**Лабораторная работа**

**«Разработка моделей и контроллеров** ASP.NET MVC **приложения баз данных»**

**Цель работы:** ознакомиться с возможностями ASP.NET Core MVC и Entity Framework Core для разработки слоя доступа к данным, хранящимся в базе данных, и обработки запросов пользователя посредством контроллеров.

[1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ](#sect1)

[2. ЗАДАНИЕ](#sect2)

[3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ПРИМЕРЫ](#sect3)

[4. ОФОРМЛЕНИЕ ОТЧЕТА](#sect4)

[5. КОНТРОЛЬНЫЕ ВОПРОСЫ](#sect5)

[6. ЛИТЕРАТУРА](#sect6)

**1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

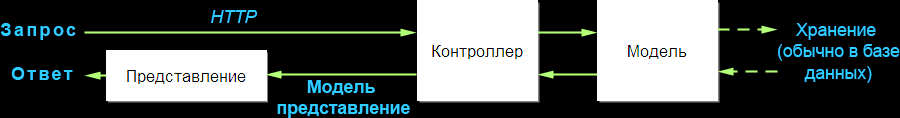
**ASP.NET MVC** – фреймворк для создания сайтов и веб-приложений с помощью реализации паттерна MVC.

Концепция паттерна (шаблона) MVC (model - view - controller) предполагает разделение приложения на три компонента:

**Контроллер (controller)** - класс, обеспечивающий связь между пользователем и системой, представлением и хранилищем данных. Он получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления.

**Представление (view)** - визуальная часть или пользовательский интерфейс приложения. Как правило, html-страница, которую пользователь видит, зайдя на сайт.

**Модель (model)** класс, описывающий логику используемых данных.



Каждый запрос, поступающий в приложение, обрабатывается контроллером. Контроллер может обрабатывать запрос произвольным образом до тех пор, пока он не пересекает границу ответственности модели и представления. Это означает, что контроллеры не должны содержать или сохранять данные, равно как не генерировать пользовательские интерфейсы.

* При использовании контроллеров существуют соглашения об именованииназвания контроллеров должны оканчиваться на суффикс "Controller", остальная же часть до этого префикса считается именем контроллера.
* Чтобы обратиться контроллеру из веб-браузера, нам надо в адресной строке набрать **адрес\_сайта/Имя\_контроллера/**.

Так, по запросу **адрес\_сайта/Home/** система маршрутизации по умолчанию вызовет метод **Index** контроллера **HomeController** для обработки входящего запроса.

* Если мы хотим отправить запрос к конкретному методу контроллера, то нужно указывать этот метод явно: **адрес\_сайта/Имя\_контроллера/Метод\_контроллера**,

*Например*, ***адрес\_сайта/Home/Buy*** - обращение к методу **Buy** контроллера **HomeController**.

Для создания проекта, применяющего MVC надо выбрать шаблон **Web Application (Model-View-Controller)**:

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

Оставим все настройки по умолчанию и нажмем на ОК. Visual Studio создаст новый проект MVC. Структура создаваемого проекта будет определяться рядом папок и файлов:

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

* **Dependencies**: все добавленные в проект пакеты и библиотеки
* **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

Если мы запустим проект на выполнение, 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:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

**Сервисы MVC**

Функциональность MVC и ее работа в приложении зависит от добавляемых сервисов. В примере выше мы использовали метод AddControllersWithViews() для добавления сервисов MVC, благодаря чему система маршрутизации смогла связать запрос с контроллером. Есть ряд опций по встраиванию сервисов, которые мы можем при необходимости использовать:

* AddMvc(): добавляет все сервисы фреймворка MVC (в том числе сервисы для работы с аутентификацией и авторизацией, валидацией и т.д.)
* AddMvcCore(): добавляет только основные сервисы фреймворка MVC, а всю дополнительную функциональность, типа аутентификацией и авторизацией, валидацией и т.д., необходимо добавлять самостоятельно
* AddControllersWithViews(): добавляет только те сервисы фреймворка MVC, которые позволяют использовать контроллеры и представления и связанную функциональность. При создании проекта по типу ASP.NET Core Web App (Model-View-Controller) используется именно этот метод
* AddControllers(): позволяет использовать контроллеры, но без представлений.

И в зависимости от того, насколько широко нам надо использовать возможности фреймворка MVC, выбирается соответствующий метод. Например, в примере выше мы могли бы использовать вместо вызова AddControllersWithViews() метод AddMvc().

1. **Конвейер обработки запроса и middleware**

Термин middleware обозначает небольшие компоненты приложения, которые могут быть внедрены в конвейер обработки HTTP-запроса. Каждый компонент может либо выполнить некоторую обработку запроса, либо передать его дальше в цепи конвейера обработки запроса.



Для создания конвейера обработки запроса используются делегаты запроса, которые потом участвуют в обработке запроса. Делегаты запроса конфигурируются с помощью методов расширений Run, Map и Use объекта IApplicationBuilder, Каждый делегат запроса может быть вынесен в отдельный класс, и такой класс как раз и будет представлять компонент middleware.

Каждый делегат в конвейере может выполнять обработку запроса как до, так и после следующего в конвейере делегата. Каждый делегат может либо передать запрос далее следующему в цепи делегату, либо выполнить обработку и закончить работу конвейера.

Кроме использования делегатов в методах Run/Use/Map мы можем создавать свои компоненты middleware в виде методов расширений к типу IApplicationBuilder.

Например, напишем свой компонент. Для этого добавим в новый проект новый класс TokenMiddleware:

|  |
| --- |
| using Microsoft.AspNetCore.Http;  using System.Threading.Tasks;    public class TokenMiddleware  {      private readonly RequestDelegate \_next;        public TokenMiddleware(RequestDelegate next)      {          this.\_next = next;      }        public async Task Invoke(HttpContext context)      {          var token = context.Request.Query["token"];          if (string.IsNullOrWhiteSpace(token) || token!="12345678")          {              context.Response.StatusCode = 403;              await context.Response.WriteAsync("Token is invalid");          }          else          {              await \_next.Invoke(context);          }      }  } |

Данный класс устанавливает в конструкторе ссылку на тот делегат запроса, который стоит следующим в конвейере обработки запроса и который представляет объект RequestDelegate. И также в классе определен метод Invoke(), который будет вызываться при обработке запроса.

Суть действия класса заключается в том, что мы получаем из запроса параметр "token". Если полученный токен равен строке "12345678", то передаем запрос дальше следующему компоненту, вызвав метод \_next.Invoke(). Иначе возвращаем пользователю сообщение об ошибке.

И также добавим класс, который назовем TokenExtensions:

|  |
| --- |
| using Microsoft.AspNetCore.Builder;  public static class TokenExtensions  {      public static IApplicationBuilder UseToken(this IApplicationBuilder builder)      {          return builder.UseMiddleware<TokenMiddleware>();      }  } |

Здесь создается метод расширения для типа IApplicationBuilder. С помощью метода UseMiddleware<T> в конструктор объекта TokenMiddleware будет внедряться объект для параметра RequestDelegate next. Поэтому явным образом передавать значение для этого параметра нам не нужно.

