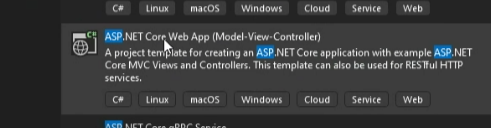
23.03.2024



Сначала запускаем проект, соглашаемся на установку сертификатов

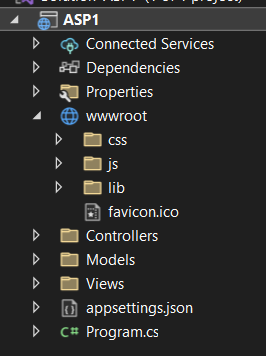
certlm.msc

<https://learn.microsoft.com/ru-ru/dotnet/core/tools/dotnet-dev-certs>

Проверить наличие сертификата

dotnet dev-certs https

Но! Нужно закрыть все браузеры и перезапуститься

Папка wwwroot – туда имеет доступ браузер (**то, что происходит на клиенте**)

Уже установленны библиотеки Бутстрап, Jquery

Сюда вкладываем статические компоненты, которые передаются на клиента

Для сервера это неактивные компоненты, они передаются только на клиента (выполняются браузером)

Все, что находится вне этой папки уже передается на сервер

wwwroot – то, что не менется

Controllers, Models, Views – то, что выполняется сервером. Паттерн MVC.

appsettings – конфигурационные настройки

Program.cs – стартовый файл, нас интересует вот эта строчка:

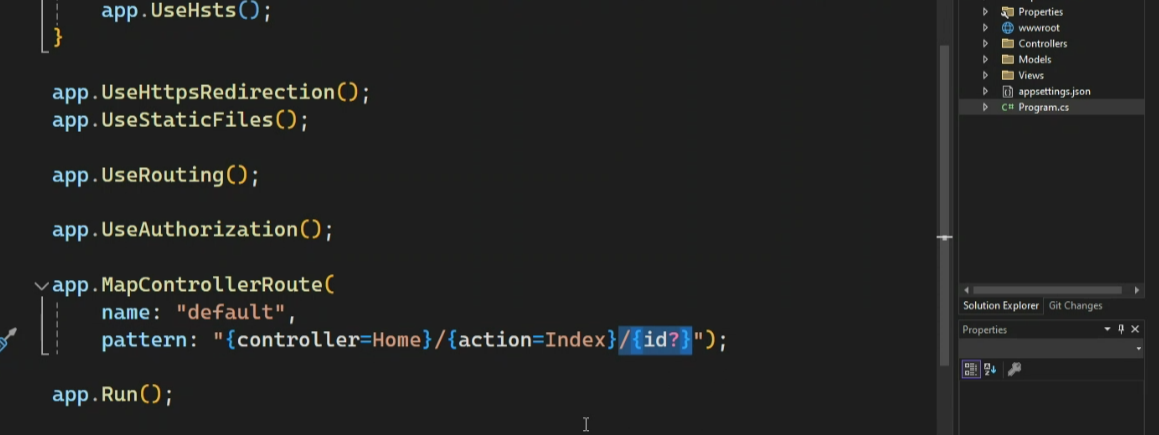
pattern: "{controller=Home}/{action=Index}/{id?}");

Когда мы в браузере набираем адрес нашего сайта, наш сайт начинает его анализировать следующим способом:

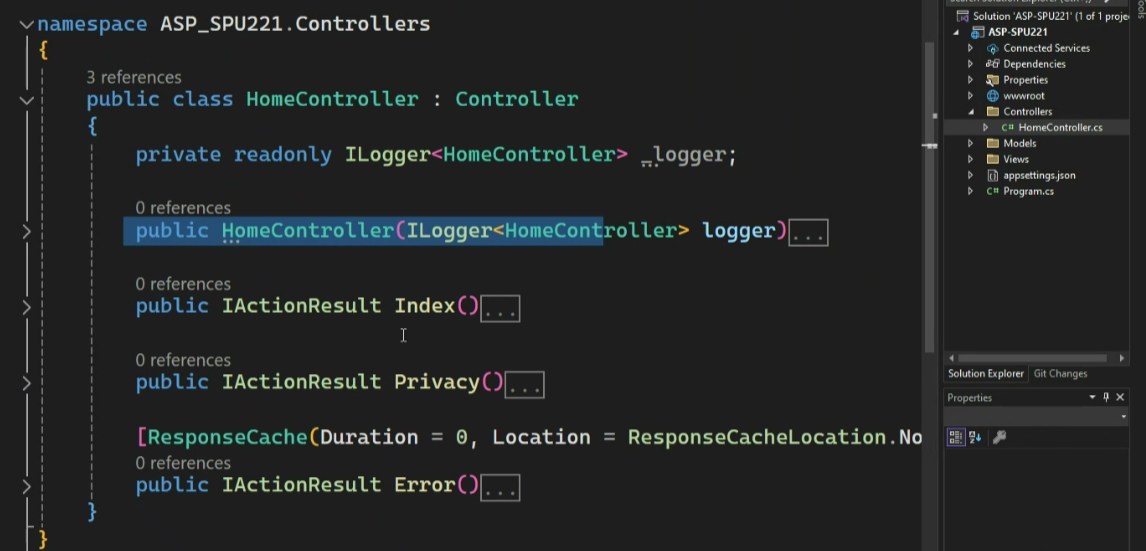
Все, что идет до первого слеша считается контроллером, дефолтное значение Home, после первого слеша до второго – экшен, по умолчанию – Индекс, дальше все до конца считается id? - ? значит null

Это общая структура маршрутизации, система ASP разбирает запрос по этой схеме

Это приводит к тому, что к слову Home дописывается слово controller и в папке Controllers ищется соответстующий файл



Общая структура марштрутизации, которая приводит к тому, что система разбирает запрос по этой схеме, первая часть считается контроллером, вторая – экш и т.д. и в папке Controller ищется соответствующий файл



Помимо конструктора мы имеет три метода: Index, Privacy, Error, они называются Action, что подчеркивается типом IActionResult.

**Action называются методы Контроллеров. Слово метод несколько конфликтует со словом метод запроса (Get, Post). Чтобы не путать методы контролеров и методы запросов, для методов контроллеров используется слово Action.**

**Значит передается управление на HomeController и вторая часть передает управление на action Index (тоже ищет по имени).**

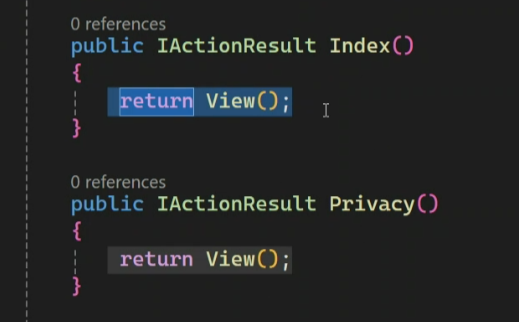
Из коробки нам дается три экшены Index, Privace, Error

Контроллеров может быть много, название составляется из имени + Controller

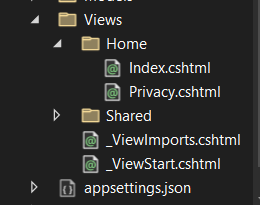
В нем есть разные Action, которые являются вторым слешем. Index – по умолчанию, остальные надо прямо указать

В браузере через слеши мы указываем, какой экшн какого контроллера нужно запустить, а дальше система сама разбирает этот адрес, находит контроллер и запускает соответствующую точку.

Когда эта точка запущена, происходит возвращение представления View()



return View работает не как запуск какого-то метода, а тоже как поиск – заходит в папку View, находит папку по имени контроллера и в ней файл с соответствующим именем экшена



Как правило, View совпадает с названием Action (можно вручную указать, какое представление подключать)

Для вставки кодов в html используется Razor – технология, похожая на php. Его основой является символ @. Это html, в который можно включить какие-то программные сущности (cshtml)

В файле Privacy нет менюшки, нет футера, они берутся из папки папке Shared – находятся вью для разных контроллеров (там же файл Error), т.е. одно и то же представление для разных контроллеров (View сначала ищется в своей папке, потом в общей папке)

Layout.cshtml – главный шаблон для всех

Внутри есть

<div class="container">

<main role="main" class="pb-3">

@RenderBody()

</main>

</div>

Этот код включает наше представление в центр и на ответ идет страница

ASP (Active Server Page) – технология от Microsoft для создания веб-приложений. Вариант MVC один из наиболее современных и позволяет реализовывать другие варианты проектов своими средствами.

C помощью ASP и MVC мы можем создать как Razor Page, тaк и WebApi и другие типы проектов.

В ASP кроме Razor есть так называемые хелперы, которые позволяют создавать ссылки

<h3><a asp-controller="" asp-action="">Razor></a></h3>

Вместо прямой гиперссылки рекомендуется использование этих хелперов, которые позволяют собрать правильную ссылку из данных про контроллер и экшн (слеш контроллер, слеш экшн, это по умолчанию, мы можем это поменять, но чтобы не переделывать все ссылки, лучше сразу закладывать в них правильную структуру)

<h3><a asp-controller="Home" asp-action="Razor">Razor></a></h3>

Итак, делаем ссылку на контроллер Home и его экшн Razor

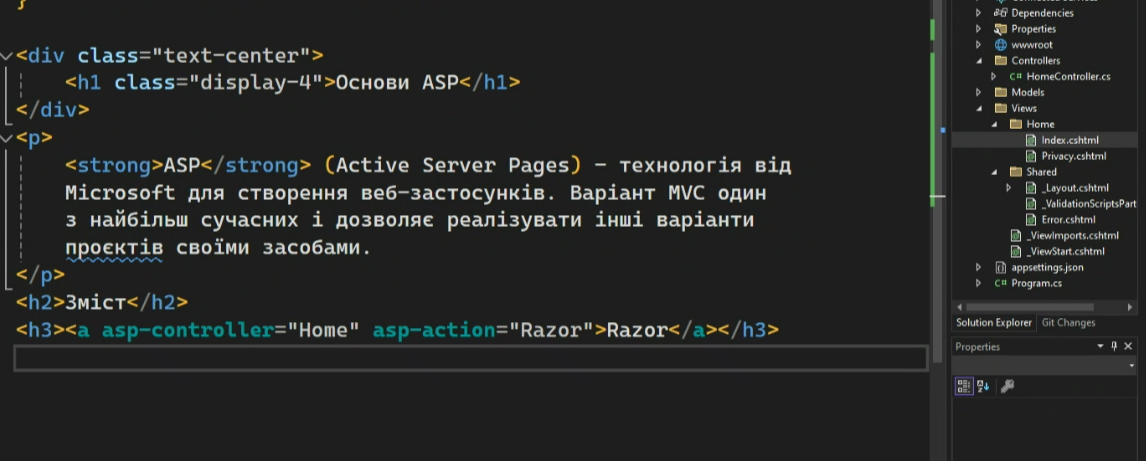
Еще раз: Открываем Program.cs, ищем сроку

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

Принцип маршрутизации ASP состоит в том, что адрес преобразуется в выбор контроллера и выбор экшена. Адрес раскладывается и из нее определяется, какой контрллер и какой экшен нужно запустить.

Этот паттерн можно поменять. Например, добавить перед контроллером информацию про локаль (язык) (традиционна локализация идет первой). В таком случае на первом месту будет локаль, а на втором – контроллер. Если напрямую написать слеш Home/Index эта ссылка станет неправильной

Чтобы не закладывать проблему перезаписи сразу в ссылке закладываем контроллер и экш, а как оно соберется в ссылку, решит уже сам ASP из анализа паттерна



При таком подходе получается по сути относительный путь

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

То есть, прописанные asp-controller, asp-action – это только инструкции для формирования гиперссылки. В html этого не будет, будет просто нормальная гиперссылка. Но, чтоб он поставил на свои места контроллер и экшен, мы это прописываем в ссылке

Чтобы сделать ресурс, нужно сделать в контроллере Home

HomeController.cs делаем action Razor

public IActionResult Razor()

{

return View();

}

Но View мы еще не сделали, поэтому выдает ErrorPage

Изображение выглядит как текст, программное обеспечение, Шрифт, снимок экрана

Автоматически созданное описание

Тут видно, где он ищет представление

Идем в папку Views=>Home=>правой кнопкой =>Add=>View=>Razor Empty=> Razor (название View совпадает с названием Action)

Создается представление. Заготовка

@{

ViewData["Title"] = "Razor";

}

<div class="text-center">

<h1 class="display-3">Основи синтаксису Razor</h1>

</div>

Основу синтаксиса Razor является символ @

Можно выделить два режима Razor: операции и выражения

* Выражение – инструкции, у которых есть результат и Razor выводит автоматически этот результат – включает его в html
* Операция (инструкция) – инструкции без результата, Razor их просто выполняет, что позволяет, в частности, вводить переменные, организовывать циклы, реализовывать условные блоки.

Выражения – наиболее используемый способ Razor, в простейших случаях выражение просто обозначается символом @, например,

@ViewData["Title"] выведет Razor

Анализатор Razor самостоятельно пытается определить пределы выражения, а также игнорирует выражения, которые по шаблону являются электронной почтой.

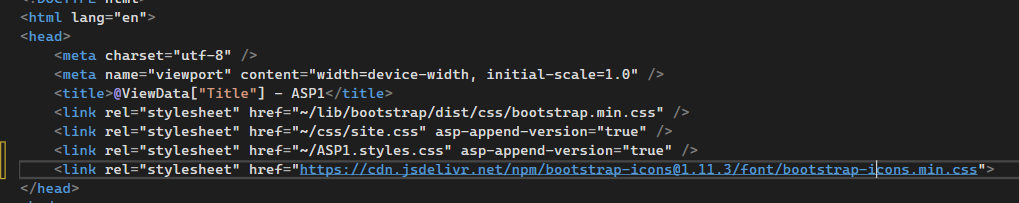
Это значит, что можно написать [user@ukr.net](mailto:user@ukr.net) и оно не будет пытаться подставить его как выражение

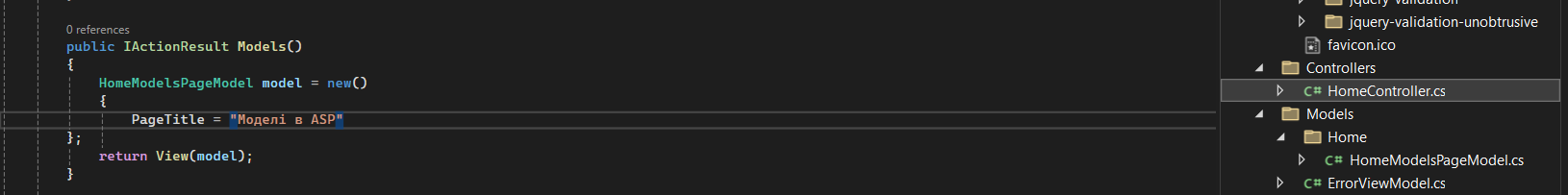
<https://getbootstrap.com/docs/5.1/getting-started/introduction/>

https://icons.getbootstrap.com/

Подключить иконки BootStrap: (Shared=>Layout>

<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.11.3/font/bootstrap-icons.min.css">





Подключить Модель:

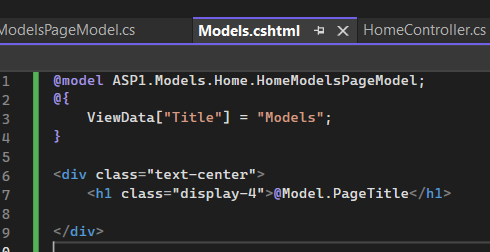
@model ASP1.Models.Home.HomeModelsPageModel;

Обращение к модели:

<div class="text-center">

<h1 class="display-4">@Model.PageTitle</h1>

</div>



Передача с контроллера на вьюшку происходит через Model

Есть еще один механизм передачи с контроллера на View через ViewData – суперглобальный Dictionary, в который можно в одном месте положить данные, в другом – достать. Это более старый механизм. PageModel – то, что содержит все, что необходимо для отображения страницы.

