"></partial>
| </div>
```

Этот код будет расположен в том месте, где располагался вывод списка задач в файле *Index.cshtml*. Далее мы пропишем вывод данных.

Ключевым моментом в определении интерфейса на страницах Razor Page является использование конструкций движка Razor. Благодаря Razor мы можем применять на странице выражения языка C#. Синтаксис Razor довольно прост - все его конструкции предваряются символом `@`, после которого происходит переход от кода HTML к коду C#. При генерации ответа клиенту Razor обрабатывает выражения языка C# и на их основе генерирует код HTML.

В нашем случае вывод данных происходит в таблице (через тэг *table*). В *tbody* пропишем следующее:

```
1  <table id = "itemList" class="table mb-4">
2  <thead>
3  <tr>
4  <th scope="col">Todo item</th>
5  <th scope="col">Completed</th>
6  <th scope="col">Actions</th>
7  </tr>
8  </thead>
9  <tbody>
10  @foreach (var todoElement in Model)
11  {
12  <tr>
13  <td>@todoElement.Text</td>
14  <td>
15  <input class="form-check-input" type="checkbox" checked="@todoElement.Completed" onClick="return false;">
16  </td>
17  <td>
18  @Html.ActionLink("Delete", "Delete", new { id = @todoElement.Id } , new { @class = "btn btn-danger"})
19  @Html.ActionLink("Finished", "Finish", new { id = @todoElement.Id } , new { @class = "btn btn-success ms-1"})
20  </td>
21  </tr>
22  }
23  </tbody>
24  </table>
25
```

То есть при помощи цикла, который обозначается как *@foreach* мы проходим по всем задачам из списка, затем выводим в каждой строке данные о задачах. Стоит заметить что мы применили *@Html.ActionLink* для вывода кнопок-действий (по сути своей они являются ссылками) с конкретной задачей. В качестве первого аргумента указываем отображаемый текст, в качестве второго – действие, в третьем аргументе мы присвоили *id* нашему элементу (он равен *id* самой задаче в БД), а в последнем аргументе мы указали стили.

Пока отставим в сторону представления, мы к ним еще вернемся и перейдем к контроллерам.

Перейдем к файлу *HomeController.cshtml*, расположенному в директории *Controllers*. Контроллер у нас выглядит следующим образом:

```

You, last week | 1 author (You)
1 using System.Diagnostics;
2 using Microsoft.AspNetCore.Mvc;
3 using ProgramEx.Models;
4
5 namespace ProgramEx.Controllers;
6
3 references | You, last week | 1 author (You)
7 public class HomeController : Controller
8 {
9     1 reference
10     private readonly ILogger<HomeController> _logger;
11
12     0 references
13     public HomeController(ILogger<HomeController> logger)
14     {
15         _logger = logger;
16
17     0 references
18     public IActionResult Index()
19     {
20         return View();
21
22     0 references
23     public IActionResult Privacy()
24     {
25         return View();
26
27     [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
28     0 references
29     public IActionResult Error()
30     {
31         return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
32     }

```

В данном случае мы видим, что контроллер имеет конструктор, через который посредством механизма *dependency injection* передается сервис *ILogger*, используемый для логгирования. Также контроллер определяет три метода - *Index*, *Privacy* и *Error*. Все 3 метода возвращают тип *IActionResult*. Этот тип возвращает готовый HTML документ. В ASP.Net Core представления берутся из папки представлений, которая имеет то же имя, что и сам контроллер (в нашем случае *Home*), причем имена самих файлов соответствуют названиям методов данного контроллера, то есть, если метод называется *Index*, то он и будет возвращать по умолчанию файл *Index.cshtml* (если таковой существует).

Первоначально изменим в конструкторе параметр, теперь будет не объект *ILogger*, а объект *IToDoElementRepository* для дальнейшей работы с данными таблицы:

```

1 reference | You, 1 second ago | 1 author (You)
public class HomeController : Controller
{
    1 reference
    private readonly IToDoElementRepository _iToDoElementRepository;