Теперь применим этот класс в классe Program:

var builder = WebApplication.CreateBuilder();

builder.Services.AddTransient<TimeService>();

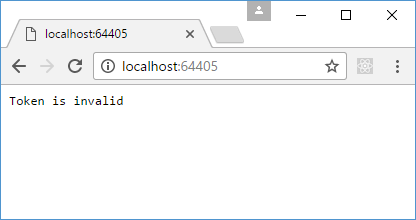
var app = builder.Build();

app.UseToken();

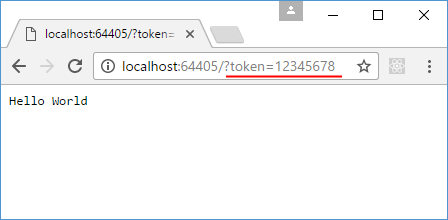
app.Run(async (context) => await context.Response.WriteAsync("Hello World"));

app.Run();

Если мы не передадим параметр token или передадим для него значение, отличное от "12345678", то браузер отобразит ошибку:



Если же будет передан корректный токен, то метод app.UseToken() передаст управление методу app.Run():



1. **Передача параметров**

Изменим класс TokenMiddleware, чтобы он извне получал образец токена для сравнения:

|  |
| --- |
| using Microsoft.AspNetCore.Http;  using System.Threading.Tasks;    public class TokenMiddleware  {      private readonly RequestDelegate \_next;      string pattern;      public TokenMiddleware(RequestDelegate next, string pattern)      {          this.\_next = next;          this.pattern = pattern;      }        public async Task Invoke(HttpContext context)      {          var token = context.Request.Query["token"];          if (string.IsNullOrWhiteSpace(token) || token!=pattern)          {              context.Response.StatusCode = 403;              await context.Response.WriteAsync("Token is invalid");          }          else          {              await \_next.Invoke(context);          }      }  } |

Образец токена, с которым идет сравнения, устанавливается через конструктор. Чтобы передать его в конструктор, изменим класс TokenExtensions:

|  |
| --- |
| using Microsoft.AspNetCore.Builder;  public static class TokenExtensions  {      public static IApplicationBuilder UseToken(this IApplicationBuilder builder, string pattern)      {          return builder.UseMiddleware<TokenMiddleware>(pattern);      }  } |

В метод builder.UseMiddleware можно передать набор значений, которые передаются в конструктор компонента middleware.

Затем в классe Program Configure() уже можно передать в метод расширения UseToken конкретное значение:

|  |
| --- |
| app.UseToken("555555");        app.Run(async (context) =>      {          await context.Response.WriteAsync("Hello World");      }); |

1. **Контроллеры**
2. **Контроллеры и их действия**

Центральным звеном в архитектуре ASP.NET Core MVC является контроллер. При получении запроса система маршрутизации выбирает для обработки запроса нужный контроллер и передает ему данные запроса. Контроллер обрабатывает эти данные и посылает обратно результат обработки.

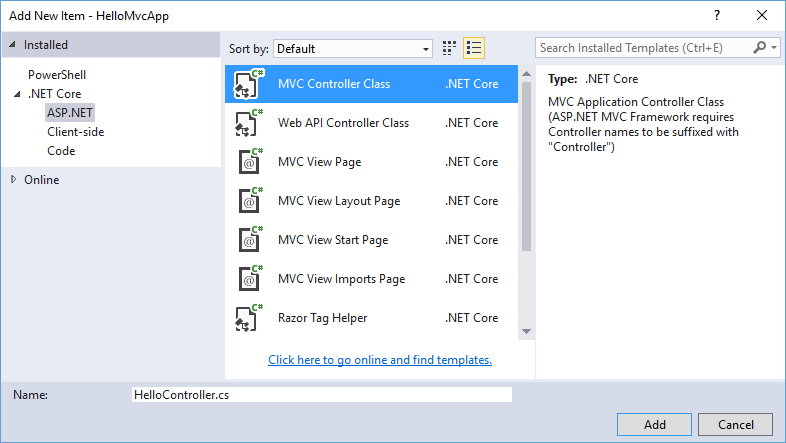
В ASP.NET Core MVC контроллер представляет обычный класс на языке C#, который наследуется от абстрактного базового класса Microsoft.AspNetCore.Mvc.Controller.

По умолчанию проект ASP.NET Core MVC содержит как минимум один контроллер - HomeController:

|  |
| --- |
| public class HomeController : Controller  {      public IActionResult Index()      {          return View();      }        public IActionResult About()      {          ViewData["Message"] = "Your application description page.";            return View();      }        public IActionResult Contact()      {          ViewData["Message"] = "Your contact page.";            return View();      }        public IActionResult Error()      {          return View();      }  } |

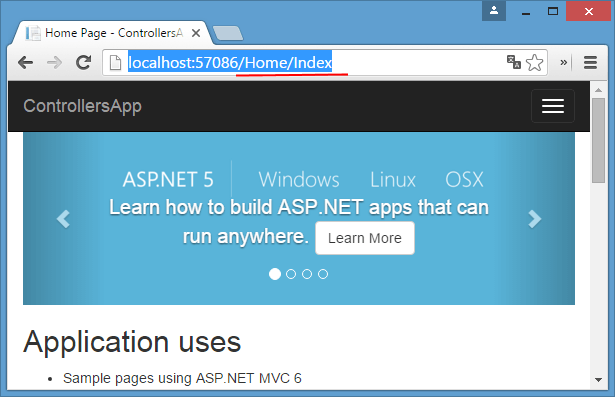
При использовании контроллеров существуют некоторые условности. Во-первых, в проекте контроллеры помещаются в каталог *Controllers*. И во-вторых, по соглашениям об именовании названия контроллеров должны оканчиваться на суффикс "Controller", остальная же часть до этого суффикса считается именем контроллера, например, HomeController.

Если нам нужен еще один контроллер, то мы можем добавить папку *Controllers* новый класс, который будет наследоваться от класса Controller. Либо мы можем использовать готовый шаблон MVC Controller Class:



Контроллер, как и любой класс на языке C#, может иметь поля, свойства, методы. По умолчанию HomeController имеет четыре метода, которые можно назвать действиями. Действия контроллера - это публичные методы, которые могут сопоставляться с запросами. Например, стандартный контроллер содержит четыре метода, все они публичные и поэтому могут использоваться для обработки запроса.

Чтобы обратиться контроллеру из веб-браузера, нам надо в адресной строке набрать адрес\_сайта/Имя\_контроллера/Действие\_контроллера. Так, по запросу адрес\_сайта/Home/Index система маршрутизации по умолчанию вызовет метод Index контроллера HomeController для обработки входящего запроса. Например:



Однако такое сопоставление строки url с названием контроллера и его метода происходит благодаря системе маршрутизации. Если мы обратимся к классу Program, то мы можем найти там определение единственного для приложения маршрута:

app.MapControllerRoute(

    name: "default",

    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

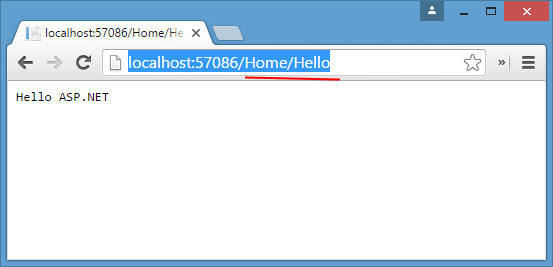
Здесь метод MapControllerRoute добавляет один маршрут с именем default и шаблоном "{controller=Home}/{action=Index}/{id?}". Данный шаблон устанавливает трехсегментную структуру строки запроса: *controller/action/id*. То есть в начале идет название контроллера, затем название действия, и далее может идти необязательный параметр id.

Собственно поэтому система может соотнести запрос типа *localhost:xxxx/Home/Index* с контроллером и его действием.