Alt+Shift+. – находит следующие вхождения слова в VisualStudio

**30.03.24**

**Инверсия управления. Сервисы**

Создание страницы начинаем с Controller

Идем в HomeController => создаем Action для инверсии управления

public ViewResult IoC()

{

return View();

}

Далее идем во View=>Home=>создаем новое представление IoC

@{

ViewData["Title"] ="IoS";

}

<div class="text-center">

<h1 class ="display-4"> Инверсия управления. Сервисы </h1></div>

**Inversion of control (IoC),** инверсия управления – такой архитектурный прием (паттерн) к проектам, согласно которому управление жизненным циклом объектов перекладывается на отдельный модуль (инжектор, контейнер зависимостей).

В проекте вместо команд new создаются точки инжекции, к которым включаются объекты, созданные инжектором.

Это создается для того, чтобы уменьшить количество объектов в проекте.

Роль контейнера можно сравнить с глобальными переменными, которые доступны во всех частях проекта, но этот контейнер не только содержит объекты, а и контролирует их время жизни (scope) – решает, создавать ли разные объекты для разных точек инжекции, или передавать один и тот же объект.

Например, подключение к БД не должно дублироваться, с другими ресурсами также. Это создает определенную централизацию, уменьшение ресурсов, легкость замены через централизацию – плюсы инверсии управления. Тот же жизненный цикл (есть объекты, которые должны пересоздаваться, есть те, которые не нужно пересоздавать).

**Сервисами в этом контексте называют именно те объекты, которые являются единственными на весь проект** (которые не пересоздаются). Инверсионные/инжекционные сервисы – те долгоживующие объекты, которые передаются между контроллерами. Эти объекты, как правило, что-то делают (у них есть методы), их можно противопоставить моделями (объектам без методов. Данные есть, методов нет).

Инверсия управления считается относительно сложным для понимания паттерном.

Процесс работы при наличии инверсии управления (IoC) выглядит следующим образом:

- описываются сервисы – интерфейсы и их имплементации (классы)(Объекты из класса инстанциируются, а Интерфейсы – имплементируются).

- осуществляется регистрация этих типов данных (классов) в инжекторе (в контейнере сервисов)

- в классах-потребителях оглашается инжекция этих сервисов

По сути с объектами мы нигде не имеем дело.

При первом обращении к сервису будет создан его объект (инстанциировано) и в дальнейшем он может использоваться повторно, если такое его время жизни. Чаще всего (в 90-95%) они постоянной жизни, очень редко приходится делать кратко живущие объекты.

Ленивая инстанциация – если не будет запроса на этот сервис, то его объект и не создастся.

При оглашении сервисов следует придерживаться принципа DIP (з SOLID) – Dependency Inversion Principle – принципа инверсии зависимостей.

(!! Не путать термины: «инверсия управления», «инверсия зависимости», «инжекция зависимости» - передача объектов в точки инжекции (контроллеры, другие точки)).

DIP (не создавать зависимости от реализаций, создавать зависимости от абстракций) рекомендует описывать сервисы начиная с интерфейсов. Чем выше абстракцию берем для зависимости, тем лучше.

Антипример: есть сервис, в котором зарегистрирован американский стандарт-шифратор, и есть контроллер, в котором идет впровадження, который зависит от этот сервиса (зависимость от имплементации, т.е. от конкретного американского шифратора):

Service: AesService --> HomeController(AesService)

Проблема: если изменить сервис AesService --> KalinaService, то придется переписывать все контроллеры (точки инжекции) для приема нового сервиса.

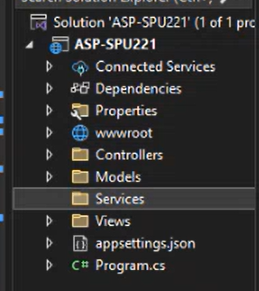
Решение: создать интерфейс (абстракцию), например, какой-то ICryptoService и через него осуществлять инжекцию – HomeController(ICryptoService)

Тогда изменение конкретного блока (конкретной имплементации) не вынудит менять что-то в контроллерах (будет приемлемой без переделок), главное, чтобы оба сервисы реализовывали один и тот же интерфейс

Практический вывод: **сервисы всегда создаются «парой»: интерфейс-класс.** Даже если пока ничего не собираемся менять, все равно сервисы создаются именно таким способом.

Это нужно для большей стойкости! Если что-то поменяется, не придется переделывать.

Сервисы – отдельная сущность приложения. Имеют относительно самостоятельную роль. Можно даже переносить из одного проекта в другой.

На уровне MVC (в самом проекте) создаем новую папку Services.

На примере создания сервисов хеширования, аутентотификации по паролю. Для регистрации пользователей используем пароли, а пароли в базе данных не должны существовать как открытые пароли (они должны быть шифрованными).

Стандарты, по которым они должны быть преобразованы: следуем требованиям стандарта:

<https://datatracker.ietf.org/doc/html/rfc2898>

Стандарт описывает одну из схем аутентификации, которая использует пароль. Чтобы данные сохранялись в базе выделяются две сущности:

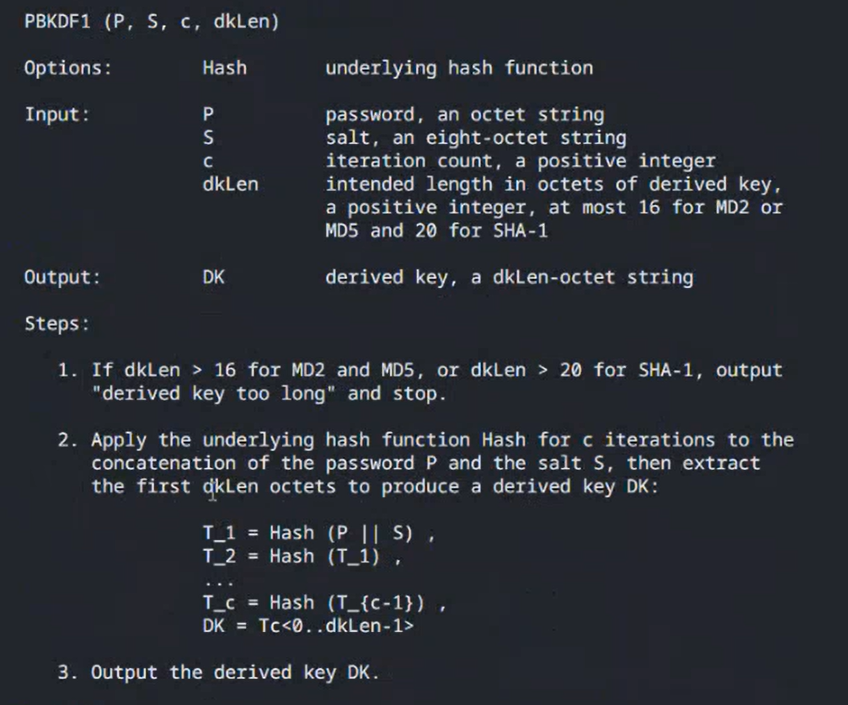
- salt - соль – случайная величина, Она генерируется чтобы несколько улучшить надежность хранения хешей. И когда мы записываем пароль – мы исходный пароль как-то сохраняем с солью и получаем то, что называется derivedKey (выходной ключ).

- derivedKey (хеш) – результат преобразования пароля и соли.

DK (Derived Key) – это то, что часто называют хеш, результат преобразования пароля и соли.

Процесс преобразования derivedKey называется Key Derivation Function (KDF)

По стандарту первый вариант, как он рекомендует это делать:



На первом этапе используется инструмент хеширования, т.е. пароль складывается с солью и получается хеш.

На втором этапе высчитывается хеш того, что было перед этим и т.д. столько раз, сколько нужно, в конце результат обрезается до необходимого количества символов.

Наша задача: описать два сервисы: хеш-сервис и KDF-сервис, которые будут реализовывать требования данного стандарта.

Последовательность работы:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | (единоразово) создаем папку для сервисов (всех) – Services; | | |  |
|  | создаем сервис хеширования. Для каждого сервиса, поскольку файлов будет, как минимум, два, также создаем свои папки | | |  |
|  | в папке сервиса Hash создаем два файла (интерфейс IHashService и какую-то его имплементацию, например Md5HashService – один из хешеров, 128-битный хеш, достаточно простой для нагрузки с определенными недостатками, но для задач аутентификации достаточный. | | |  |
|  | Описываем файл IHashService  I – Interface Segrigation – интерфейс не следует отяжелять, их надо делать максимально компактными. В идеале, чтобы в интерфесе был один мето ди все  //Digest - слово синонім від hash (як правило hash назва функції (алгоритм), а Digest - результат його работи)  MD5 – Message Digest (сокращенно) – один из криптоалгоритмов | | | namespace ASP1.Services.Hash  {  //Просто декларує (І в SOLID)  public interface IHashService  {  String Digest(String input);  }  } |
|  | Описываем файл Md5HashService – имплементация интерфеса IHashService  namespace ASP1.Services.Hash  {  public class Md5HashService **: IHashService**  {  public string Digest(string input)=>  Convert.ToHexString(System.Security.Cryptography.MD5.Create()  .ComputeHash(System.Text.Encoding.UTF8.GetBytes(input)));    }  } | | | |
|  | Регистрируем эти сервисы в Program.cs | | | Находим строчку  // Add services to the container.  builder.Services.AddControllersWithViews();  зарегистрировать сервисы надо между builder и app |
| //Реєстрація сервісу - його додавання до контейнеру build.Services  builder.Services.AddSingleton<IHashService, Md5HashService>();  При следовании DIP устанавливается «связка» между интерфейсом и имплементацией. Данную инструкцию можно читать таким образом:  «если будет запрос на инжекцию IHashService, то выдать объект класса Md5HashService». | | | |
|  | Инжектируем сервис в HomeController | Традиционно инжекция сервисов осущетсвляется через конструктор. Это исключает возможность создания объекта без передачи ему зависимостей (сервисов).  В изначальном проекте инжекция демонстрируется на примере \_logger  private readonly ILogger<HomeController> \_logger;  Оглашаем собствернную зависимость (инжекцию) – поле с модификатором private readonly – это не позволит изменять это поле случайными командами + зависимость через интерфейс. DIP- тип зависимости – абстрация (интерфейс)  private readonly IHashService \_hashService;  нажимаем по полю правой кнопкой => QuickActions => Add parameter to conctructor | | |
|  | Использование сервиса в любом месте, например в нашем представлении IoC | | | public ViewResult IoS()  {  HomeIoCPageModel model = new()  {  HashExample = \_hashService.Digest("123"),  };  return View(model);  } |
|  | В папке Models => Home => создаем HomeIoCPageModel  Создаем модель – необходимые данные для передачи с контроллера на представление | | | namespace ASP1.Models.Home  {  public class HomeIoCPageModel  {  public string HashExample { get; set; } = null!;  } |
|  | Идем на представление и заявляем, что работаем с определенной моделью | | | Файл IoS.cshtml  @model ASP1.Models.Home.HomeIoCPageModel |
|  | затем результат работы контролера передаем на представление:  <i>@Model.HashExample<i> | | | Вывелся хеш числа 123 |
| Мы нигде не писали new! Где создается объект – мы этого не увидим! Все происходит под капотом в рамках builder.Service и то, тут он просто регистрируется, сам конструктор запустится тогда, когда он увидит первую инжекцию, то есть когда мы первый раз зайдем на страницу IoC. Мы просто пишем типы данных, нигде вручную объект не создаем  Нам в конструктор передаются какие-то параметры. Передает их инжектор. Когда конструктор будет создаваться как объект, снова будет запрос к инжектору (к этому контейнеру «дай все нужные зависимости», это тоже будет определять фреймворк)  В этом суть инверсии управления: не мы управляем созданием объектов, а кто-то, что-то управляет, и уничтожением тоже.  Где срабатывает этот конструктор? Отстуствие видимой связи. Создание – не наше дело.  Поэтому такие проекты сложновато читать, нужно видеть, где точки инжекции, где настраиваем контейнер | | | | |
|  | Имитируем задачу перехода на новый способ хеширования.  По OCP (из SOLID) не вмешиваемся в работу предыдущего сервиса, а создаем новый (ShaHashService) | | | |
|  | Идем в папку Services=>Hash=>AddClass=>  SgaHashSrvice.cs | | namespace ASP1.Services.Hash  {  public class ShaHashService:IHashService  {  public string Digest(string input) =>  Convert.ToHexString(System.Security.  Cryptography.SHA1.Create()  .ComputeHash(System.Text.Encoding.  UTF8.GetBytes(input)));  }  } | |
|  | Меняем регистрацию сервиса в Program.cs | | builder.Services.AddSingleton<IHashService, ShaHashService>(); | |
|  |  | |  | |

**SCOPE**

**06.04.24**

**Работа с БД**

Entity Framework – один з інструментів роботи з БД

Нужно установить пакеты:

* Microsoft.EntityFrameworkCore – основной
* Microsoft.EnityFrameworkCore.Tools – инструменты комнадной строки
* Microsoft.EntityFramework.SqlServer – драйвер (провайдер, коннектор), который согласовывает работу фреймворка с конкретной БД. Если используются другие СУБД, третья библиотека будет другой

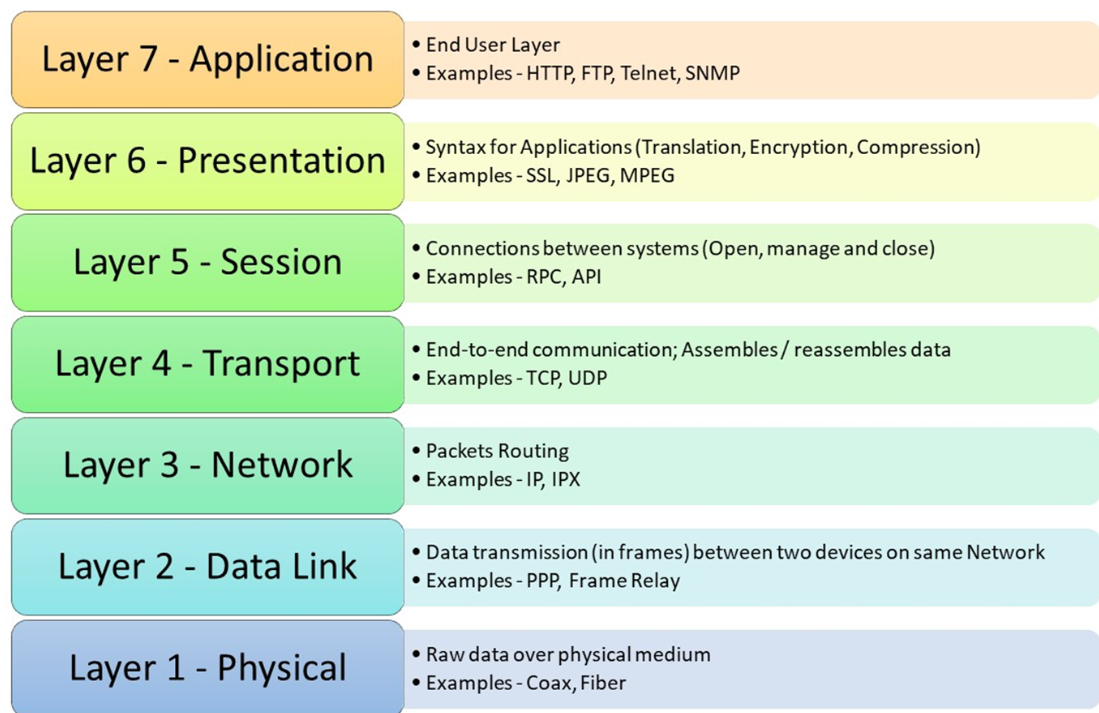
Контекст данных – отдельная архитектурная прослойка проекта. Это значит, что это относительно самостоятельное образование, которое может быть или перенесено в другой проект, или заменен другим образованием при переходе на другого поставщика данных, или другие технологии работы с данными, например, на WebApi (когда данные берутся не напрямую из БД, а из какого-то веб-сервиса, который предоставляет эти данные).