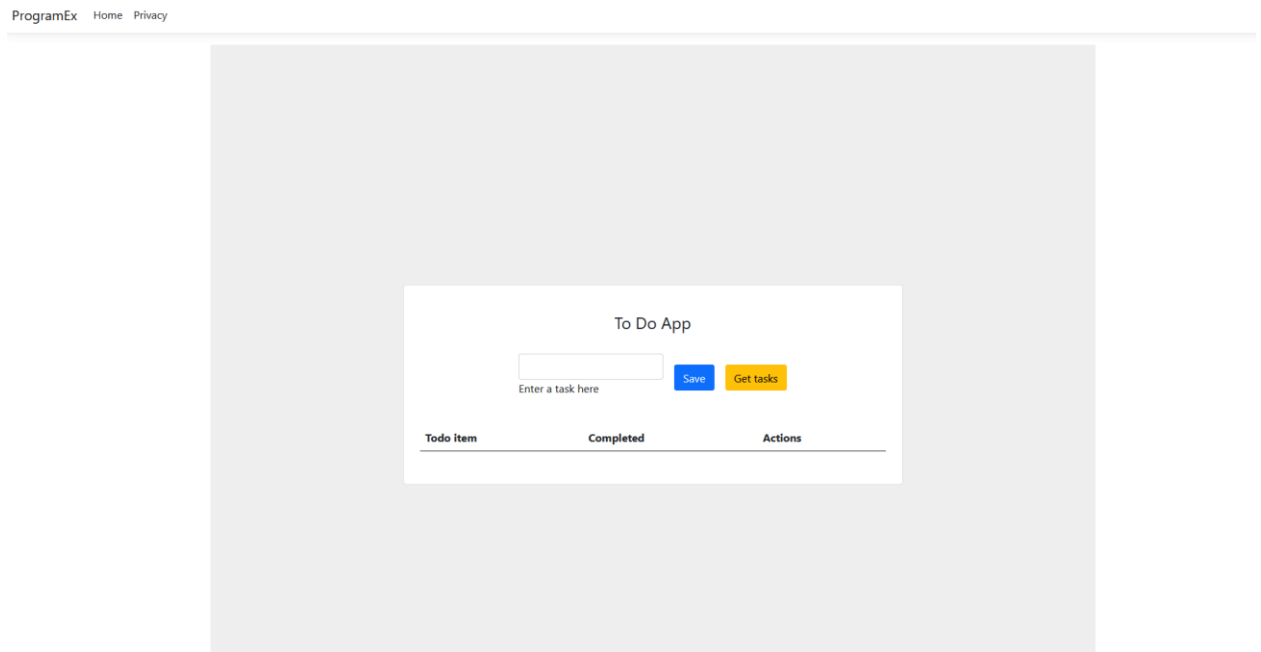
    0 references
    public HomeController(IToDoElementRepository context)
    {
        _iToDoElementRepository = context;
    }

```

Затем передадим дополнительный аргумент в методе *View()*, который возвращается в *Index*.

```
0 references
public IActionResult Index()
{
    return View(_iToDoElementRepository.GetAll().ToList());
}
```

То есть в само представление мы передали список элементов таблицы нашей БД. Теперь при запуске проекта мы можем перейти на страницу Index и увидеть следующее:



Если мы добавим записать в таблицу через pgAdmin, то у нас программа будет корректно выводить данные.

После метода *Index* пропишем еще 3 метода, на добавление, изменение и удалении записи:

```

[HttpGet, ActionName("Delete")]
0 references
public IActionResult DeleteElement(int id)
{
    _iToDoElementRepository.Remove(_iToDoElementRepository.GetById(id));
    _iToDoElementRepository.SaveChanges();
    return RedirectToAction("Index");
}

[HttpGet]
0 references
public IActionResult AddElement(string textElement)
{
    var newToDoElement = new ToDoElement(textElement);
    _iToDoElementRepository.Add(newToDoElement);
    _iToDoElementRepository.SaveChanges();
    return PartialView("ToDoElementList", _iToDoElementRepository.GetAll().ToList());
}