Однако не все методы контроллера являются действиями. Контроллер также может иметь непубличные методы - такие методы не рассматриваются как действия и соответственно не могут соотноситься с запросами. Например, определим в контроллере следующий метод:

|  |
| --- |
| protected internal string Hello()  {      return "Hello ASP.NET";  } |

Поскольку его модификатор отличается от public, то мы не сможем обратиться к этому методу с запросом *localhost:xxxx/Home/Hello*. Хотя такие не публичные методы также могут быть полезными - в них можно определять какие-нибудь промежуточные вычисления и затем использовать в действиях контроллера. При этом если мы изменим модификатор метода на public, то метод Hello станет полноценным действием:



1. **Атрибуты NonController, ActionName и NonAction**

Возможно, сопоставление по умолчанию бывает не всегда удобно. Например, у нас есть класс в папке Controllers, но мы не хотим, чтобы он мог обрабатывать запрос и использоваться как контроллер. Чтобы указать, что этот класс не является контроллером, нам надо использовать над ним атрибут [NonController]:

|  |
| --- |
| [NonController]  public class HomeController : Controller  {      //...........  } |

Аналогично, если мы хотим, чтобы какой-либо публичный метод контроллера не рассматривался как действие, то мы можем использовать над ним атрибут NonAction:

|  |
| --- |
| [NonAction]  public string Hello()  {      return "Hello ASP.NET";  } |

Атрибут [ActionName] позволяет для метода задать другое имя действия. Например:

|  |
| --- |
| [ActionName("Welcome")]  public string Hello()  {      return "Hello ASP.NET";  } |

В этом случае чтобы обратиться к этому методу, надо отправить запрос *localhost:xxxx/Home/Welcome*. А запрос *localhost:xxxx/Home/Hello* работать не будет.

1. **Типы запросов**

Выше для обработки запросов использовалась следующая пара методов:

|  |
| --- |
| [HttpGet]  public IActionResult Buy(int id)  {      Order order = new Order { PhoneId = id };      return View(order);  }  [HttpPost]  public string Buy(Order order)  {      db.Orders.Add(order);      db.SaveChanges();      return "Спасибо, " + order.User + ", за покупку!";  } |

Несмотря на то, что здесь два разных метода, но они в соответствии с именем образуют одно действие Buy. Допустимо определять в контроллере методы с одним и тем же именем, только в этом случае они должны различаться по параметрам, как в данном случае.

Кроме того, методы в рамках одного действия могут обслуживать разные запросы. Для указания типа запроса HTTP нам надо применить к методу один из атрибутов:

[HttpGet], [HttpPost], [HttpPut], [HttpDelete] и [HttpHead].

Если атрибут явным образом не указан, то считается, что метод предназначен для обработки GET-запросов.

1. **Передача данных в контроллер**

Вместе с запросом приложению могут приходить различные данные. И чтобы получить эти данные, мы можем использовать разные способы. Самым распространенным способом считается применение параметров.

Определение в методах контроллера параметров ничем не отличается от определения параметров в языке C#. Параметры могут представлять примитивные типы, как int или string, а могут представлять и более сложные классы.

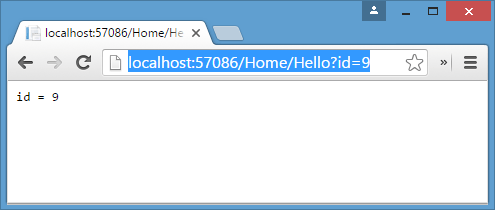
Передавать значения для параметров можно различными способами. При отправке GET-запроса значения передаются через строку запроса. Стандартный get-запрос принимает примерно следующую форму:

*название\_ресурса?параметр1=значение1&параметр2=значение2*.

Например, определим следующий метод:

|  |
| --- |
| public string Hello(int id)  {      return $"id= {id}";  } |

Чтобы передать значение для параметра id, нам надо отправить запрос типа *http://localhost:57086/Home/Hello?id=9*:



Если метод принимает несколько параметров, то:

|  |
| --- |
| public string Square(int a, int h)  {      double s = a \* h / 2;      return $"Площадь треугольника с основанием {a} и высотой {h} равна {s}";  } |

В этом случае мы можем обратиться к действию, набрав в адресной строке *Home/Square?a=10&h=3*, и приложение выдало бы нам нужный результат.

Если же мы не используем параметры в строке запроса, то для параметров будут передаваться значения по умолчанию. Например, при отправке запроса *Home/Square/* параметры a и h будут равны 0. Но на случай подобной ситуации мы можем использовать параметры по умолчанию, которые будут работать, если через строку запроса не передается никаких параметров:

|  |
| --- |
| public string Square(int a = 3, int h = 10)  {      double s = a \* h / 2;      return $"Площадь треугольника с основанием {a} и высотой {h} равна {s}";  } |

Система привязки MVC, сопоставляет параметры запроса и параметры метода по имени. То есть, если в строке запроса идет параметр a, то его значение будет передаваться именно параметру метода, который также называется a. При этом должно быть также соответствие по типу, то есть если параметр метода принимает числовое значение, то и через строку запроса надо передавать для этого параметра, а не строку.

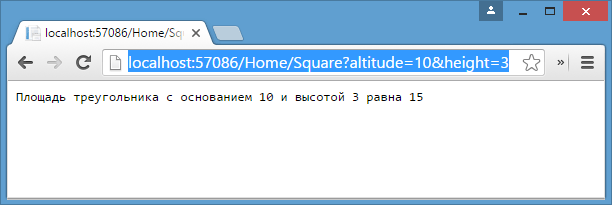
1. **Передача сложных объектов**

Хотя строка запроса преимущественно используется для передачи данных примитивных типов, но мы также можем принимать более сложные объекты. Например, определим рядом с контроллером класс Geometry:

|  |
| --- |
| public class HomeController : Controller  {      public string Square(Geometry geometry)      {          return $"Площадь треугольника с основанием {geometry.Altitude} и высотой {geometry.Height} равна {geometry.GetSquare()}";      }        // остальное содержимое  }  public class Geometry  {      public int Altitude { get; set; } // основание      public int Height { get; set; } // высота        public double GetSquare() // вычисление площади треугольника      {          return Altitude \* Height / 2;      }  } |

Класс Geometry определяет два свойства и метод для подсчета площади. И теперь в контроллере метод Square принимает параметр типа Geometry. Как в этом случае мы можем передать контроллеру данные? Для этого нам надо отправить запрос наподобие следующего *http://localhost:57086/Home/Square?altitude=10&height=3*.

Здесь параметры строки запроса должны соответствовать по имени свойствам объекта. Регистр названий при этом не учитывается:



1. **Передача массивов**

Допустим, метод принимает массив чисел:

|  |
| --- |
| public string Sum(int[] nums)  {      return $"Сумма чисел равна {nums.Sum()}";  } |

Чтобы передать значения для массива, нам надо использовалась строку запроса наподобие *http://localhost:57086/Home/Sum?nums=1&nums=2&nums=3*. В этом случае в массиве nums окажется три элемента.

Теперь изменим метод Sum, чтобы он принимал массив объектов ранее созданного класса Geometry:

|  |
| --- |
| public string Sum(Geometry[] geoms)  {      return $"Сумма площадей равна {geoms.Sum(g=>g.GetSquare())}";  } |

Данный метод подсчитывает сумму всех площадей в массиве geoms. И чтобы передать в этот метод данные, нам надо использовать запрос типа *http://localhost:57086/Home/Sum?geoms[0].altitude=10&geoms[0].height=3&geoms[1].altitude=16&geoms[1].height=2*.

В этом случае в массиве geoms будут два элемента Geometry.

1. **Передача данных в запросе POST**

Кроме GET-запросов также широко применяются POST-запросы. Как правило, такие запросы отправляются с помощью форм на веб-странице. Но основные принципы передачи данных будут теми же, что и в GET-запросах.