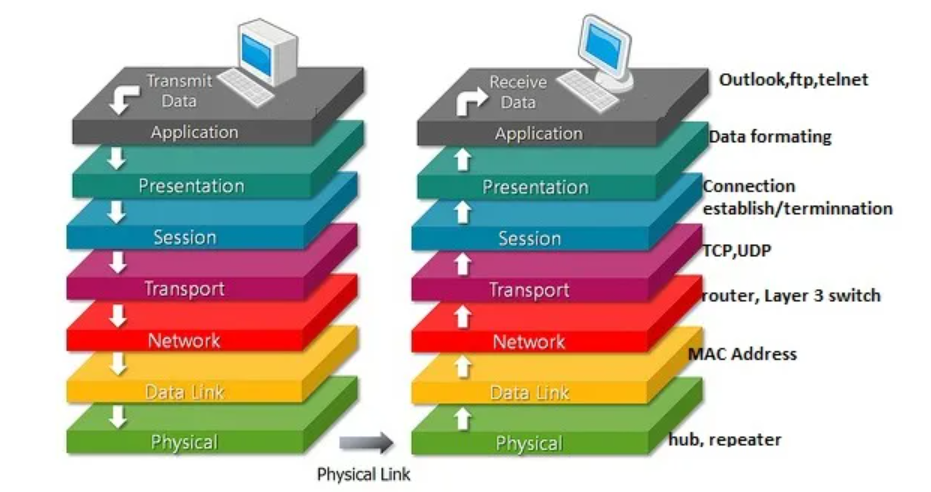
Слоеная архитектура – процесс постепенной передачи данных

Почитать: Open System Interconnection

<https://everybit.cloud/open-systems-interconnection-osi-model/>

Выделяется 7 слоев передачи данных, от физического до уровня приложения





Общение между двумя приложениями. Первое начинает отправлять, т.е. сигнал доходит до физического уровня, заходит в другое приложение в виде сигнала. К ПК подключен либо оптический кабель, либо железный кабель, либо Wi-Fi антенна. Когда физический уровень этот сигнал принял, преобразовал в 0 и 1, далее они собираются в байты, затем в символы, потом в протоколы HTTP, протоколы собираются уже в файлы, и т.д.

Т.е. это такое представление о том, как происходит преобразование данных от природного вида к более абстрактному.

Этот стандарт описывает словами идею, которая в принципе архитектура (структура программы) строится так, что каждый слой, пропуская через себя повышает абстракцию от какой-то конкретной формы к более абстрактной формы.

Еще раз:

В нашем проекте контекст данных – отдельный архитектурный слой проекта, который преобразовывает данные из источника данных (БД, API, файловое хранилище) в программную форму – объекты.

Задача DataContext - обеспечить как-то взаимодействие с источником данных и все, что приходит подать в виде объектов, коллекции объектов и т.д. Т.е. преобразовать в программную форму.

Поэтому выход в виде объектов делает этот слой заменяемым. Т.е., если мы переходим к другому источнику данных, все, что нужно сделать – это привести эту среду к тем самым объектам. В этом случае мы просто один слой другим, он нам выдает те же объекты, просто заполняет их из другого источника.

В классическом MVC именно такой слой называется моделью.

В ASP понятие модели несколько смещается в то время когда он был не только MVC и модель – это скорее те самые объекты, которые выходят из Модели (с большой буквы), поэтому для всего архитектурного слоя используется «Контекст данных».

Контекст состоит из нескольких частей:

* Entities – сущности (модели данных, DTO – Data transfer Objects) – объекты (классы), предназначенные для отображения данных в программу. У них фактически нет методов, они несут только данные. Их данные относительно простые, отображают СУБД данные, они должны быть сериализуемыми.
* DAO (Data Access Objects) – объекты-сервисы с методами, предназначенными для работы с данными.
* Context (DAL - Data Access Layer) – «центральный представитель» этого слоя, который собирает в себе образ (представление, отображение) источника данных (ORM - Object Relation Mapping)

(Entity) User {Id, Name, ...}

(Entity) Product {Id, Name, ...}

(Context) DataContext {Users, Products, ....}

(DAO) UserDao {authenticate(login,password), getCart()}

Т.е.

Entity – просто отображает данные

Context – отображает коллекции данных, связи между ними

DAO – добавляет логику конкретной сущности. Иногда этот слой не создают и прямо через Контекст дают доступ в контроллерах, но это плохая практика, потому что при переходе на другой источник данных нужно будет вносить изменения

Создаем в проекте папки:

* На уровне проекта – Data
* Внутри Data: Entities, Context, DAL

В Entities добавляем класс User (public!)

Свойства:

public Guid Id { get; set; }

UUID – стандарт идентификации и многие СУБД напрямую поддерживают работу с этим стандартом. Нужно следовать стандарту, поэтому Id должно быть типа **Guid (Global unique identificatory),** не будет существовать двух одинаковых единиц с одинаковым идентификатором. Размера хватит – 128 бит

public class User

{

public Guid Id { get; set; }

public string Name { get; set; } = null!;

public string Email { get; set; }=null!;

public string AvatarUrl { get; set; } = null!; //работа с файлами (изображениями)

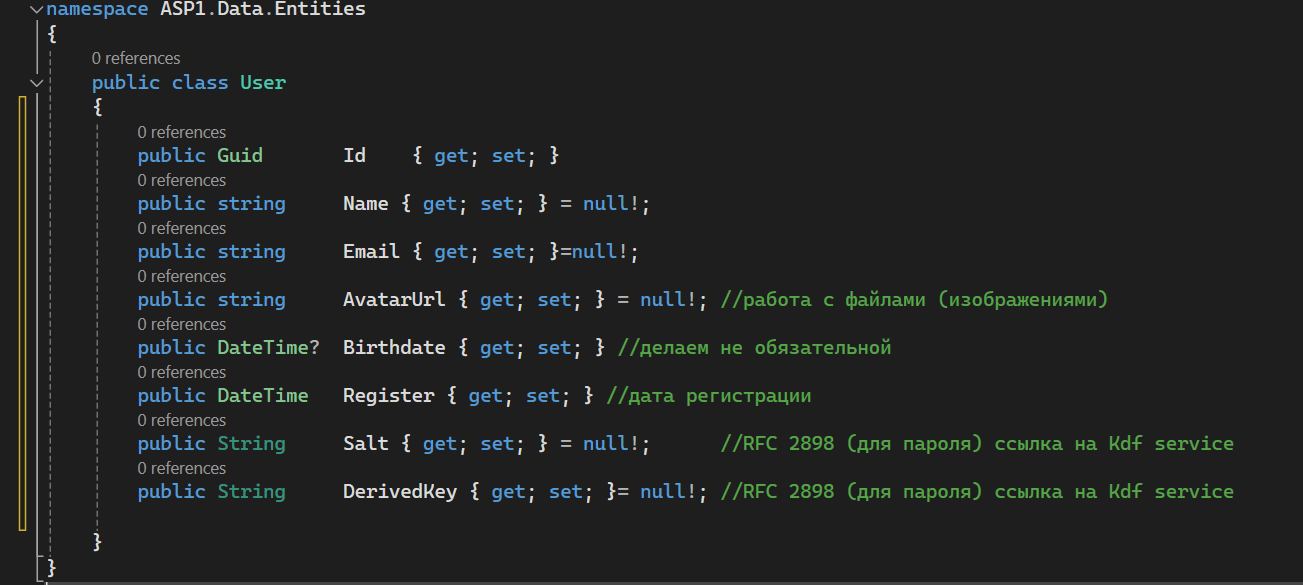
public DateTime? Birthdate { get; set; } //делаем не обязательной

public DateTime Register { get; set; } //дата регистрации

public String Salt { get; set; } = null!; //RFC 2898 (для пароля) ссылка на Kdf service

public String DerivedKey { get; set; }= null!; //RFC 2898 (для пароля) ссылка на Kdf service

}



В папке Сontext создаем класс DataContext, который будет собирать разные сущности

using ASP1.Data.Entities;

using Microsoft.EntityFrameworkCore;

namespace ASP1.Data.Context

{

public class DataContext:DbContext

{

public DbSet<User> Users { get; set; }

//нужно перегрузить конструктор, который принимает options и передаем его в базовый конструктор, при этом по сути он ничего не делает, дополнительных действий не делает, но передает эти options

//Подключение к реальной БД хотя и является свойством контекста данных, но его параметры традиционно сохраняются в файлах конфигурации всего проекта (appsettings.json).

Соответственно, передача этих параметров будет осуществляться из Program.cs, где подключаются эти файлы

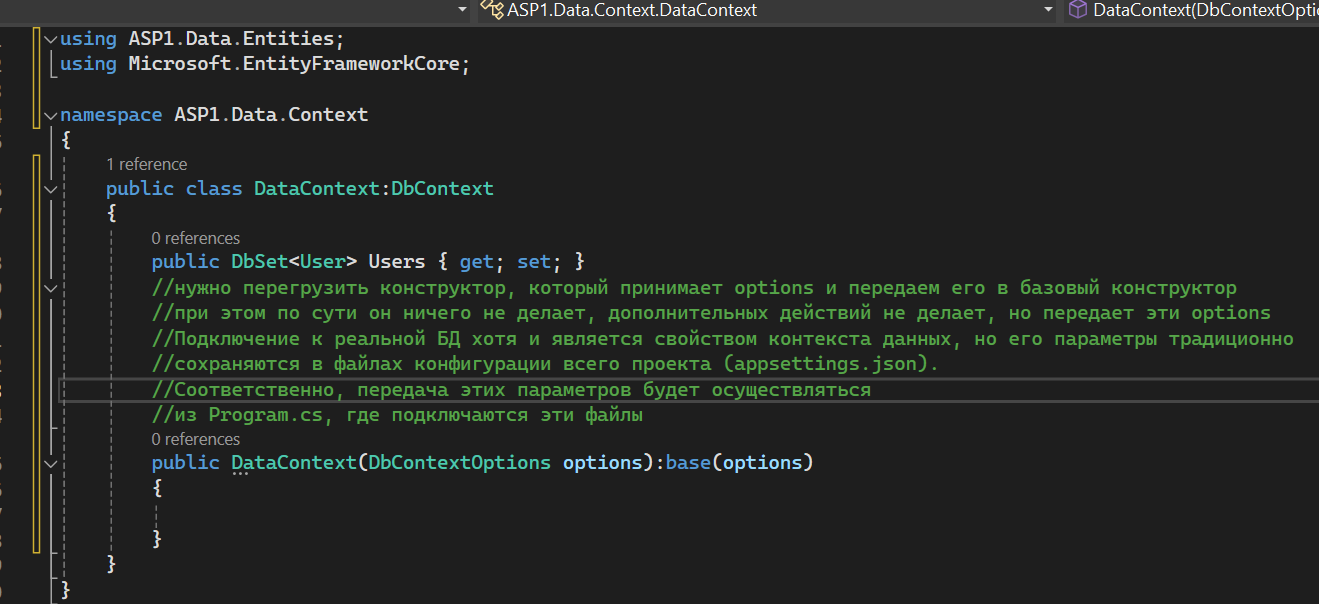
public DataContext(DbContextOptions options):base(options)

{

}

}

}



При изучении ADO прямо тут реализовывали подключение к БД и, в принципе, это правильно, но когда мы говорим, что проект может куда-то переноситься, проект имеет свою конфигурации, то нужно или сюда передать всю конфигурацию, что не очень правильно, или в момент старта будет решаться, к какой БД подключаться и уже это решение сюда будет спущено для подключения. Т.е. контекст вроде и сам мог бы подключить к БД, но в силу структуры проекта мы это переносим в другое место.

Идем в файл appsettings.json и прописываем в нем строки подключения

EntityFramework можно показать строку подключения или к пустой БД, или даже к несуществующей БД, если он имеет права их создавать.