[HttpGet, ActionName("Finish")]
0 references
public IActionResult FinishElement(int id)
{
    var changedToDoElement = _iToDoElementRepository.GetById(id);
    changedToDoElement.Completed = true;
    _iToDoElementRepository.Update(changedToDoElement);
    _iToDoElementRepository.SaveChanges();
    return RedirectToAction("Index");
}

```

You: last week \* Add project files

Методы контроллера обычно работают в автоматическом режиме, то есть программа сама подставляет методы запросов, а также пути к этим методам (конечная точка пути при запросе будет такая же, как и название метода). Также можно явно прописать необходимые параметры (по принципу с *DataAnnotation*, который мы использовали ранее). В данном случае мы явно прописали методы, а также явно прописали конечные точки для двух методов (через свойство *ActionName*). Стоит отметить, что в методах на удаление и на изменение данных (*DeleteElement*, *FinishElement*) в качестве возвращаемого значения мы использовали метод *RedirectToAction*, благодаря данному методу переходит перенаправление на главную страницу (*Index*), но в методе *AddElement* мы применили метод *PartialView*, который возвращает значение как раз на часть представления (это создано для выполнения запроса без обновления страницы, позже мы это рассмотрим). Также, в качестве второго аргумента мы передали список элементов, которые будут выводиться.

С контроллером вопрос решен, продолжим работу с представлением.

В данном случае добавление элемента у нас происходит в форме:

```

<form
  class="row row-cols-lg-auto g-3 justify-content-center align-items-center mb-4 pb-2"
  id="addToDoElement" method="post" enctype="multipart/form-data">
  <div class="col-12">
    <div class="form-outline">
      <input type="text" id="textElement" name="textElement" class="form-control" />
      <label class="form-label" for="form1">Enter a task here</label>
    </div>
  </div>

  <div class="col-12">
    <input type="button" id="inputTextElement" class="btn btn-primary" value="Save" onclick="FormSubmit();" />
  </div>
</form>

```

Чего-либо сверхъестественного в этой форме нет, поэтому подробно рассматривать мы не будем его. Но можно заметить, что при нажатии на кнопку у нас срабатывает метод *FormSubmit*, он уже реализован через JavaScript.

Сама реализация прописана сразу после формы:

```
@section Scripts
{
    <script>
        var elText = document.getElementById("textElement");
        async function FormSubmit() {

            var url = "/Home/AddElement";
            let response = await fetch(url + "?textElement=" + elText.value);
            let data = await response.text();

            var itemList = document.getElementById("itemList");
            itemList.innerHTML = "";
            itemList.innerHTML = data;

            document.getElementById("textElement").value = "";
        }
    </script>
}
```

Через *@section Scripts* мы отделяем скрипты JS от html кода для более удобного чтения (хотя в данном случае это не столь важно). Важно то, что мы выполняем асинхронный запрос на сервер, через который вызывается метод *AddElement*, то есть идет добавление записи. Так как сервер нам возвращает html-страницу Partial View, то мы просто в заранее подготовленный блок div вносим весь переданный код, таким образом, выполняя асинхронный запрос и изменяя только часть представления мы добиваемся добавления записи без перезагрузки страницы.

По такой же аналогии реализуйте удаление и изменение записи.

## Финишная прямая

В данном руководстве вы познакомились с технологией ASP.Net Core MVC, научились запускать проект в контейнере Docker, а также управлять несколькими контейнерами (контейнер с проектом и с БД PostgreSQL) через docker-compose. Надеюсь, вы приобрели базовые знания паттернов, умение работать с Docker контейнерами и, непосредственно с самим ASP.Net Core. Эти знания могут Вам в дальнейшем пригодиться, ведь подобный принцип работы, который есть в ASP.Net Core, существует и в других языках программирования, только он представлен в другом виде.