Для передачи POST-запросов определим в представлении Index.cshtml, которое должно быть по умолчанию в папке Views/Home, простенькую форму:

|  |
| --- |
| <form method="post" action="~/Home/Square">      <label>Высота:</label><br />      <input type="number" name="height" /><br />      <label>Основание:</label><br />      <input type="number" name="altitude" /><br />      <input type="submit" value="Отправить" />  </form> |

Форма устанавливает метод отправки - post, адрес отправки - Home/Square и два поля ввода чисел.

И чтобы метод Square принимал отправляемые данные, нам надо изменить его код следующим образом:

|  |
| --- |
| [HttpPost]  public string Square(int altitude, int height)  {      double square = altitude \* height / 2;      return $"Площадь треугольника с основанием {altitude} и высотой {height} равна {square}";  } |

Чтобы система могла связать параметры метода и данные формы, необходимо, чтобы атрибуты name у полей формы соответствовали названиям параметров.

Причем здесь действуют те же правила привязки, поэтому мы также может с той же формы получать более сложные объекты:

|  |
| --- |
| [HttpPost]  public string Square(Geometry geometry)  {      return $"Площадь треугольника с основанием {geometry.Altitude} и высотой {geometry.Height} равна {geometry.GetSquare()}";  } |

1. **Получение данных из контекста запроса**

Параметры представляют самый простой способ получения данных, но в действительности нам необязательно их использовать. В контроллере доступен объект Request, у которого можно получить как данные строки запроса, так и данные отправленных форм.

Данные строки запроса доступны через свойство Request.Query, которое представляет объект IReadableStringCollection. Например:

|  |
| --- |
| public string Square()  {      string altitudeString = Request.Query.FirstOrDefault(p => p.Key == "altitude").Value;      int altitude = Int32.Parse(altitudeString);        string heightString = Request.Query.FirstOrDefault(p => p.Key == "height").Value;      int height = Int32.Parse(heightString);        double square = altitude \* height / 2;      return $"Площадь треугольника с основанием {altitude} и высотой {height} равна {square}";  } |

В данном случае метод Square обрабатывает GET-запросы, и мы можем к нему обратиться через запрос типа *http://localhost:57086/Home/Square?altitude=20&height=4*.

Для получения данных отправленных форм можно использовать свойство Request.Form. Это свойство представляет объект IFormsCollection, но работает аналогично Request.Query:

|  |
| --- |
| [HttpPost]  public string Square()  {      string altitudeString = Request.Form.FirstOrDefault(p => p.Key == "altitude").Value;      int altitude = Int32.Parse(altitudeString);        string heightString = Request.Form.FirstOrDefault(p =>p.Key == "height").Value;      int height = Int32.Parse(heightString);        double square = altitude \* height / 2;      return $"Площадь треугольника с основанием {altitude} и высотой {height} равна {square}";  } |

1. **Передача зависимостей в контроллер**

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.

Нередко для установки зависимостей в подобных системах используются специальные контейнеры - IoC-контейнеры (Inversion of Control). Такие контейнеры служат своего рода фабриками, которые устанавливают зависимости между абстракциями и конкретными объектами и, как правило, управляют созданием этих объектов.

ASP.NET Core имеет встроенный контейнер внедрения зависимостей, который представлен интерфейсом IServiceProvider. А сами зависимости еще называются сервисами, собственно поэтому контейнер можно назвать провайдером сервисов. Этот контейнер отвечает за сопоставление зависимостей с конкретными типами и за внедрение зависимостей в различные объекты.

Как и любой класс, контроллер может получать сервисы приложения через механизм dependency injection. В контроллере это можно делать следующими способами:

* Через конструктор
* Через параметр метода, к которому применяется атрибут FromServices
* Через свойство HttpContext.RequestServices

Например, пусть в проекте определен интерфейс ITimeService и его реализация SimpleTimeService:

|  |
| --- |
| public interface ITimeService  {      string Time { get; }  }  public class SimpleTimeService : ITimeService  {      public SimpleTimeService()      {          Time = DateTime.Now.ToString("hh:mm:ss");      }      public string Time { get; }  } |

И в классе Startup происходит регистрация сервиса ITimeService:

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {      services.AddTransient<ITimeService, SimpleTimeService>();      services.AddMvc();  } |

1. **Передача через конструктор**

Когда приходит запрос к контроллеру, инфраструктура MVC вызывает провайдер сервисов для создания объекта HomeController. Провайдер сервисов проверят конструктор класса HomeController на наличие зависимостей. Затем создает объекты для всех используемых зависимостей и передает их в конструкторо HomeController для создания объекта контроллера, который затем обрабатывает запрос.

Например, получим зависимость в конструкторе контроллера:

|  |
| --- |
| public class HomeController : Controller  {      private readonly ITimeService \_timeService;        public HomeController(ITimeService timeServ)      {          \_timeService = timeServ;      }      public string Index()      {          return \_timeService.Time;      }  } |

В данном случае процесс установки зависимостей будет выглядеть следующим образом:

1. Приложение получает запрос к методу контроллера HomeController
2. Фреймворк MVC обращается к провайдеру сервисов для создания объекта контроллера HomeController
3. Провайдер сервисов смотрит на конструктор класса HomeController и видит, что там имеется зависимость от интерфейса ITimeService
4. Провайдер сервисов среди зарегистрированных зависимостей ищет класс, который представляет реализацию интерфейса ITimeService
5. Если нужная зависимость найдена, то провайдер сервисов создает объект класса, который реализует интерфейс ITimeService
6. Затем провайдер сервисов создает объект HomeController, передавая в его конструктор ранее созданную реализацию ITimeService
7. В конце провайдер сервисов возвращает созданный объект HomeController инфраструктуре MVC, которая использует контроллер для обработки запроса
8. **Передача зависимостей в методы. FromServices**

Иногда зависимость используется только в одном методе. И в этом случае нет необходимости передавать ее в контроллер, поскольку она напрямую может быть внедрена в сам метод, который ее использует. Для передачи зависимости в метод применяется атрибут [FromServices]:

|  |
| --- |
| public string Index([FromServices] ITimeService timeService)  {      return timeService.Time;  } |

Атрибут FromServices предоставляется инфраструктурой MVC - он определен в пространстве имен Microsoft.AspNetCore.Mvc.

1. **HttpContext.RequestServices**

В методах контроллера можно обращаться к объекту контекста запроса через свойство HttpContext, а через свойство HttpContext.RequestServices можно получить все зарегистрированные в приложении сервисы:

|  |
| --- |
| using Microsoft.Extensions.DependencyInjection;  //........................................    public string Index()  {      ITimeService timeService = HttpContext.RequestServices.GetService<ITimeService>();      return timeService?.Time;  } |

1. **Представления**

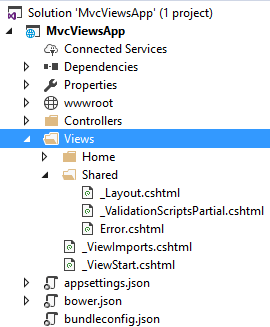
В большинстве случаев при обращении к веб-приложению пользователь ожидает получить веб-страницу с какими-нибудь данными. В MVC для этого, как правило, используются представления, которые и формируют внешний вид приложения. В ASP.NET MVC Core представления - это файлы с расширением cshtml, которые содержат код пользовательского интерфейса в основном на языке html, а также конструкции Razor - специального движка представлений, который позволяет переходить от кода html к коду на языке C#.