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"MsSQL": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=ClassWorkAsp;Integrated Security=True"

}

}

Т.е. можно указать ему несуществующую БД и он сам ее создаст, для этого берем любую строку подключения, меняем название в графе Catalog

Дальше идем в Program.cs

Тут нужно прописать ДО!!! строки var app = builder.Build();

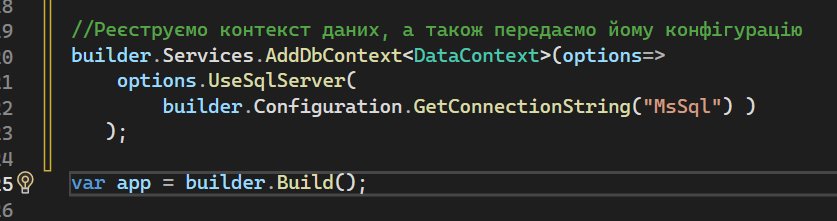
//Реєструємо контекст даних, а також передаємо йому конфігурацію

builder.Services.AddDbContext<DataContext>(options=>

options.UseSqlServer(

builder.Configuration.GetConnectionString("MsSql") )

);



Возращаемся в DataContext

Связи между данными (Relations) указываются в специальном методе

override On: выдает два варианта:

OnConfiguring(DbContextOptionsBuilder optionsBuilder) – мы прокинули optionBuilder через конструктор, т.е. конфигурин прошел через конструктор

OnModelCreating(ModelBuilder modelBuilder) – выбираем его

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//Укажем UNIQUE ограничения для E-mail, которое используется вместо логина

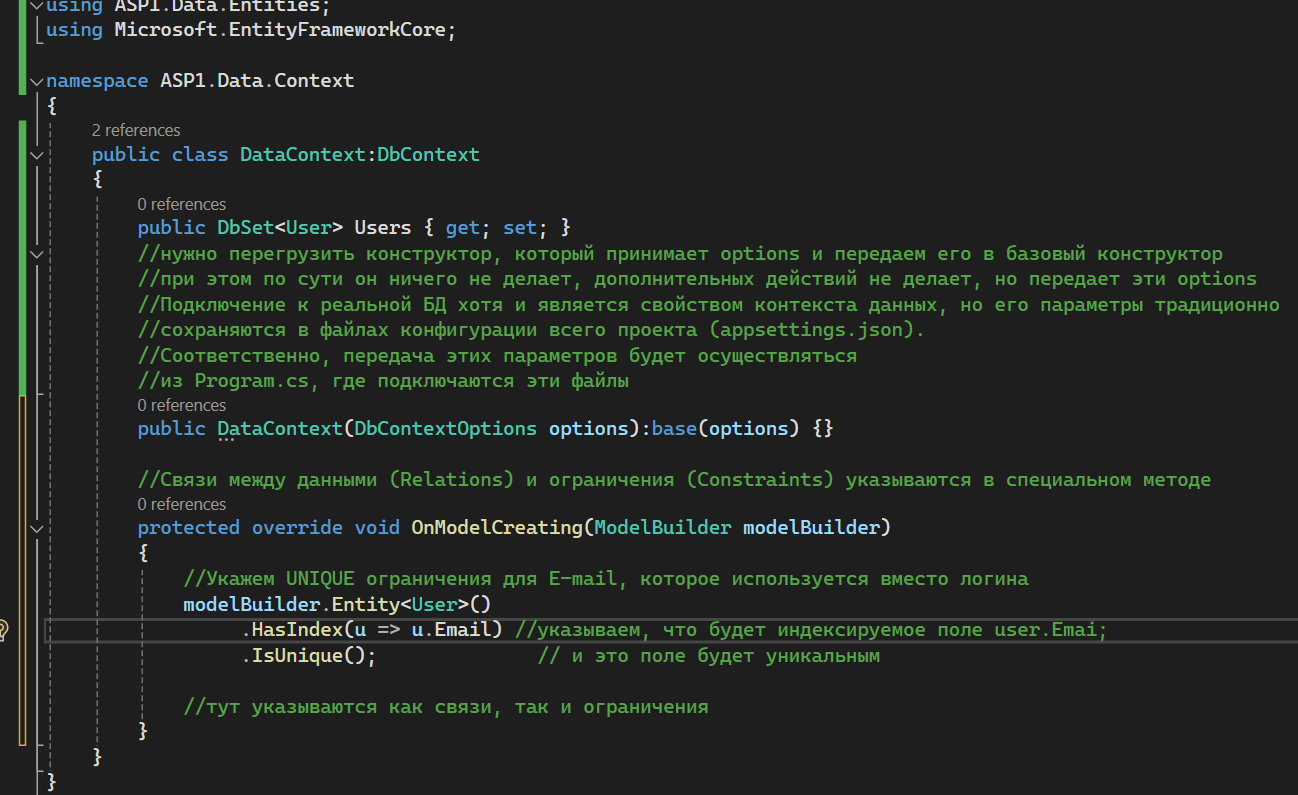
modelBuilder.Entity<User>()

.HasIndex(u => u.Email) //указываем, что будет индексируемое поле user.Emai;

.IsUnique(); // и это поле будет уникальным

//тут указываются как связи, так и ограничения

}



**Итого:**

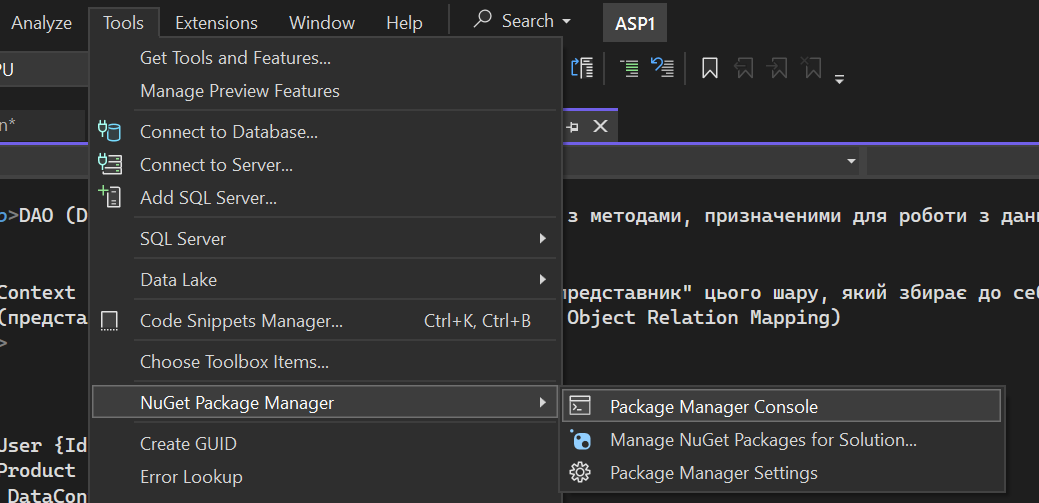
**Entity – одинокий объект**

**Context – совокупность Entity и связей между ними, а также Constaint (органичений), которые накладываются на эти Entity**

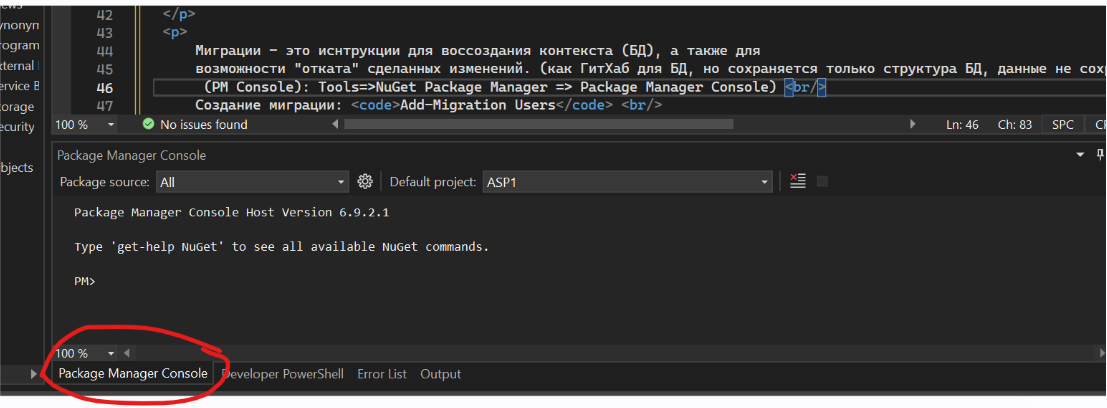
Миграции - это иснтрукции для воссоздания контекста (БД), а также для возможности "отката" сделанных изменений. (как ГитХаб для БД, но сохраняется только структура БД, данные не сохраняются в миграции)

Миграции - это иснтрукции для воссоздания контекста (БД), а также для возможности "отката" сделанных изменений. (как ГитХаб для БД, но сохраняется только структура БД, данные не сохраняются в миграции)<br/>

**(PM Console): Tools=>NuGet Package Manager => Package Manager Console**

****

ЛИБО



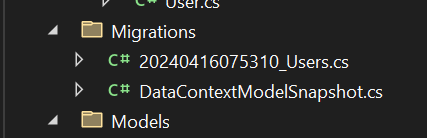
Создание миграции через команду в PM Consile: Add-Migration Users

*Users – название миграции*

По результату должна быть создана директория Migrations и файл с соответстующей миграцией.

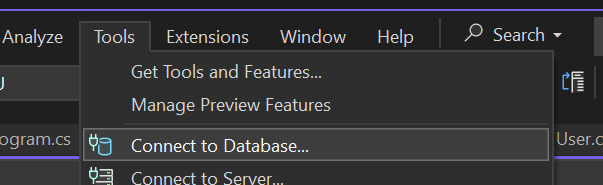


Сначала билдится проект, потом в его метаданных находится DbContext, запускается анализ DbContex и добавляется миграция:

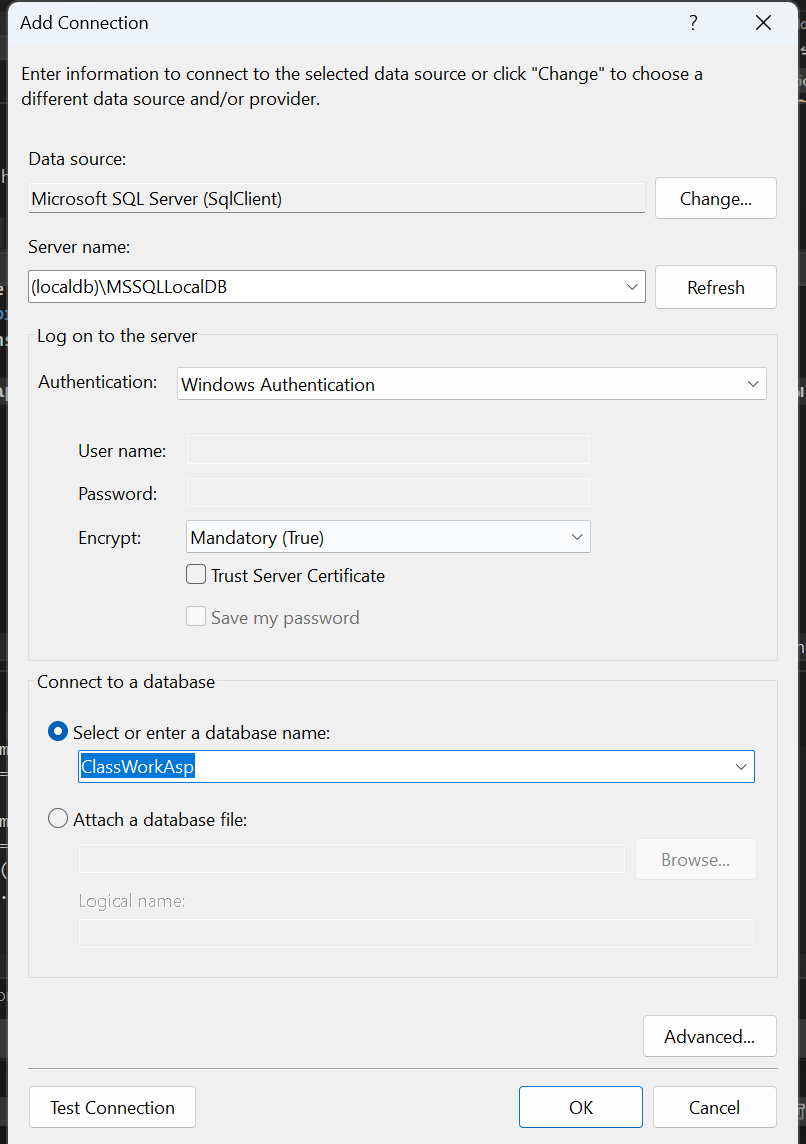


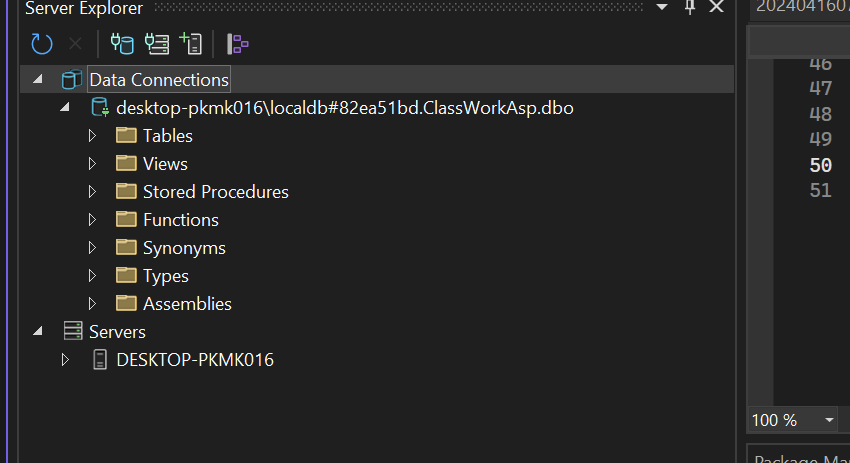
Примeнение миграции: Update-Database (команда в PM Console)

Контроль БД: Tools=>Connect to DataBase (до этого из appsettings скопировать название сервера (localdb)\\MSSQLLocalDB, чтобы руками не вводить)



При копировании имени сервера УБРАТЬ ОДИН СЛЕШ!





НА ЭТОМ БАЗА ДАННЫХ ПОДКЛЮЧЕНА

НОВАЯ ТЕМА: **РЕГИСТРАЦИЯ ПОЛЬЗОВАТЕЛЯ**

В проекте заходим в папку Views => Shared => Layout.cshtml

Добавляем ссылку для входа <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="SignIn">Sign in</a>

Идем в HomeController

Добавляем action

public ViewResult SignIn()

{

return View();

}

Идем в папку View=>Home=>Add=>View=>Razor Empty=>SignUp.cshtml

@{

ViewData["Title"] = "Sign Up";

}

<div class="text-center">

<h1 class="display-4">Реєстрація нового користувача</h1>

</div>

<form asp-action="SingUp" method="post" enctype="multipart/form-data">

<div class="row">

<div class="col">

<div class="input-group mb-3">

<span class="input-group-text" id="username-addon1"><i class="bi bi-person-fill"></i></span>

<input type="text" class="form-control" placeholder="Введіть ім'я"

name="signup-username" aria-label="Username" aria-describedby="username-addon1">

</div>

</div>

<div class="col">

<div class="input-group mb-3">

<span class="input-group-text" id="user-email-addon1"><i class="bi bi-person-fill"></i></span>

<input type="text" class="form-control" placeholder="Електронна пошта"

name="signup-email" aria-label="User-email" aria-describedby="user-email-addon1">

</div>

</div>

</div>

<div class="row">

<div class="col">

<div class="input-group mb-3">

<label class = "input-group-text" for="signup-avatar"><i class="bi bi-person-bounding-box"></i></label>

<input type="file" class="form-control" name="signup-avatar" id="singup-avatar"/>

</div>

</div>

<div class="col">

<div class="input-group mb-3">

<span class="input-group-text" id="birthdate-addon1"><i class="bi bi-cake"></i></span>

<input type="date" class="form-control" placeholder="Дата народження"

name="signup-birthdate" aria-label="Birthdate" aria-describedby="birthdate-addon1">

</div>

</div>

</div>

<div class="row">

<div class="col">

<div class="form-check">

<input name="signup-confirm" class="form-check-input" type="checkbox" value="" id="flexCheckDefault">

<label class="form-check-label" for="flexCheckDefault">

Погоджуюсь із правилами сайту

</label>

</div>

</div>

<div class="col">

<button type="button" class="btn btn-primary">Реєстрація</button>

</div>

</div>

</form>

**ИДЕМ в Models=>Home=>Add new Folder=>SingUp=>Add class=>SingUpFormModel.cs**

public class SignUpFormModel

{

[FromForm(Name = "signup-username")]

public String UserName { get; set; }

[FromForm(Name = "signup-email")]

public String UserEmail { get; set; }

[FromForm(Name = "singhup-avatar")]

public IFormFile AvatarFile { get; set; } = null!;