Например, возьмем простейшее представление:

|  |
| --- |
| @{      Layout = null;  }  <!doctype html>  <html>  <head>      <title>Hello ASP.NET</title>      <meta charset="utf-8" />  </head>  <body>      <h2>Привет ASP.NET Core!</h2>  </body>  </html> |

Данное представление напоминает обычную страницу html. Здесь могут быть определены все стандартные элементы разметки html, здесь могут подключаться стили, скрипты. Но полноценной html-страницей представление все равно не является, потому что во время выполнения эти представления компилируются в сборки и уже затем используются для генерации html-страниц, которые видит пользователь в своем браузере.

Для хранения представлений в проекте ASP.NET MVC предназначена папка Views:

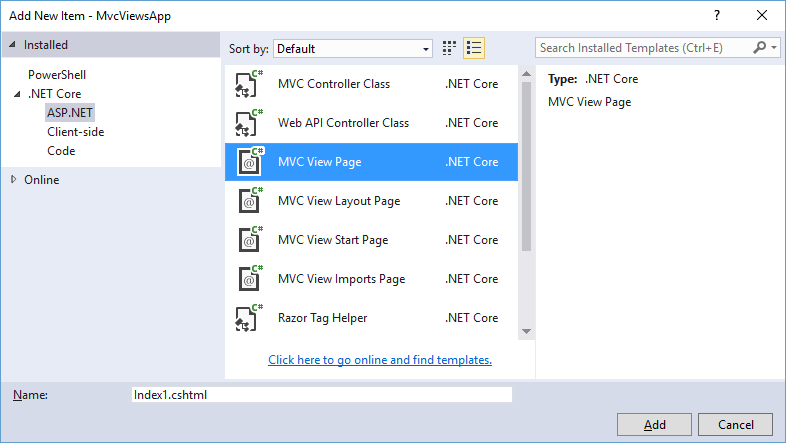


В этой папке уже есть некоторая подструктура. Во-первых, как правило, для каждого контроллера в проекте создается подкаталог в папке Views, который называется по имени контроллера и который хранит представления, используемые методами данного контроллера. Так, по умолчанию имеется контроллер HomeController 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 \_Layout.cshtml (используется в качестве мастер-страницы), Error.cshtml (использутся для отображения ошибок) и \_ValidationScripsPartial.cshtml (частичное представление, которое подключает скрипты валидации формы).

И в корне каталога Views также можно найти два файла \_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 добавления представления нужно правой кнопкой мыши на подкаталог в папке Views (или на саму папку Views) и в контекстном меню выбрать Add -> New Item. Затем в появившемся окне добавления нового элемента выбрать компонент MVC View Page:



1. **ViewResult**

За работу с представлениями отвечает объект ViewResult. Он производит рендеринг представления в веб-страницу и возвращает ее в виде ответа клиенту.

Чтобы возвратить объект ViewResult используется метод View:

|  |
| --- |
| public class HomeController : Controller  {      public IActionResult Index()      {          return View();      }  } |

Вызов метода View возвращает объект ViewResult. Затем уже ViewResult производит рендеринг определенного представления в ответ. По умолчанию контроллер производит поиск представления в проекте по следующим путям:

/Views/Имя\_контроллера/Имя\_представления.cshtml

/Views/Shared/Имя\_представления.cshtml

Согласно настройкам по умолчанию, если название представления не указано явным образом, то в качестве представления будет использоваться то, имя которого совпадает с именем действия контроллера. Например, вышеопределенное действие Index по умолчанию будет производить поиск представления Index.cshtml в папке */Views/Home/*.

Метод View() имеет четыре перегруженных версии:

* View(): для генерации ответа используется представление, которое по имени совпадает с вызывающим методом
* View(string viewName): в метод передается имя представления, что позволяет переопределить используемое по умолчанию представление
* View(object model): передает в представление данные в виде объекта model
* View(string viewName, object model): переопределяет имя представления и передает в него данные в виде объекта model

Вторая версия метода позволяет переопределить используемое представление. Если представление находится в той же папке, которая предназначена для данного контроллера, то в метод View() достаточно передать название представления без расширения:

|  |
| --- |
| public class HomeController: Controller  {      public IActionResult Index()      {          return View("About");      }  } |

В этом случае метод Index будет использовать представление *Views/Home/About.cshtml*. Если же представление находится в другой папке, то нам надо передать полный путь к представлению:

|  |
| --- |
| public class HomeController: Controller  {      public IActionResult Index()      {          return View("~/Views/Some/Index.cshtml");      }  } |

Пример формирование таблицы в строго типизированном представлении с использованием строго типизированных хелперов:

@model IEnumerable<FuelStation.Models.Tank>

<h2>Емкости</h2>

<table class="table">

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.TankType)

</th>

<th>

@Html.DisplayNameFor(model => model.TankWeight)

</th>

<th>

@Html.DisplayNameFor(model => model.TankVolume)

</th>

<th>

@Html.DisplayNameFor(model => model.TankMaterial)

</th>

</tr>

</thead>

<tbody>

@foreach (var item in Model)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.TankType)

</td>

<td>

@Html.DisplayFor(modelItem => item.TankWeight)

</td>

<td>

@Html.DisplayFor(modelItem => item.TankVolume)

</td>

<td>

@Html.DisplayFor(modelItem => item.TankMaterial)

</td>

</tr>

}

</tbody>

Т. е. формирование таблицы осуществляется путем циклического динамического формирования набора HTML-тегов строк и ячеек.

Такое представление может вызываться, например, следующим контроллером:

public class FuelsController: Controller

{

private readonly FuelsContext \_context;

public FuelsController(FuelsContext context)

{

\_context = context;

}

// GET: Fuels

public IActionResult Index()

{

// Сортировка и фильтрация данных

var fuels = \_context.Fuels;

return View(fuels);

}

}

**Использование атрибута ResponseCache для кэширования.** <https://metanit.com/sharp/aspnet5/14.2.php>

**2. ЗАДАНИЕ**

Создать с использованием ASP.NET Core MVC Web-приложение, содержащее набор классов, моделирующих предметную область, и осуществляющих генерацию и заполнение тестовыми наборами записей базу данных. Разработать один компонент middleware, контроллеры и представления для выборки и отображения информации из не менее чем 3- таблиц базы данных с использованием механизма внедрение зависимостей.

Для выполнения задания необходимо создать:

* Классы, моделирующие не менее чем три таблицы базы данных согласно вашему варианту. Перечень таблиц предварительно согласовывается с преподавателем. Одна из таблиц обязательно должна находиться на стороне отношения «многие» связи с другой таблицей в схеме базы данных.
* Класс контекста данных.
* Другие классы, например классы View Model и т.п. (при необходимости).
* Компонент middleware для инициализации базы данных путем заполнения ее таблиц тестовым набором записей.
* Классы контроллеров (по одному на каждую таблицу базы данных) для обработки обращений пользователя, выборки данных из таблиц и вызова соответствующих представлений для отображения выбранных данных.
* Разработать представления для отображения данных из таблиц, выбранных контроллерами. Представления, работающими с таблицами, стоящими на стороне отношения «многие» в схеме базы данных, должны выводить вместо кодов внешних ключей смысловые значения из связанных таблиц, стоящих на стороне отношения «один».
* Используя предварительно созданный и сконфигурированный профиль кэширования, подключить кэширование вывода для страниц с использованием атрибута ResponseCache для соответствующих методов контроллера. Данные в кэше хранить неизменными в течение 2\*N+240 секунд, где N- номер вашего варианта.
* С использованием средств разработчика браузера (Chrome, Firefox) продемонстрировать ускорение обработки запроса при наличии кэширования с использованием атрибута ResponseCache.