[FromForm(Name = "signup-birthdate")]

public DateTime? Birthdate { get; set; }=null!;

[FromForm(Name = "signup-confirm")]

public bool Confirm { get; set; }

}

В папке Sing Up=>Add new class=> SignUpPageModel (для представления)

public class SignUpPageModel

{

public String? Message { get; set; }

public bool? IsSuccess { get; set; } //удачно/не удачно

public SignUpFormModel? signUpFormModel { get; set; }

}

Когда мы первый раз заходим на форму – никаких месседжей нет, поэтому поле с формой может быть пустым

Далее реализуем оборот. Идем в HomeController

//принял ФормМодель, обработал, перекинул на ВьюМодель и передал на вьюшку

public ViewResult SignUp(SignUpFormModel? formModel)

{

SignUpPageModel pageModel = new()

{

signUpFormModel = formModel

};

return View(pageModel);

}

Идем в SingUp.cshtml и заявляем, что к нам приходит модель

@model ASP1.Models.Home.SignUp.SignUpPageModel

Делаем проверку

@if(Model.IsSuccess!=null)

{

<div class="@(Model.IsSuccess ?? false ? "alert alert-success":"alert alert-danger")" role="alert">

@Model.Message

</div>

}

**Валидация формы (пишем метод в HomeController**

private Dictionary <String, String> \_ValidateSingUpModel(SignUpFormModel? formModel)

{

Dictionary<String, String> res = new(); //перелік помилок по кожному з полів моделі

if (formModel == null) {

res[nameof(formModel)] = "Model is null";

}

else

{

if (String.IsNullOrEmpty(formModel.UserName))

{

res[nameof(formModel.UserName)] = "Name is empty";

}

if (String.IsNullOrEmpty(formModel.UserEmail))

{

res[nameof(formModel.UserEmail)] = "Email is empty";

}

//Проверить, присутствует ли почта уже в БД

}

return res;

}

Тут останавливаемся и идем в папку Data=>Dal=>Add class=>UserDao.cs

public class UserDao

{

private readonly DataContext \_context;

public UserDao(DataContext context)

{

\_context = context;

}

public bool IsEmailFree(String email)

{

return \_context

.Users

.Where(u => u.Email == email)

.Any();

}

}

Создаем в этой же папку класс-аксессор (будет один на всех, помогает избежать дополнительных инжекций)

//сервіс - реєструємо у Program.cs

public class DataAccessor

{

public DataContext \_context;

public UserDao UserDao { get; private set; }

public DataAccessor(DataContext context)

{

\_context = context;

UserDao = new(\_context);

}

}

**Идем в Program.cs и региструрием**

builder.Services.AddSingleton<DataAccessor>();

Теперь каждый может его инжектировать. А он в свою очередь может инжектировать DbContext, который тоже зарегистрирован в Program.cs

Дальше мы работаем с DataAccessor

Идем в Контроллер и выполняем инжекцию

private readonly DataAccessor \_dataAccessor;

Правой кнопкой=>QuickActions=>Add parameter – автоматически добавляет поле как параметр в конструктор HomeController

Теперь вместо инжекций отдельных DAO мы используем DataAccessor

ВОЗВРАЩАЕМСЯ ниже в HomeController – дописываем метод валидации

private Dictionary <String, String> \_ValidateSingUpModel(SignUpFormModel? formModel)

{

Dictionary<String, String> res = new(); //перелік помилок по кожному з полів моделі

if (formModel == null) {

res[nameof(formModel)] = "Model is null";

}

else

{

if (String.IsNullOrEmpty(formModel.UserName))

{

res[nameof(formModel.UserName)] = "Name is empty";

}

if (String.IsNullOrEmpty(formModel.UserEmail))

{

res[nameof(formModel.UserEmail)] = "Email is empty";

}

//Проверить, присутствует ли почта уже в БД

if(\_dataAccessor.UserDao.IsEmailFree(formModel.UserEmail)) {

res[nameof(formModel.UserEmail)] = "Email in use";

}

//Соглашение с условиями

if(!formModel.Confirm)

{

res[nameof(formModel.Confirm)] = "Confirm expeced";

}

}

return res;

}

Мы реализовали перечень ошибок, который нужно переставить в PageModel, чтобы потом их отображить.

Идем в SingUpPageModel

Добавляем поле

public Dictionary<String, String> ValidationErrors { get; set; } = null!;

Идем в HomeController

public ViewResult SignUp(SignUpFormModel? formModel)

{

SignUpPageModel pageModel = new()

{

signUpFormModel = formModel,

ValidationErrors=\_ValidateSingUpModel(formModel)

};

return View(pageModel);

}

Дописываем валидацию в HomeController

public ViewResult SignUp(SignUpFormModel? formModel)

{

SignUpPageModel pageModel = new()

{

signUpFormModel = formModel,

ValidationErrors=\_ValidateSingUpModel(formModel)

};

//проверяем, есть ли какие-то ошибки и есть переданные данные

if(formModel?.UserEmail!=null) //есть переданные данные

{

if(pageModel.ValidationErrors.Count > 0) //и ошибки)

{

pageModel.Message = "Реєстрацію відхилено";

}

else

{

//регистрируем пользователя

pageModel.Message = "Реєстрація успішна";

}

}

return View(pageModel);

}

//на входе модель, на выходе - сущность Entity

private Data.Entities.User mapUser(SignUpFormModel formModel)

{

String salt = Guid.NewGuid().ToString();//у нас есть сервис для соли, используем его!

Data.Entities.User user = new()

{

Id = Guid.NewGuid(),

Name = formModel.UserName,

Email = formModel.UserEmail,

Register = DateTime.Now,

DerivedKey = \_kdfService.GetDerivedKey("123", salt)

};

}

Идем в UserDao и дописываем метод

public void SignUpUser(User user)

{

\_context.Users.Add(user);

\_context.SaveChanges();

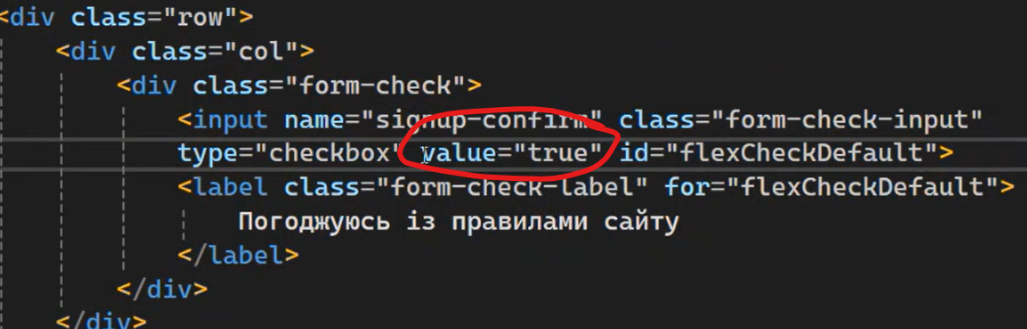
}

Возвращаемся в HomeController и дописываем функционал регистрации пользователя

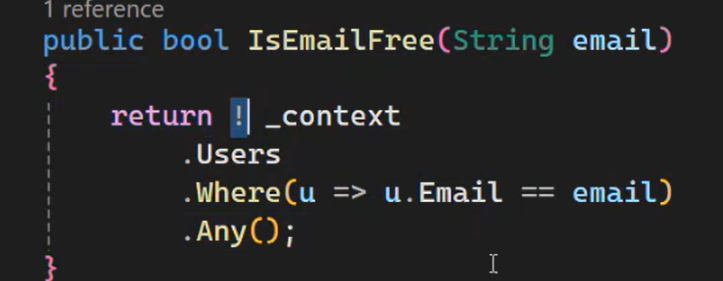
\_dataAccessor.UserDao.SignUpUser(mapUser(formModel));

ИСПРАВЛЯЕМ ОШИБКИ:

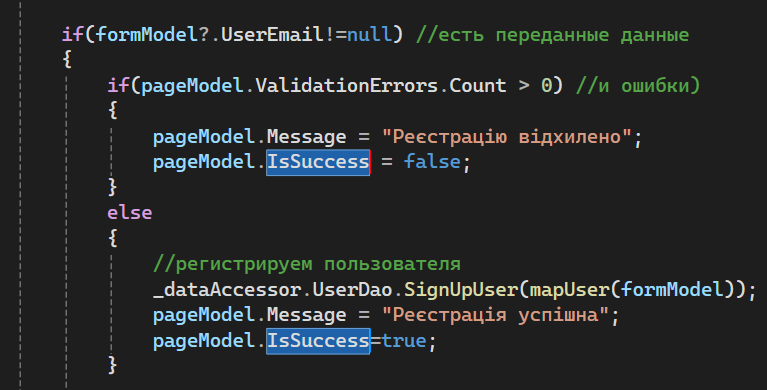
1. В файле SignUp.cshtml для checkbox ставим value=true



1. В файле Userdao сделали инверсию



1. В HomeController добавили IsSuccess



Комбинация клавиш для выделения следующих вхождений слова: Alt+Shift+точка (.)

ЕСЛИ ЧТО-ТО Меняем в структуре БД, нужно обновить МИГРАЦИЮ

Add-Migration name

Update-Migration

ПРОПУСТИЛИ ПАРОЛИ! В SignUp.cshtml:

<div class="row">

<div class="col">

<div class="input-group mb-3">

<span class="input-group-text" id="password-addon1"><i class="bi bi-lock"></i></span>

<input type="text" class="form-control"

name="signup-password"

aria-label="Username"

placeholder="Вигадайте пароль"

aria-describedby="password-addon1">

</div>

</div>

<div class="col">

<div class="input-group mb-3">

<span class="input-group-text" id="user-repeat-addon1"><i class="bi bi-unlock"></i></span>

<input type="password" class="form-control"

name="signup-repeat"

aria-label="Repeat Password"

placeholder="Повторіть пароль"

aria-describedby="user-repeat-addon1">

</div>

</div>

</div>

Идем в Models=>Home=>SignUp=>SignUpFormModel и добавляем в форму свойства для пароля

[FromForm(Name = "signup-password")]

public String Password { get; set; } = null!;

[FromForm(Name = "signup-repeat")]

public String PasswordRepeat { get; set; }= null!;

Затем в Контроллере нужно в валидаторе нужно это проверить, добавляем два блока:

if (String.IsNullOrEmpty(formModel.Password))

{

res[nameof(formModel.Password)] = "Password is empty";

}

if (formModel.Password!=formModel.PasswordRepeat)

{

res[nameof(formModel.PasswordRepeat)] = "Password mismatch";

}

Далее идем в сам алгоритм регистрации mapUser

Вместо «123» подставляем реальный пароль

DerivedKey = \_kdfService.GetDerivedKey(formModel.Password, salt)

**13.04.2024**

Аутентификация – вход в систему, удержание этой аутентификации, создание ситуации, при которой нет необходимости повторного ввода пароля и технология, как это реализовать, которая называется MiddleWere.

Первое, вход в систему мы уже заложили рисунок Sign In. Разместив его на Layout мы имеем доступ к нему из любой страницы и соответственно, начинаем реализовывать.

Для входа в систему достаточно меньшей формы, чем для регистрации. Только вывести то, что для этого нужно: мейл и пароль. Возьмем Bootstrap элементы, которые называются модальные окна и реализуем вход в систему в виде таких модальных окон.

<https://getbootstrap.com/docs/5.1/components/modal/> (ЕСТЬ ДЕМО)

Модальные окна – замена стандартным Алертам, которые позволяют выводить сообщения, выводить кнопки. Они могут закрываться сами, так и кнопками.

Копируем код и переносим на наш Layout (Views=>Shared=>Layout)

Копируем Live demo (основной пример), в котором нужно просто поменять один из стилей для центровки посередине.

<!-- Button trigger modal -->

<button type="button" class="btn btn-primary" data-bs-toggle="modal" data-bs-target="#exampleModal">

Launch demo modal

</button>

<!-- Modal -->

<div class="modal fade" id="exampleModal" tabindex="-1" aria-labelledby="exampleModalLabel" aria-hidden="true">

<div class="modal-dialog">

<div class="modal-content">

<div class="modal-header">

<h5 class="modal-title" id="exampleModalLabel">Modal title</h5>

<button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>

</div>

<div class="modal-body">

...

</div>

<div class="modal-footer">

<button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Close</button>

<button type="button" class="btn btn-primary">Save changes</button>

</div>

</div>

</div>

</div>

Чтобы кнопка запускала модальное окно, она должна иметь вот эти два атрибута:

data-bs-toggle="modal"

data-bs-target="#exampleModal"

Переносим эти атрибути на ссылку <a class="nav-link text-dark" data-bs-toggle="modal" data-bs-target="#exampleModal**"(ЭТО ТРИГГЕР!)>**Sign in</a>

Чтобы красиво работал курсор (был указателем) добавляем атрибут href="#"

<a class="nav-link text-dark" href="#" data-bs-toggle="modal" data-bs-target="#exampleModal">Sign in</a>

Меняем в идентификатере «ехample» на свое название (во всех местах вхождения). Выделяем с помощью Alt+Shift+. (точка)

В теле модального окна меняем тайтл, потом идем в SingUp и берем пару инпутов (емейл, пароль)

Поскольку Layout на каждой странице, в том числе и на странице регистрации, айдишки, которые мы скопировали, начнут дублироваться, поэтому их нужно попереименовывать

Форму в модальном окне не используем, поэтому в input name не нужно, меняем его на id

Сделаем асинхронно в стиле AJAX.