Разместить выполненный проект в репозитории на GitHub, создать README.md файл.

Используя средство GitHub Actions, написать рабочий процесс, который будет осуществлять компиляцию проекта под две разные платформы при любом изменении в репозитории.

Отредактировать README.md файл опубликованного проекта, вставив в него код для создания эмблемы состояния рабочего процесса (status badge), показывающей, чем в данный момент завершился рабочий процесс.

**3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ПРИМЕРЫ**

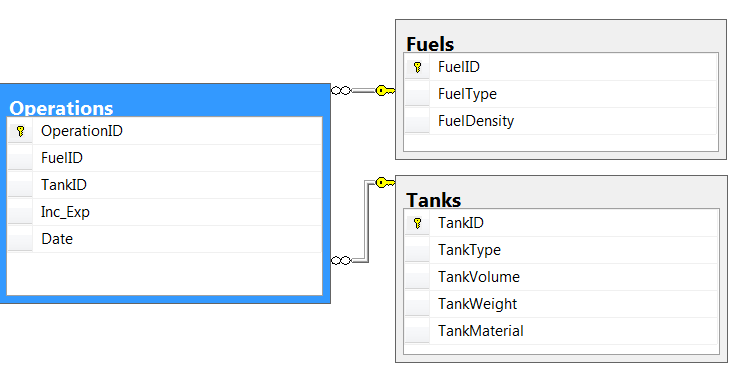
**Создание простого приложения MVC**

Есть Пусть мы хотим создать и использовать простую базу данных TOPLIVO, с тремя таблицами:

1. виды топлива (Fuels)

2. список емкостей (Tanks)

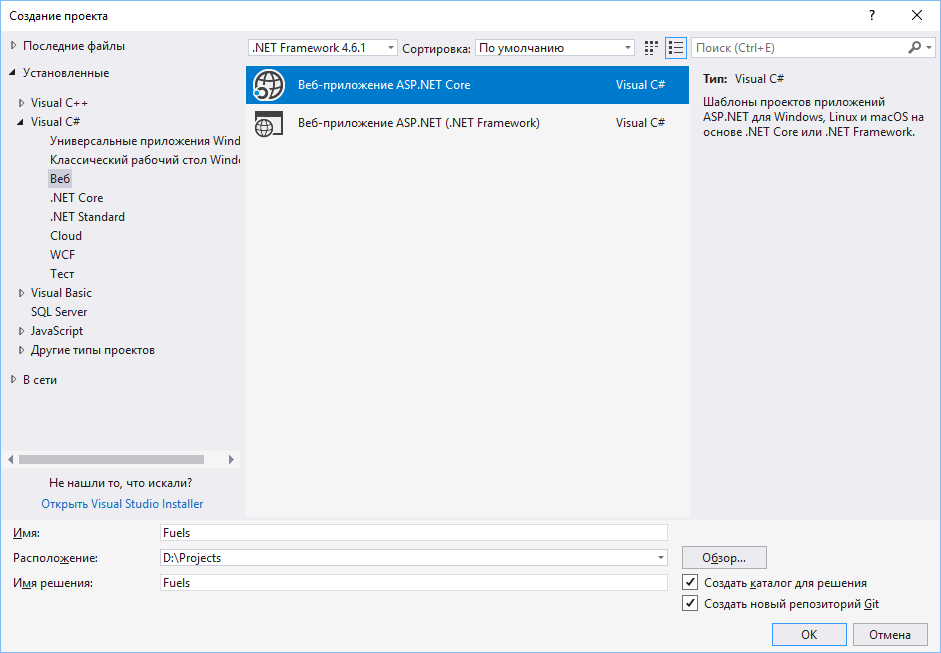
3. факты совершения операций прихода, расхода топлива (Operations)



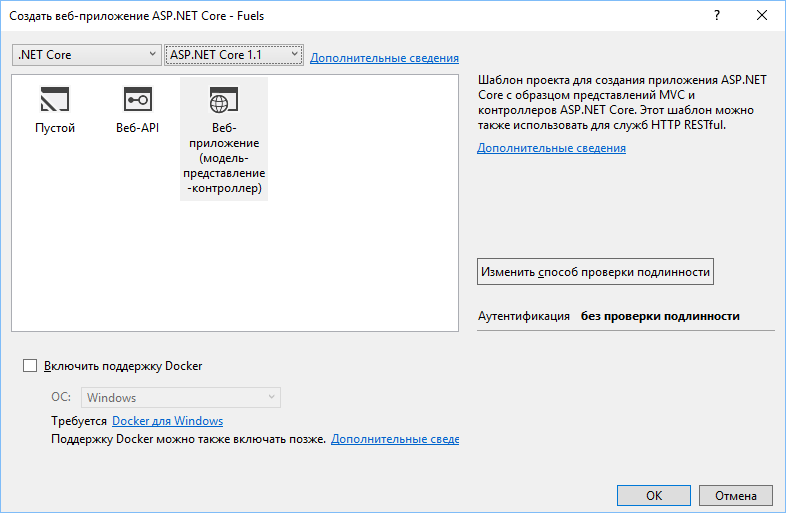
Опишем процедуру создания части МVС приложения для работы с ней, используя средства Visual Studio

**Создание проекта**

Откроем Visual Studio File -> New Project и создадим новый проект. Назовем новый проект, например, Fuels.



Затем в окне создания нового проекта выберем Веб-приложение (модель-представление-контроллер). А в правой части окна выберем тип аутентификации без аутентификации (так как пока нам система аутентификации не нужна):



После этого будет создан проект, который практически не обладает никакой функциональностью, хотя уже имеет базовую структуру.

**Создание классов моделей.**

Определим модели данных нашего приложения.

## Соглашения по наименованию при миграции

При создании таблиц и их столбцов в базе данных в Entity Framework по умолчанию действуют некоторые соглашения по именованию, которые указывают, какие имена должны быть у таблиц, столбцов, какие типы и т.д.

Если нас не устраивают названия таблиц и столбцов по умолчанию, то мы можем переопределить данный механизм с помощью Fluent API или аннотаций.

Добавим в проект новую папку, которую назовем Models и которая будет хранить модели приложения. Добавим в эту папку классы: Fuel (вид топлива), Tank (список емкостей), Operation (факт совершения операции прихода, расхода топлива)

Добавим в класс Fuel код, описывающий модель вида топлива:

using System.Collections.Generic;

namespace Fuels.Models

{

public class Fuel

{

//Id Топлива

public int FuelID { get; set; }

//Название вида топлива

public string FuelType { get; set; }

//Плотность вида топлива

public float FuelDensity { get; set; }

public virtual ICollection<Operation> Operations { get; set; }

public Fuel()

{

Operations = new List<Operation>();

}

}

}

Аналогично добавим классы, описывающие модели емкостей для хранения топлива (класс Tank) и операций с топливом и емкостью (Operation).