Идем в wwwroot=>js=>site.js

Желательно работать в специальном обработчике событий DOMContentLoaded, что гарантирует, что все элементы уже существуют и можно их искать,можно с ними работать

document.addEventListener('DOMContentLoaded', () => {

})

document.addEventListener('DOMContentLoaded', () => {

const authButton = document.getElementById("auth-button");

//если нашли кнопку, к ней прикрепляем обработчик события

//если мы вошли в систему, этой кнопки не будет

if (authButton) authButton.addEventListener('click', authButtonClick);

});

function authButtonClick(){

const authEmail = document.getElementById("auth-email");

if (!authEmail) throw "Element '#auth-email' not found";

const authPassword = document.getElementById("auth-password");

if (!authPassword) throw "Element '#auth-password' not found";

const authWarning = document.getElementById("auth-warning");

if (!authWarning) throw "Element '#auth-warning' not found";

const email = authEmail.value.trim(); //убираем лишние пробелы

const password = authPassword.value; //тут не убираем пробелы, пароль может содержать пробелы

if (email == "") {

authWarning.classList.remove('visually-hidden');

authWarning.innerText = "Введіть e-mail";

return;

}

if (password == "") {

authWarning.classList.remove('visually-hidden');

authWarning.innerText = "Введіть пароль";

return;

}

//Запитуємо сервіс автентифікації

}

Дальше создаем новый контроллер папка Controllers=>Add=>Controller=>API (Common)=>API Controller – Empty=>Add=>AuthController (название пишем сами)

Контроллеры API и MVC

|  |  |
| --- | --- |
| MVC | API |
| Разные методы (action), действия запускаются разными адресами | Разные действия запускаются одним адресом, но разными методами запроса |
| /Home/Index  /Home/Privacy | Этот адрес указывается перед контроллером в виде атрибута  [Route("api/[controller]")]  Но рекомендуется вместо [controller] указать конкретно auth (хотя это одно и тоже)  GET /api/auth  POSR /api/auth |
| действия вовращают предсталения или другие IActionResult | Действия возвращают объекты, которые ASP преобразовывает в JSON или коллекции (обобщенные объекты) |

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

namespace ASP1.Controllers

{

[Route("api/auth")]

[ApiController]

public class AuthController : ControllerBase

{

[HttpGet]

public object Get(String email, String password) //название метода роли не играет, главное - атрибут

{

return new { status = $"Auth works: {email}, {password}" };

}

}

}

В файле site.js

//Запитуємо сервіс автентифікації

fetch(`/api/auth?email=${email}&password=${password}`) //идет запрос в AuthController /api/auth=>AuthController (передаются параметры, которые мы вытащили из формы)

.then(r => r.json()) // return new { status = $"Auth works: {email}, {password}" };

.then(j => { //приходит JSON (j={status ="..."})

console.log(j);

})

Джаваскрипт не строго типизированный, и мы не можем быть уверены, что оно нам даст, например, мы берем GetElementById, можно ли сказать, что у него есть classlist или value? Нельзя value есть только у input, Джаваскрипт не знает, что ему придет

Переменные могут меняться на лету, поэтому синтаксис нужно проверять вручную. Можно перейти на TypeScript, он в этом плане лучше, но он уменьшает возможности Джаваскрипт, связанные с отсутствием строгой типизации

Например, можно сделать свой метод

Изображение выглядит как текст, Шрифт, снимок экрана, Графика

Автоматически созданное описание

И тогда console.Log будет тоже правильно, компилятор не может сам проверить, есть ли такой метод или нет

Это и большой плюс Джаваскрипта, что мы можем создать абсолютно любой объект, но это и источник ошибок, потому что компилятор ничего не скажет, а в процессе выполнения могут возникнут ошибки

[Route("api/auth")]

[ApiController]

public class AuthController : ControllerBase

{

[HttpGet]

//fetch отправляет данные

public object Get(String email, String password) //название метода роли не играет, главное - атрибут

{

return new { status = $"Auth works: {email}, {password}" }; //тут создается объект и возвращается в формате json и передается на ответ

}

}

Дальше нужно проанализировать логин и пароль и дать вывод: пускаем/не пускаем

Чтобы это сделать, мы должны обратиться к нашему DataAccessor и в нем предусмотреть соответствующий метод

В AuthController делаем инжекцию DataAccessor

private readonly DataAccessor \_dataAccessor;

По имени поля правой кнопкой – генерируем конструктор

public AuthController(DataAccessor dataAccessor)

{

\_dataAccessor = dataAccessor;

}

Дальше идем в составляющую DataAccessor – UserDao и добавляем еще один метод

Сначала инжектируем IIkfService

private readonly IKdfService \_kdfService;

Добавляем его в конструктор через Quick Actions

public User? Authenticate(String email, String password) //будет возвращен либо найдено, либо null

{

User? user=\_context.Users.FirstOrDefault(u => u.Email == email);

//сначала нужно найти пользователя по логину, соль и потом проверить соответствие пароля

//поэтому задачу разделяем, сначала ищем пользователя (логины уникальны)

//пароль, который нам пришел, смешиваем с солью, которая лежит у пользователя, повторяем генерацию ключа

//и проверяем, соответствуют ли пароли

//наоборот сделать нельзя, то есть нельзя, зная ключ, сделать из него назад пароль, в этом и состоит суть защиты механизма

//проверить можем, а подделать или подобрать - нет

if (user != null && \_kdfService.GetDerivedKey(password, user.Salt) == user.DerivedKey)

{

return user;

}

return null;

}

Затем нужно внести соответствующие измения в DataAcessor

public class DataAccessor

{

public DataContext \_context;

private readonly IKdfService \_kdfService;

//DataAcessor через контейнер получает все зависимости и всереди уже перераспределяет между теми

//Dao, которые в них нуждаются

public UserDao UserDao { get; private set; }

public DataAccessor(DataContext context, IKdfService kdfService)

{

\_context = context;

\_kdfService = kdfService;

UserDao = new(\_context, \_kdfService);

}

}

В AuthController проверяем

public object Get(String email, String password) //название метода роли не играет, главное - атрибут

{

var user=\_dataAccessor.UserDao.Authenticate(email, password);

String status;

if (user == null)

{

status = "error";

}

else

{

status = "success";

//return new { status = $"Auth works: {email}, {password}" }; //тут создается объект и возвращается в формате json и передается на ответ

}

return new { status };

}

Возвращаемся на site.js

//Запитуємо сервіс автентифікації

fetch(`/api/auth?email=${email}&password=${password}`) //идет запрос в AuthController /api/auth=>AuthController

.then(r => r.json()) // return new { status = $"Auth works: {email}, {password}" };

.then(j => { //приходит JSON (j={status ="ok/error"}), мы уже знаем, что приходит, а значит, можем сделать проверку

console.log(j);

if (j.status == "success") {

//зміна статусу автентифікації вимагає перезавантаження сторінки (ресурсу)

window.location.reload();

}

else {

authWarning.classList.remove('visually-hidden');

authWarning.innerText = "Вхід відхилено, перевірте дані";

}

})

КАК ЗАПОМНИТЬ ДАННЫЕ? С учетом того, что после успешного входа страница перезагружается, но нужно, чтобы сохранились данные от предыдущей работы. Куда их нужно так сохранить, чтобы они пережили обновление?

Для этого есть понятие сессия

HTTP-сессии – средства для сохранения данных между разными запросами.

Когда приходит запрос. Контроллер создается по новой и все, что было создано предыдущим запросом не доступно, так называемый request scope, который пересоздает все объекты с каждым новым запросом. Это сильно усложняет процедуру вот таких «запоминаний» и для реализации этого используются сессии.

Сессии требуют настройки, детали в документации <https://learn.microsoft.com/ru-ru/aspnet/core/fundamentals/app-state?view=aspnetcore-8.0>

Настройки сессии осуществляются через файл Program.cs и в него нужно сделать несколько вставок (зеленым указаны три момента, которые нужно вставить на странице выше)

Изображение выглядит как текст, Мультимедийное программное обеспечение, программное обеспечение, снимок экрана

Автоматически созданное описание

//Настройка Http-сессии

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(options =>

{

//главная настройка - время бездеятельности, т.е. если в течение 10 секунд мы ничего не делаем с сайтом, сессия отменяется

options.IdleTimeout = TimeSpan.FromSeconds(10);

options.Cookie.HttpOnly = true;

options.Cookie.IsEssential = true;

});

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

//Включение Http-сессии

app.UseSession();

Эта технология позволяет сохранять данные и потом их воспроизводить (почитать)

После включения сессии доступны через HTTPContext везде, где этот контекст существует

Дальше дописываем сохранение сессии в AuthController

[Route("api/auth")]

[ApiController]

public class AuthController : ControllerBase

{

private readonly DataAccessor \_dataAccessor;

public AuthController(DataAccessor dataAccessor)

{

\_dataAccessor = dataAccessor;

}

[HttpGet]

//fetch отправляет данные

public object Get(String email, String password) //название метода роли не играет, главное - атрибут

{

var user=\_dataAccessor.UserDao.Authenticate(email, password);

String status;

if (user == null)

{

status = "error";

}

else

{

status = "success";

//return new { status = $"Auth works: {email}, {password}" }; //тут создается объект и возвращается в формате json и передается на ответ

//сохраняем в сессию данные автентификации (как набор байт, объекты нужно сериализовать), с

HttpContext.Session.SetString("auth-user-id", user.Id.ToString()); //"auth-user-id" - ключ и сам идентификатор пользователя

}

return new { status };

}

}

Для демонстрации в Layout (в Razor HttpContext доступен просто как Context)

<footer class="border-top footer text-muted">

<div class="container">

@if(Context.Session.GetString("auth-user-id")!=null)

{

<b>Авторизований режим</b>

}

&copy; 2024 - ASP1 - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>

</div>

</footer>

Если в сессии что-то лежит, значит, автентификация прошла успешно и мы авторизированны

Cессия существует на всех адресах, и HomeController, PrivacyController видят эту сессию

Писать это все время не совсем, по хорошему, этот String нужно еще проверить, действительно ли это user.id, если мы это везде будем проверять, то размажем код, желательно это вынести в единое место, которое будет доступно для всех частей проекта. Т.е. нужно это куда-то перенести, углубить эту проверку

**Новая тема: MiddleWare -**

https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0

Идея MiddleWear – создание прослоек, которые через себя пропускают как request, так и response. Он находится в середине между Запросами и Контроллерами, также мы можем другие MiddleWear встраивать в MiddleWear, но не можем поставить раньше стандартных MiddleWear

Изображение выглядит как текст, снимок экрана, дизайн, смартфон

Автоматически созданное описание

, но не можем поставить раньше стандартных MiddleWear

Изображение выглядит как текст, линия, диаграмма, программное обеспечение

Автоматически созданное описание

MiddleWear вставляется специальными командами, которые вставляются в Program.cs

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Это команды use+что-то

Что-то по дефолту подключается, что-то прямо сами прописываем

Например, сессионный MiddleWear, который мы подключили

Изображение выглядит как текст, снимок экрана, Шрифт, Графика

Автоматически созданное описание

Прикол в том, что через MiddleWear проходят все запросы до того как они будут распределены на Контроллер. Тобто через MiddleWear пройдет и на APIController, MVCController

То есть, если нужно какую-то работу выполнить для всех запросов, независимо от их ардес, методов, и т.д., то мы делаем MiddleWear, в котором это реализуем. Соответственно, мы хотим сделать свой MiddleWear, свой класс, свою активность, который будет проверять эту сессию, есть ли там заложенный пользователь.

Как написать свой MiddleWear прописано тут:

<https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/write?view=aspnetcore-8.0>

Можно прописать код непосредственно в Program.cs (не очень), другой вариант – написать свой класс и его строить

Для работы с MiddleWare добавляем еще одну папку => еще класс SessionAuthMiddleware.cs

Есть разные механизмы авторизации (через токены, сессии и т.д.). Мы используем сессионный способ

В нем создаем еще класс расширения

using System.Globalization;

namespace ASP1.MiddleWare

{

//Класс с функциональностью

public class SessionAuthMiddleware

{

//При запуске проекта выстраивается последовательность запуска Middleware и

//каждый класс получает ссылку на следующий, внедряется через конструктор

private readonly RequestDelegate \_next;

public SessionAuthMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

//в процессе работы Middleware должно принять решение, продолжать ли дальнейшую обработку запроса

//если это так, то должен быть вызван следующий обработчик

//Чаще всего Middleware не останавливает работу, а что-то добавляет, фильтрует

// Call the next delegate/middleware in the pipeline.

await \_next(context);

}

}

//Класс-расширение для создания короткого метода app.UseSessionAuth()

public class SessionAuthMiddlewareExtension

{ }

Изображение выглядит как текст, снимок экрана, диаграмма, программное обеспечение

Автоматически созданное описание

Работа MidlleWare условно делится на «прямой ход» - то, что next(), обрабатывают request и «обратный ход» - то, что после next(), по сути обрабатывают response. Но не часто так бывает, что используют и прямой и обратный ход

MiddleWare делится на до и после next (мы делаем до)

public class SessionAuthMiddleware

{

//При запуске проекта выстраивается последовательность запуска Middleware и

//каждый класс получает ссылку на следующий, внедряется через конструктор

//конструктор вызываеся для связывания цепочки

private readonly RequestDelegate \_next;

public SessionAuthMiddleware(RequestDelegate next)

{

\_next = next;

}

//посколько конструктор занят под настройки цепочки вызовов

//инжекция осуществляется в метод InvokeAsync - его аргументы (параметры),

//это те сервисы, которые обычно и в конструкторе других классов

public async Task InvokeAsync(HttpContext context, DataAccessor dataAccessor)

{

var userId = context.Session.GetString("auth-user-id");

if (userId != null) //у сессии есть данные для аутентификации

{

//проверяем эти данные

}

//в процессе работы Middleware должно принять решение, продолжать ли дальнейшую обработку запроса

//если это так, то должен быть вызван следующий обработчик

//Чаще всего Middleware не останавливает работу, а что-то добавляет, фильтрует

// Call the next delegate/middleware in the pipeline.

await \_next(context); //контекс передается дальше

}

}

//Класс-расширение для создания короткого метода app.UseSessionAuth()

public class SessionAuthMiddlewareExtension

{

}

Чтобы проверить данные идем в UserDao и делаем там соответствующий метод

public User? GetUserById(String id)

{

try { return \_context.Users.Find(Guid.Parse(id)); }

catch { return null; }

}

Возвращаемся в SessionAuthMiddleware

using ASP1.Data.Dal;

using System.Globalization;

using System.Security.Claims;

namespace ASP1.MiddleWare

{

//Класс с функциональностью

public class SessionAuthMiddleware

{

//При запуске проекта выстраивается последовательность запуска Middleware и

//каждый класс получает ссылку на следующий, внедряется через конструктор

//конструктор вызываеся для связывания цепочки

private readonly RequestDelegate \_next;

public SessionAuthMiddleware(RequestDelegate next)

{

\_next = next;

}

//посколько конструктор занят под настройки цепочки вызовов

//инжекция осуществляется в метод InvokeAsync - его аргументы (параметры),

//это те сервисы, которые обычно и в конструкторе других классов

public async Task InvokeAsync(HttpContext context, DataAccessor dataAccessor)

{

var userId = context.Session.GetString("auth-user-id");

if (userId != null) //у сессии есть данные для аутентификации

{

//проверяем эти данные

var user=dataAccessor.UserDao.GetUserById(userId);

if (user != null)

{

//Используем встроенную к ASP схему работы с данными аутентификации (утверждения Claims)

//представляют собой пары ключ-значение для обозначения типовых данных

Claim[] claims = new Claim[]

{

new Claim(ClaimTypes.Name, user.Name),

new Claim(ClaimTypes.Email, user.Email),

new Claim(ClaimTypes.Sid, user.Id.ToString()),

new Claim(ClaimTypes.UserData, user.AvatarUrl??""),

new Claim(ClaimTypes.DateOfBirth, user.Birthdate?.ToString()??"")