using System.Collections.Generic;

namespace Fuels.Models

{

public class Tank

{

//ID емкости

public int TankID { get; set; }

//Тип емкости

public string TankType { get; set; }

//Вес емкости

public float TankWeight { get; set; }

//Объем емкости

public float TankVolume { get; set; }

//Материал емкости

public string TankMaterial { get; set; }

//ссылка на файл изображения емкости

public string TankPicture { get; set; }

public virtual ICollection<Operation> Operations { get; set; }

}

}

namespace Fuels.Models

{

public class Operation

{

//ID операции

public int OperationID { get; set; }

//ID топлива

public int? FuelID { get; set; }

//ID емкости

public int? TankID { get; set; }

//Приход/Расход

public float? Inc\_Exp { get; set; }

//Дата операции

public System.DateTime Date { get; set; }

//ссылка на виды топлива

public virtual Fuel Fuel { get; set; }

//ссылка на емкости

public virtual Tank Tank { get; set; }

}

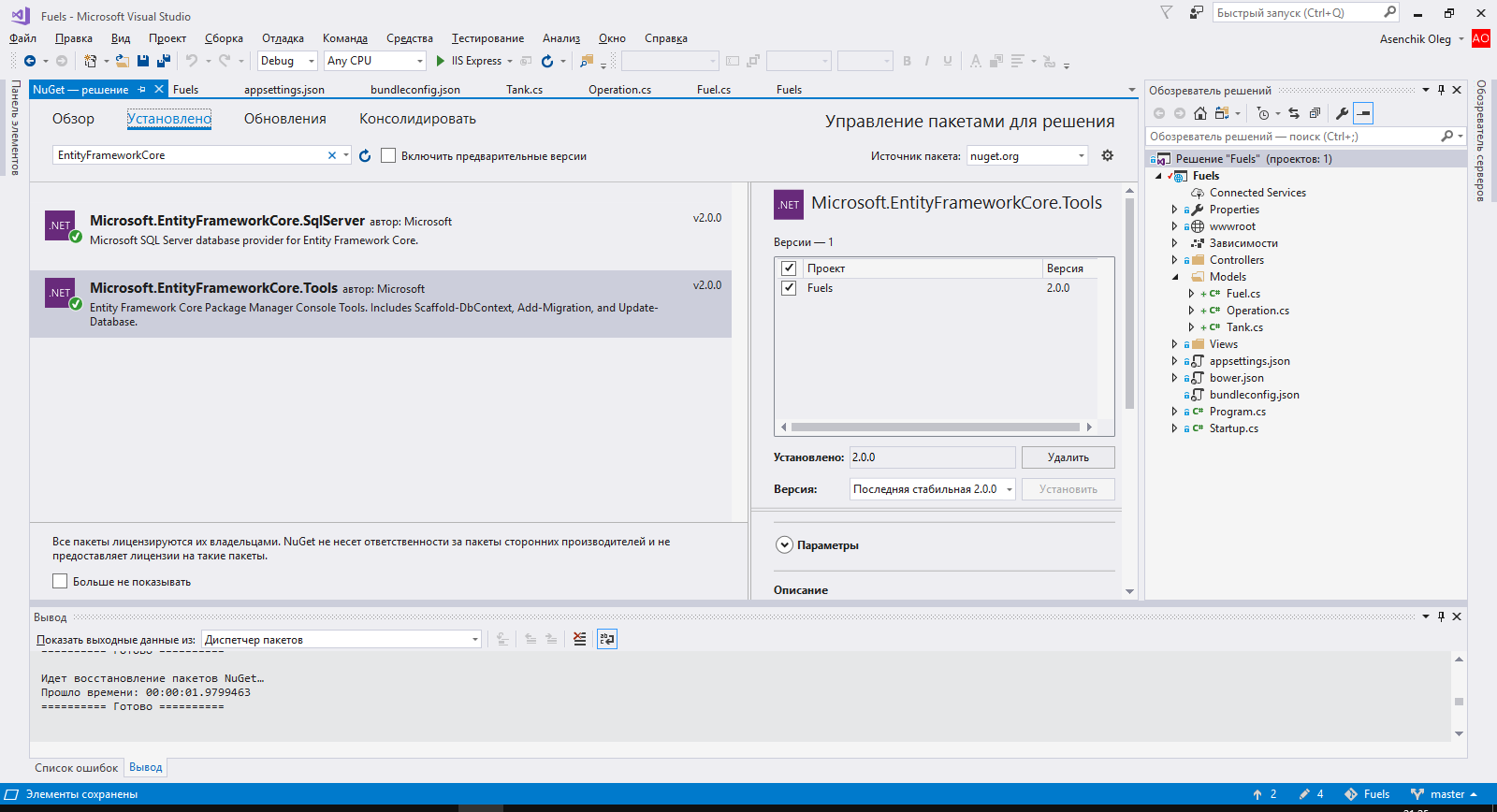
}

**Подключение EntityFramework**

После определения моделей надо выбрать хранилище данных для этих моделей. Мы будем использовать MS SQL Server. Для работы с MS SQL Server компания Microsoft рекомендует использовать ORM-технологию Entity Framework, хотя ее использование необязательно. Мы также можем применять другие ORM-технологии или доступные средства ADO.NET. Преимущество фреймворка Entity Framework состоит в том, что он позволяет абстрагироваться от структуры конкретной базы данных и вести все операции с данными через модель.

В данном случае для взаимодействия с MS SQL Server через Entity Framework нам нужен пакет Microsoft.EntityFrameworkCore.SqlServer. Но, кроме того, перед работой с базой данных нам предварительно надо создать эту базу данных в соответствии с вышеопределенными моделями. И для этого потребуется пакет Microsoft.EntityFrameworkCore.Tools.

Если мы работаем с проектом ASP.NET Core 1.0/1.1, то умолчанию эти пакеты отсутствуют, и нам их надо добавлять через пакетный менеджер Nuget.



Если мы работаем с проектом ASP.NET Core 2.0 и выше, то эти пакеты уже имеются в проекте в рамках общего пакета Microsoft.AspNetCore.All.

Чтобы взаимодействовать с базой данных нам нужен контекст данных. Причем Entity Framework Core использует подход Code First, при котором нам надо сначала определить модели и контекст данных, а потом уже исходя и этих моделей и класса контекста будет создаваться бд и все ее таблицы.

### Создание контекста данных

После завершения установки создадим контекст данных. Контекст данных использует EntityFramework для доступа к БД на основе некоторой модели. Итак, добавим в папку Models новый класс FuelsContext:

using Microsoft.EntityFrameworkCore;

namespace Fuels.Models

{

public class FuelsContext: DbContext

{

public FuelsContext(DbContextOptions<FuelsContext> options): base(options)

{

}

public DbSet<Fuel> Fuels { get; set; }

public DbSet<Operation> Operations { get; set; }

public DbSet<Tank> Tanks { get; set; }

}

}

Чтобы создать класс контекст, нам надо унаследовать его от класса DbContext. Свойства наподобие public DbSet<Fuel> Fuels { get; set; } помогают получать из БД набор данных определенного типа (например, набор объектов Fuel).

Для того чтобы указать EF путь и методы доступа к создаваемой базе данных, определим строку подключения в файле *appsettings.json*

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=fuels;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Warning"

}

}

}

или

{

"ConnectionStrings": {

"DefaultConnection": "Server=.\\sqlexpress;Database=fuels;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Warning"

}

}

}

Для подключения необходимых сервисов изменим файл Startup.cs. В нем нам надо изменить метод ConfigureServices():

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Fuels.Models;

using Microsoft.EntityFrameworkCore;

namespace Fuels

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

// This method gets called by the runtime. Use this method to add services to the container.

public void ConfigureServices(IServiceCollection services)

{

string connection = Configuration.GetConnectionString("DefaultConnection");

services.AddDbContext<FuelsContext>(options => options.UseSqlServer(connection));

services.AddMvc();

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

app.UseBrowserLink();

}

else

{

app.UseExceptionHandler("/Home/Error");

}

app.UseStaticFiles();

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

}

}

}

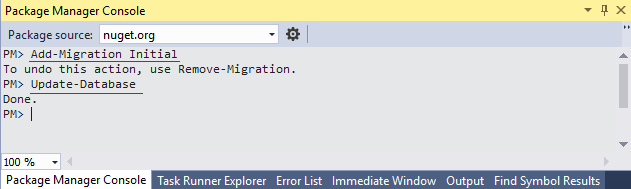
Если база данных по указанному в строке подключения адресу не существует, ее необходимо создать. Для этого откроем окно Package Manager Console. Его можно найти в меню Tools -> Nuget Package Manager -> Package Manager Console. Последовательно введем в это окно две команды. Сначала выполним команду для фиксации текущего состояния базы данных (миграцию):

Add-Migration Initial

Эта команда добавит в проект файлы миграции, которая будет называться Initial и которая будет использоваться для создания таблиц в базе данных. Затем выполним команду:

Update-Database

Эта команда выполнит ранее созданную миграцию, в результате чего на сервере MS SQL Server будет создана новая база данных, в которой будут определены таблицы, соответствующие моделям и контексту данных.