};

//тут context.User - не наш, это стандартный HTTP, ASP-поле, которое

//собирает данные Claims и схемы аутентификации

//по сути из нашего Usera мы собираем ASP-User, который собирается из стандартизированные Claims

context.User = new ClaimsPrincipal(

new ClaimsIdentity(

claims,

nameof(SessionAuthMiddleware) //название схемы аутентификации

)

);

}

}

//в процессе работы Middleware должно принять решение, продолжать ли дальнейшую обработку запроса

//если это так, то должен быть вызван следующий обработчик

//Чаще всего Middleware не останавливает работу, а что-то добавляет, фильтрует

// Call the next delegate/middleware in the pipeline.

await \_next(context); //контекс передается дальше

}

}

//Класс-расширение для создания короткого метода app.UseSessionAuth()

public static class SessionAuthMiddlewareExtension

{

public static IApplicationBuilder UseSessionAuth(

this IApplicationBuilder builder)

{

return builder.UseMiddleware<SessionAuthMiddleware>();

}

}

}

Дальше идем в Program.cs и после app.UseSession() внедряем наш MiddleWare app.UseSessionAuth()

Итак, все запросы будут проходить через наш MiddleWare, они будут смотреть, если ли в сессии id, если есть - проверять, есть ли такой user в БД, если есть – будет переводить его данные в стандартные Claim, а context.User – везде доступный, и во View, Controller можно доставать этого user и с ним работать

Возвращаемся в Layout

@using System.Security.Claims

@{

//тут не наш User, это Claims!

bool isAuthenticated=Context.User.Identity?.IsAuthenticated??false;

String avatarSrc = isAuthenticated

? Context.User.Claims.First(c=>c.Type==ClaimTypes.UserData).Value

: "";

String userName = isAuthenticated

? Context.User.Claims.First(c=>c.Type==ClaimTypes.Name).Value

: "";

}

@if(isAuthenticated)

{

<b>@userName</b>

}

else{

<a class="nav-link text-dark" href="#" data-bs-toggle="modal" data-bs-target="#authModal">Sign in</a>

}

Доделываем работу с аватарками! В HomeController в методе-валидаторе дописываем

//Если нет ошибок, то обрабатываем файл-аватарку

{

if(res.Count==0)

{

if(formModel.AvatarFile!=null)

{

//Отделяем расширение файла

String ext= Path.GetExtension(formModel.AvatarFile.FileName);

//Определяем место для сохранения

String path = Directory.GetCurrentDirectory() + "/wwwroot/img/avatars";

//Генерируем новое имя для файла (старые нельзя сохранять, возможны конфликты, если

//пользователи будут загружать файлы с одинаковыми именами

String savedName=Guid.NewGuid().ToString() + ext; //берем на базе GUID, но сохраняем расширение

//Сохраняем

using var stream = System.IO.File.OpenWrite(path + savedName);

formModel.AvatarFile.CopyTo(stream);

//Передаем сохраненное имя в модель

//В модели нужно создать дополнительное поле, чтобы это имя файла туда заложить

}

}

}

В wwwroot создаем папку «img», в ней папку «avatars»

В SignUpFormModel добавляем еще одно поле для сохраненного имени файла

//дополнительное поле (не из формы) для имени сохраненного файла

public String? SavedFileName { get; set; }

Дописываем в В HomeController в методе-валидаторе

//Если нет ошибок, то обрабатываем файл-аватарку

{

if(res.Count==0)

{

if(formModel.AvatarFile!=null)

{

//Отделяем расширение файла

String ext= Path.GetExtension(formModel.AvatarFile.FileName);

//Определяем место для сохранения

String path = Directory.GetCurrentDirectory() + "/wwwroot/img/avatars";

//Генерируем новое имя для файла (старые нельзя сохранять, возможны конфликты, если

//пользователи будут загружать файлы с одинаковыми именами

String savedName=Guid.NewGuid().ToString() + ext; //берем на базе GUID, но сохраняем расширение

//Сохраняем

using var stream = System.IO.File.OpenWrite(path + savedName);

formModel.AvatarFile.CopyTo(stream);

//Передаем сохраненное имя в модель

//В модели нужно создать дополнительное поле, чтобы это имя файла туда заложить

formModel.SavedFileName= savedName;

}

}

}

Дальше в методе- map и это передаем

private Data.Entities.User mapUser(SignUpFormModel formModel)

{

String salt = Guid.NewGuid().ToString();//у нас есть сервис для соли, используем его!

Data.Entities.User user = new()

{

Id = Guid.NewGuid(),

Name = formModel.UserName,

Email = formModel.UserEmail,

Register = DateTime.Now,

Birthdate=formModel.Birthdate,

AvatarUrl=formModel.SavedFileName,

Salt = salt,

DerivedKey = \_kdfService.GetDerivedKey(formModel.Password, salt)

};

return user;

}

Идем в Layout

@if(isAuthenticated)

{

<img src="/img/avatars/@(avatarSrc)"/>

}

else{

<a class="nav-link text-dark" href="#" data-bs-toggle="modal" data-bs-target="#authModal">Sign in</a>

}

**20.04.2024**

Задача – создать что-то типа электронного магазина

Начнем с загрузки файлов

Нужно перенести процесс загрузки файлов в сервис

Сервис случайный чисел??

В папке Services=>Add=>new Folder=>Upload

Помним, что сервисы идут, как минимум, парой (интерфейс+имплементация этого интерфейса). Это следствие принципа D (DIP) из SOLID

Upload=>Add Inteface=>IUploadService (создаем интерфейс)

namespace ASP1.Services.Upload

{

public interface IUploadService

{

String SaveFormFile(IFormFile formFile); //просто сохраняет в дефолтном месте файл

String SaveFormFile(IFormFile formFile, String path); //перегрузка с дополнительным путем, куда мы будем сохранять

String SaveFormFile(IFormFile formFile, String path,

IEnumerable<String> extensionsAllowed); //добавляем перечень допустимых расширений для доп. контроля

}

}

Делаем ему имплементацию

Создаем класс UploadService и имплементируем IUploadService

namespace ASP1.Services.Upload

{

public class UploadServiceV1 : IUploadService

{

public string SaveFormFile(IFormFile formFile)

{

//делегируем и указываем дефолтный путь

return SaveFormFile(formFile, "wwwroot/upload/");

}

public string SaveFormFile(IFormFile formFile, string path)

{

//делегируем дальше с пустым массивом передачи разрешенных расширений

return SaveFormFile(formFile, path, []);

}

public string SaveFormFile(IFormFile formFile, string path, IEnumerable<string> extensionsAllowed)

{

//сюда не могут прийти нулевые пути и файл, кидаемся исключениями

ArgumentNullException.ThrowIfNull(formFile, nameof(formFile));

ArgumentNullException.ThrowIfNull(path, nameof(path));

//Отделяем расширение файла

String ext = Path.GetExtension(formFile.FileName);

//Генерируем новое имя для файла (старые нельзя сохранять, возможны конфликты, если

//пользователи будут загружать файлы с одинаковыми именами

String savedName = Guid.NewGuid().ToString() + ext; //берем на базе GUID, но сохраняем расширение

//Определяем место для сохранения

String location = Path.Combine(Directory.GetCurrentDirectory(), path, savedName);

//Сохраняем

using var stream = System.IO.File.OpenWrite(location);

formFile.CopyTo(stream);

//передаем сохраненное имя как результат

return savedName;

}

}

}

В БД не очень хорошая идея вставлять изображения, потому что БД – это один файл, они становятся частью этого файла. Это сильно увеличивает размеры БД. Для БД рекомендуется, чтобы она полностью помещалась в оперативную память сервера, иначе она подтормаживает. Плюс скорость работы с файловой системой быстрее, чем через БД, поэтому мы картинки сохраняем на сайте, а не в БД

Поэтому в БД не ставят картинки, которые являются динамическими. Туда можно заливать картирки для группы для товаров, которые если и меняются, то редко (одну удалил, другую вставил).

Поэтому в БД лежит имя файла картинки, а где он лежит физически – это уже другой вопрос. Обычно делается отдельный сервер файлов, который работает по Web-протоколу

БД только ищет быстрее, если нужно что-то искать по метаданным, благодадря индексируемому поиску, файловая система не может ускориться.

Поэтому в БД сохраняются имена файлов, а уже система знает, где они хостятся. И по сути эти Uploader файлов могут быть какими-то облачными, можно url сохранять

Надо отфильтровать .exe!

public string SaveFormFile(IFormFile formFile, string path, IEnumerable<string> extensionsAllowed)

{

//сюда не могут прийти нулевые пути и файл, кидаемся исключениями

ArgumentNullException.ThrowIfNull(formFile, nameof(formFile));

ArgumentNullException.ThrowIfNull(path, nameof(path));

//Отделяем расширение файла

String ext = Path.GetExtension(formFile.FileName);

//Проверяем расширение на разрешение (исключаем .exe)

//проверяем, если не пустое исключение и хоть что-то есть, но есть то, что мы прописали в исключениях

if (extensionsAllowed.Any() && !extensionsAllowed.Any(e=> e==ext))

{

throw new Exception("Extension not allowed");

};

//Генерируем новое имя для файла (старые нельзя сохранять, возможны конфликты, если

//пользователи будут загружать файлы с одинаковыми именами

String savedName = Guid.NewGuid().ToString() + ext; //берем на базе GUID, но сохраняем расширение

//Определяем место для сохранения

String location = Path.Combine(Directory.GetCurrentDirectory(), path, savedName);

//Сохраняем

using var stream = System.IO.File.OpenWrite(location);

formFile.CopyTo(stream);

//передаем сохраненное имя как результат

return savedName;

}

}

Дальше нужно сервис зарегистрировать. Идем в Program.cs

builder.Services.AddSingleton<IUploadService, UploadServiceV1>();

Дальше его нужно инжектировать. Идем в HomeController

private readonly IUploadService \_uploadService; (через QuickActions добавляем в конструктор!)

ниже в методе- валидаторе меняем код

if (res.Count == 0)

{

if (formModel.AvatarFile != null)

{

try

{

formModel.SavedFileName = \_uploadService.SaveFormFile(

formModel.AvatarFile,

"wwwroot/img/avatars");

}

catch (Exception ex)

{

res[nameof(formModel.AvatarFile)] = ex.Message;

}

}

Дальше нужно сделать как минимум две сущности (Entity): товари и товарная группа

В папке Entity создаем классы Category, Product

public class Category

{

public Guid Id { get; set; }

//Slug - делаем для красоты. Это ресурсный идентификатор.

//Часть url-адреса, который идентицирует ресурс, используется, чтобы делать url на категорию

//не url/id, а url/изделия из камня

// то есть позволяют однозначно идентифицировать ресурс и являются человекопонятными

//и используются для красивых адресо

public string Slug { get; set; } = null!;

public string Name { get; set; } = null!;

public string Description { get; set; } = null!;

public string ImageUrl { get; set; } = null!;

//активная или не активная категория, по дефолту - активная, если не указано иное

public Boolean IsActive { get; set; } = true;

}

public class Product

{

public Guid Id { get; set; }

public Guid CategoryId { get; set; }

public string Name { get; set; } = null!;

public string Description { get; set; } = null!;

public string ImageUrl { get; set; } = null!;

public Boolean IsActive { get; set; } = true;

public Double Price { get; set; }

}

Если для товара нужно несколько картинок, лучше это реализовать через отдельную таблицу, потому что мы не знаем, сколько картинок там может быть, как-то ограничивать не стоит, должна быть хотя бы одна

Дальше идем в DataContext, добавляем созданные сущности

public DbSet<Category> Categories { get; set; }

public DbSet<Product> Products { get; set; }

В методе protected override void OnModelCreating(ModelBuilder modelBuilder)

Добавляем уникальность для поля Slug

modelBuilder.Entity<Category>()

.HasIndex(u => u.Slug)

.IsUnique();

Для товаров этого не делаем, потому что товары идут по айдишнику, придумать уникальное имя для каждого товара сложно

Навигационные свойства. Связи создаются не ОРМ, а самой базой СУБД (SQL во время выполнения). Понятие внешний включ – это лишь ограничение, средство ограничения, чтобы в это поле мог не любой id лечь, а только такой, который есть в Категориях. Что будет если удалить категорию? Внешний ключ будет мешать при попытке удалить категорию, при копировании БД. Идея хорошая, но она не ускоряет ничего, только создает запрет и жесткое связывание

Внешний ключ указывается для случая, когда мы используем навигационные свойства (через них можно указать эти связи, но это работает только в Entity Framework, и только в Entity). НЕ ДЕЛАЕМ НАВИГАЦИОННЫЕ СВОЙСТВА

После смены Контекста, надо сделать миграцию и обновить БД

Add-Migration Product

Update-Database

Дальше пропускаем это все через DataAccessor

В папке Dal=>Add=>Class=>ShopDao (объединим тут методы и по категориям, и по продуктам)

Часто все эти операции остаются в DataContexte и все операции с DataContext переносятся в конструкторы, потому что они достаточно простые

QuickActions – generate constructor=>QuickActions =>Use Primary constructor

public class ShopDao(DataContext context)

{

private readonly DataContext \_context = context;

}

Синхронизация контекста данных – закладіваем сразу

public class ShopDao(DataContext context, Object dbLocker)

{

private readonly DataContext \_context = context;

// нужен для синхронизации контекста данных

private readonly object \_dbLocker = dbLocker;

public List<Category> GetCategories()

{

List<Category> res;

lock (\_dbLocker)

{

res=\_context.Categories.Where(c=>c.IsActive).ToList();

}

return res;

}

}

Часто такую штуку закладывают прямо в Контроллер, т.е. ради одной строки создать целый Accessor, да, но с другой стороны, если переходить на других поставщиков данных, придется к ним обращаться как-то отдельно, а значит, нужно будет лезть в Контроллер и переписывать там логику. Поэтому создаем централизированные аксессоры, где все и настравиваем

Дальше подключаем это в DataAccessor

public class DataAccessor

{

public DataContext \_context;

private readonly IKdfService \_kdfService;

private readonly Object \_dbLocker = new(); //это не инжекция, это внутренний механизм, он извне не нужен

//DataAcessor через контейнер получает все зависимости и всереди уже перераспределяет между теми

//Dao, которые в них нуждаются

public UserDao UserDao { get; private set; }

public ShopDao ShopDao { get; private set; }

public DataAccessor(DataContext context, IKdfService kdfService)

{

\_context = context;

\_kdfService = kdfService;

UserDao = new(\_context, \_kdfService);

ShopDao = new(\_context, \_dbLocker);

}

}

Теперь к нему есть доступ.