**Инициализация базы данных**

Если нам необходимо, чтобы при первом обращении база данных уже была заполнена некоторыми начальными значениями, то мы можем произвести ее инициализацию.

Создаем класс инициализатора базы данных DbInitializer:

using System;

using System.Linq;

//Класс для инициализации базы данных путем заполнения ее таблиц тестовым набором записей

namespace Fuels.Models

{

public static class DbInitializer

{

public static void Initialize(FuelsContext db)

{

db.Database.EnsureCreated();

// Проверка занесены ли виды топлива

if (db.Fuels.Any())

{

return; // База данных инициализирована

}

int tanks\_number = 35;

int fuels\_number = 35;

int operations\_number = 300;

string tankType;

string tankMaterial;

float tankWeight;

float tankVolume;

string fuelType;

float fuelDensity;

Random randObj = new Random(1);

//Заполнение таблицы емкостей

string[] tank\_voc = { "Цистерна\_", "Ведро\_", "Бак\_", "Фляга\_", "Цистерна\_" };//словарь названий емкостей

string[] material\_voc = { "Сталь", "Платина", "Алюминий", "ПЭТ", "Чугун", "Алюминий", "Сталь" };//словарь названий видов топлива

int count\_tank\_voc = tank\_voc.GetLength(0);

int count\_material\_voc = material\_voc.GetLength(0);

for (int tankID = 1; tankID <= tanks\_number; tankID++)

{

tankType = tank\_voc[randObj.Next(count\_tank\_voc)] + tankID.ToString();

tankMaterial = material\_voc[randObj.Next(count\_material\_voc)];

tankWeight = 500 \* (float)randObj.NextDouble();

tankVolume = 200 \* (float)randObj.NextDouble();

db.Tanks.Add(new Tank { TankType = tankType, TankWeight = tankWeight, TankVolume = tankVolume, TankMaterial = tankMaterial });

}

//сохранение изменений в базу данных, связанную с объектом контекста

db.SaveChanges();

//Заполнение таблицы видов топлива

string[] fuel\_voc = { "Нефть\_", "Бензин\_", "Керосин\_", "Мазут\_", "Спирт\_" };

int count\_fuel\_voc = fuel\_voc.GetLength(0);

for (int fuelID = 1; fuelID <= fuels\_number; fuelID++)

{

fuelType = fuel\_voc[randObj.Next(count\_fuel\_voc)] + fuelID.ToString();

fuelDensity = 2 \* (float)randObj.NextDouble();

db.Fuels.Add(new Fuel { FuelType = fuelType, FuelDensity = fuelDensity });

}

//сохранение изменений в базу данных, связанную с объектом контекста

db.SaveChanges();

//Заполнение таблицы операций

for (int operationID = 1; operationID <= operations\_number; operationID++)

{

int tankID = randObj.Next(1, tanks\_number - 1);

int fuelID = randObj.Next(1, fuels\_number - 1);

int inc\_exp = randObj.Next(200) - 100;

DateTime today = DateTime.Now.Date;

DateTime operationdate = today.AddDays(-operationID);

db.Operations.Add(new Operation { TankID = tankID, FuelID = fuelID, Inc\_Exp = inc\_exp, Date = operationdate });

}

//сохранение изменений в базу данных, связанную с объектом контекста

db.SaveChanges();

}

}

}

Это класс, например, в дальнейшем можно использовать для создания собственного компонента middleware, внедренного в конвейер обработки HTTP-запроса.

**4. ОФОРМЛЕНИЕ ОТЧЕТА**

Отчет должен содержать:

* Исходные коды стартового класса Program.cs (Startup.cs), классов моделей и класса контекста данных.
* Строку подключения к базе данных из конфигурационного файла.
* Исходный код компонента middleware для инициализации базы данных путем заполнения ее таблиц тестовым набором записей.
* Исходные код контроллеров и представлений.
* Графические представления отображения представлений в браузере.
* Ссылка на код разработанного проекта на GitHub.
* Содержимое yml файла рабочего процесса.

**5. КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Общая характеристика HTTP протокола.
2. Структура строки состояния и заголовка HTTP пакета посылаемого клиентом.
3. Структура строки состояния и заголовка HTTP пакета посылаемого сервером.
4. Характеристика метода GET протокола HTTP.
5. Характеристика метода POST протокола HTTP.
6. Характеристика метода PUT протокола HTTP.
7. Характеристика метода DELETE протокола HTTP.
8. Обработка HTTP запросов контроллером ASP.NET Core.
9. Способы вызова статического HTML файла из метода контроллера
10. Запуск ASP.NET Core приложения. Классы Program и Startup.
11. Конвейер обработки запроса и middleware. Создание собственных компонентов middleware.
12. Средства для взаимодействия ASP.NET Core приложения со средой, в которой оно запущено.
13. Серверы для обеспечения хостинга ASP.NET Core приложения.
14. Понятие Dependency Injection (DI) или внедрения зависимостей. Способы реализации в ASP.NET Core.
15. Cервисы и их применение. Создание собственных сервисов.
16. Реализация шаблона MVC (Model View Controller) в ASP.NET и Visual Studio.
17. Общая характеристика контроллеров (Controllers): создание, методы, методы действий, возвращаемые результаты, способы вызова.
18. Результаты действий, классы ContentResult и JsonResult. Переадресация.
19. Внедрение зависимостей. Передача зависимостей контроллеру.
20. Жизненный цикл объектов при внедрении зависимостей. Передача конфигурации в контроллер.
21. Способы получение данных контроллером.
22. Передача параметров из представлений в контроллер. Стандартный привязчик DefaultModelBinder.
23. Передача данных из метода действия контроллера в представление. Примеры.
24. Для чего используются объекты ViewBag и ViewData? В чем их отличие?
25. Общая характеристика моделей. Entity Framework.
26. Маршрутизация URL запросов ASP.NET MVC приложением.
27. Как настраивается маршрутизация при наличии в проекте областей (Area)?
28. Для чего используются и в чем отличие классов ActionResult, PartialViewResult, ViewResult?

**6.** **ЛИТЕРАТУРА**

1. Пример на GitHub <https://github.com/Olgasn/FuelStation/tree/caching>

2. Руководство по EntityFramework Core <https://metanit.com/sharp/efcore/>

3. Руководство по ASP.NET Core - <https://metanit.com/sharp/aspnet6/>

4. Руководство по ASP.NET Core MVC - <https://metanit.com/sharp/aspnetmvc/>

5. Руководство по ASP.NET Core MVC от Microsoft ‑ <https://docs.microsoft.com/ru-ru/aspnet/core/mvc/overview>