Идем в Контроллер, создаем отдельный контроллер Controllers=>Add=>Controller=>MVS Controller Empty=>ShopController.cs

Для него создаем View обязательно с таким же названием, как и Контроллер, для этого в папке View=>Add=>New Folder=>Shop (назовем по-другому, работать не будет)

Дальше Shop=>Add=>View=>Razor Empty (Index стоит по умолчанию)

Для магазина можно сделать отдельный Layout, чтобы не проходить через стандартный

Shop=>Add=>View=>Razor Empty=>\_ShopLayout.cshtml

На главный \_Layout добавляем еще одну ссылку (меняем контроллер и Экшн)

<li class="nav-item">

<a class="nav-link text-dark" asp-area="" asp-controller="Shop" asp-action="Index">Shop</a>

</li>

Находим картинку, переименовываем (logo.jpg), кладем ее в папку wwwroot/img/

В \_ShopLayout перед закрывающимся тегом </header>

<div class="sub-header">

<h1>Найкращий вибір</h1>

<h2>Тільки в нас</h2>

</div>

Идем в site.css и описываем стили

.sub-header{

background:url(/img/logo.jpg) no-repeat;

background-size:cover;

width:100%;

height:25vh;

min-height:200px;

}

.sub-header h1, .sub-header h2{

color: aqua;

font-size:large;

}

В shop=>index.cshtml нужно переключить Layout

@{

Layout="\_ShopLayout";

}

Добавляем DataAsccessor в ShopController и сохраняем его

public class ShopController (DataAccessor dataAccessor): Controller

{

private readonly DataAccessor \_dataAccessor=dataAccessor;

public IActionResult Index()

{

//передаем категории на представление

ViewData["Categories"]=\_dataAccessor.ShopDao.GetCategories();

return View();

}

}

Теперь надо создать админку для добавления категорий

В Shop=>Index.cshtml

@using System.Security.Claims

@{

// Переключаем на наш Layout

Layout="\_ShopLayout";

// Блок проверки аутентификации

bool isAuthenticated=Context.User.Identity?.IsAuthenticated??false;

}

@if (isAuthenticated)

{

<div class="row justify-content-end">

<div class="col col-5">

<h2>Додавання нової групи товарів</h2>

<form>

<div class="row">

<div class="col">

<div class="input-group mb-3">

<span class="input-group-text" id="category-name"><i class="bi bi-person-fill"></i></span>

<input type="text" class="form-control" placeholder="Назва"

name="category-name" aria-label="Category-name" aria-describedby="category-name">

</div>

</div>

<div class="col">

<div class="input-group mb-3">

<span class="input-group-text" id="category-slug"><i class="bi bi-person-fill"></i></span>

<input type="text" class="form-control" placeholder="Посилання"

name="category-slug" aria-label="Category-slug" aria-describedby="Category-slug">

</div>

</div>

</div>

<div class="row mb-3">

<div class="input-group">

<span class="input-group-text"><i class="bi bi-info"></i></span>

<textarea class="form-control" name="category-description"

placeholder="Опис категорії"></textarea>

</div>

</div>

<div class="row mb-3">

<div class="col col-8">

<div class="input-group mb-3">

<label class = "input-group-text" for="category-image"><i class="bi bi-card-image"></i></label>

<input type="file" class="form-control" name="category-image" id="category-image"/>

</div>

</div>

<div class="col col-4">

<button type="submit" class="btn btn-primary">Додати</button>

</div>

</div>

</form>

</div>

</div>

}

Теперь нужно сделать ФормМодель, где мы все принимаем, трансформацию на EntityModel и загоняем в базу

По сути Контроллер, который примет модель формы, преобразует ее в модель БД и передаст в DataAccessor

Идем в папку Models=>Add new Folder=>Shop=>Add=>Class=>ShopCategoryFormModel

Чем она отличается от Entity, тем, что тут идут данные, характерные для HTML, в частности, файлы, нет айдишки

public class ShopCategoryFormModel

{

[FromForm(Name="category-slug")]

public string Slug { get; set; } = null!;

[FromForm(Name = "category-name")]

public string Name { get; set; } = null!;

[FromForm(Name = "category-description")]

public string Description { get; set; } = null!;

[FromForm(Name = "category-image")]

public IFormFile Image { get; set; } = null!;

}

И паралельно рассмотрим, как реализовать API, чтобы реализовать универсальную схему и мы могли, например, могли frontend на мобильном устройстве или использовать фреймворк (React|Angular), для этого нужен универсальный контроллер, через который мы будем проганять данные, чтобы не форма напрямую обращалась к своему же контроллеру, который мы потом не сможем отделить, а чтобы мы могли обработать данные и отправить соответствующие результаты

Для этого в папке Controller=>Add=>Controller=>API=>API Controller-Empty=>ShopAPIController

[Route("api/shop")] //меняем адрес с [controller] на shop

[ApiController]

public class ShopAPIController : ControllerBase

{

}

[Route("api/shop")]

[ApiController]

public class ShopAPIController : ControllerBase

{

[HttpPost("category")]

public object DoPost()

{

return new { status = "Works" };

}

}

Теперь идем в Джаваскрипт site.js

document.addEventListener('submit', e => {

//перехватываем отправку форм (всех)

const form = e.target;

if (form.id == 'category-form') { //форма добавления категорий товаров

e.preventDefault(); //останавливаем автоматическую отправку

fetch("/api/shop/category", { // и переводим к асинхронный режим

method: 'POST'

}).then(r => {

console.log(r);

})

}

})

**Айдишник добавляем в форму, и теперь только форма с таким айдишником будет перехватываться**

Запускаем проект, правой кнопкой => Посмотеть код=>Network, смотрим

Изображение выглядит как текст, снимок экрана, веб-страница, Веб-сайт

Автоматически созданное описание

Данные принимаются в ShopApiController (добавляем в качестве параметра ShopCategoryFormModel)

[Route("api/shop")]

[ApiController]

public class ShopAPIController : ControllerBase

{

[HttpPost("category")]

public object DoPost(ShopCategoryFormModel model)

{

return new { status = "Works" };

}

}

А передает данные site.js

document.addEventListener('submit', e => {

//перехватываем отправку форм (всех)

const form = e.target;

if (form.id == "category-form") { //форма добавления категорий товаров

e.preventDefault(); //останавливаем автоматическую отправку

//Собираем данные формы

let formData = new FormData(form);

fetch("/api/shop/category", { // и переводим к асинхронный режим

method: 'POST',

body: formData //передаем данные на бэкэнд

}).then(r => {

console.log(r);

})

}

})

Но! Это не все! Нужно форме разрешить передавать данные (в файле Views/Shop/Index.cshtml)

<form id='category-form' method="post" enctype="multipart/form-data">

Дальше нужно сделать валидатор

public class ShopAPIController : ControllerBase

{

[HttpPost("category")]

public object DoPost(ShopCategoryFormModel model)

{

//model==null - не может, быть нулевой, если придет нуль, средствами ASp будет отказано в операции

if (String.IsNullOrEmpty(model.Slug) ||

String.IsNullOrEmpty(model.Name) ||

String.IsNullOrEmpty(model.Description))

{

Response.StatusCode = StatusCodes.Status422UnprocessableEntity;

return "Missing required data";

}

return new { status = "Works" };

}

}

Дальше нужно проинжектировать сервисы

public class ShopAPIController (DataAccessor dataAccessor, IUploadService uploadService): ControllerBase

{

private readonly DataAccessor \_dataAccessor = dataAccessor;

private readonly IUploadService \_uploadService = uploadService;

….

}

using HWASP.Models.ServicesShop;

using HWASP.Services.Upload;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

namespace HWASP.Controllers

{

[Route("api/servicesshop")]

[ApiController]

public class ServicesShopAPIController (DataAccessor dataAccessor, IUploadService uploadService): ControllerBase

{

private readonly DataAccessor \_dataAccessor=dataAccessor;

private readonly IUploadService \_uploadService=uploadService;

[HttpPost("category")]

public object DoPost(ServicesShopCategoryFormModel model)

{

if(String.IsNullOrEmpty(model.Slug) ||

String.IsNullOrEmpty(model.Name) || String.IsNullOrEmpty(model.Description))

{

Response.StatusCode=StatusCodes.Status422UnprocessableEntity;

return "Missing required data";

}

try

{

\_dataAccessor.ServicesShopDao.AddCategory(

name: model.Name,

slug: model.Slug,

description: model.Description,

imageUrl: \_uploadService.SaveFormFile(model.Image, "wwwroot/img/servicesImg"));

Response.StatusCode = StatusCodes.Status201Created;

return "Ok";

}

catch (Exception ex)

{

Response.StatusCode = StatusCodes.Status422UnprocessableEntity;

return ex.Message;

}

}

}

}

И переводим на модель, идем в ShopDao и делаем метод, который будет добавлять категорию

public void AddCategory(String name, String slug, String description, String imageUrl)

{

AddCategory(new()

{

Name = name,

Slug = slug,

Description = description,

ImageUrl = imageUrl,

IsActive = true,

Id = Guid.NewGuid()

});

}

public void AddCategory(Category category)

{

lock(\_dbLocker)

{

\_context.Categories.Add(category);

\_context.SaveChanges();

}

}

ВАЖНО! Чтобы dbLocker был один на всех, поэтому прописываем то же в UserDao, нужно вынести этот объект в DataAccessor и во все другие Dao его передаем как ссылку и это приводит к тому, что когда они лочат, то один и тот же объект, благодаря чему синхронизируются.

Надо переделать метод в UserDao

public User? GetUserById(String id)

{

User? user;

lock(\_dbLocker )

{

try { user= \_context.Users.Find(Guid.Parse(id)); }

catch {user= null; }

}

return user;

}

По хорошему все нужно залочить, но главное это, потому что оно в MiddleWare записано, который грузится в асинхронном режиме

Однопоточная синхронизация перекладывается на нас

Возвращаемся в ShopLayout

Используем данные из категорий для загрузки списка категорий

<div class="row">

<div class="col col-2">

@{

//преобразовываем из ViewData в List и запускаем потом в цикле

var Categories = ViewData["Categories"] as List<Category>;

}

@foreach(var category in Categories)

{

<p>@category.Name</p>

}

</div>

<div class="col col-9">

<div class="sub-header">

<h1>Найкращий вибір</h1>

<h2>Тільки в нас</h2>

</div>

</div>

</div>

Идем в Index

@using System.Security.Claims

@using ASP1.Data.Entities

@{

// Переключаем на наш Layout

Layout="\_ShopLayout";

// Блок проверки аутентификации

bool isAuthenticated=Context.User.Identity?.IsAuthenticated??false;

var Categories = ViewData["Categories"] as List<Category>;

}

<div class="row">

@foreach(var category in Categories)

{

<div class = "col">

<img src="~/img/shop/@(category.ImageUrl)" alt="Image"/>

<p>@category.Name</p>

</div>

}

</div>

Нужно обновление страницы перенести в джаваскрипт

document.addEventListener('submit', e => {

//перехватываем отправку форм (всех)

const form = e.target;

if (form.id == "category-form") { //форма добавления категорий товаров

e.preventDefault(); //останавливаем автоматическую отправку

//Собираем данные формы

let formData = new FormData(form);

fetch("/api/shop/category", { // и переводим к асинхронный режим

method: 'POST',

body: formData //передаем данные на бэкэнд

}).then(r => {

console.log(r);

if (r.status == 201) {

window.location.reload();

}

else {

r.text().then(alert);

}

})

}

})

**27.04.2024**

**Рассматриваем флагманский раздел API и REST**

В современных системах используется распределенная архитектура построения проектов, в которых используется идея API (Application Program Interface)

Program – определенный информационный центр, где сохраняются данные всего проекта, к которому могут обратиться все Application

Интерфейс взаимодействия между Application и Program

Application – прикладные программы, работа которых напрямую зависят от Program (информационного центра), через которую происходит авторизация, доступ к контенту и вся работа, завязанная с бэкэндом (Программом).

Fronted является Application (сайты, веб-приложения и т.д., это части комплексу, которые запрашивают у Program какие-то данные и отображают их, или наоборот запрашивают какие-то данные у пользователя и передают их на бэкэнд).

Интерфейс взаимодействия между Application (Frontend) и Program (Backend) и называется API (Application Program Interface)

Если наша программа – это наш комплекс, как мы этот протокол сделаем, например, можем привести СУБД (мы можем запустить консоль и писать запросы, можем запустить php, SQL Management Studio и т.д., которые по сути общаются с одной и той же БД, и наше приложение тоже общается с ней же)

API позволяет легко расширять Application, создавать мобильные приложения (отдельно для Андроид, отдельно для iOS), веб-приложения, также позволяет реализовать задачи открытости

При создании API нужно соблюдать традиции, которые собраны под названием REST (Representtion State Transfer) – передача данных, который репрезентует состояние. Он накладывает определенные ограничения, и система, которая их удовлетворяет, называется RESTFull.

Нас интересуют архитектурные ограничения:

- клиент/сервер

- независимый от состояния (каждый запрос обрабатывается как новый, если по простому - не используются сессии)

- кэш – можно кэшировать данные или нет (если картинка товара – можно кэшировать, то цену – нет)

- слоеная система – (MVC – это и так слоеная система)

- унифицированный интерфейс (ЦЕНТРАЛЬНОЕ ТРЕБОВАНИЕ):

1. интерфейс ответов (идентификация ресурсов в request

2. манипуляция ресурсами через представления (мы добавляем какие-то метаданные, которых достаточно, чтобы разобраться, это наши методы запроса (Get, Post), от того, что изменится метод запроса, от этого никак не поменяется форма и данные

3. самоописывающие сообщениия – каждое сообщение содержит достаточно информации, чтобы описать, как процесс

В ASP многие требования/ограничения уже реализованы

REST – набор архитектурных рекомендаций. Желательно ему следовать

Центральной из них является «Единый интерфейс», в нашем случае – для бэкэнда это, в частности, рекомендация приведения всех ответов к единой форме.

Какой форме? Как хотите. Есть определенные традиции, но должно быть понятно, к какому ресурсу мы обращаемся и с каким действием.

Например:

К нам приходит JSON-объект, в котором уже заложено несколько полей, например, метаданные и сами данные. В метаданных мы должны разянить, что это было

GET /category/16as54-sdfsdf….-> (запрос)

|  |  |
| --- | --- |
| Ответ (отрицательный):  {  status:{  code: 404,  message: “Requested id not found”  },  meta:{  service: “Shop categories”,  serverTime: 123455,  requestParameters:{  id: “16as54-sdfsdf”,  ….  }  },  data: []  } | Ответ (положительный):  {  status:{  code: 200,  message: “Success”  },  meta:{  service: “Shop categories”,  serverTime: 123455,  requestParameters:{  id: “16as54-sdfsdf”,  ….  }  },  data: [id:…, name…, descr…]  } |

Внедряем технологию. Делаем модель

Models=>add=>New Folder=>Rest=>Add=>Class=>RestResponse

namespace ASP1.Models.Rest

{

public class RestResponce

{

public RestStatus Status { get; set; } = new()

{ Code = 200, Message = "Ok", IsOk = true };

public RestMeta Meta { get; set; } = new();

public Object Data { get; set; } = new();

}

public class RestStatus

{

public Int32 Code { get; set; }

public String Message { get; set; } = "";

public Boolean IsOk { get; set; }

}

public class RestMeta

{

public String Service { get; set; } = "";

public int Total { get; set; }

public Int64 ServerTime { get; set; } = DateTime.Now.Ticks;

public Dictionary<String, String> RequestParameters { get; set; } = [];

}

}

Дальше идем в наш APIController, заявляем, что он будет возвращать не просто object, а RestResponce

using ASP1.Data.Dal;

using ASP1.Models.Rest;

using ASP1.Models.Shop;

using ASP1.Services.Upload;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

namespace ASP1.Controllers

{

[Route("api/shop")]

[ApiController]

public class ShopAPIController(DataAccessor dataAccessor, IUploadService uploadService) : ControllerBase

{

private readonly DataAccessor \_dataAccessor = dataAccessor;

private readonly IUploadService \_uploadService = uploadService;

[HttpPost("category")]

public RestResponce DoPost(ShopCategoryFormModel model)

{

RestResponce restResponce = new();

restResponce.Meta = new()

{

Service = "/api/shop/category",

ServerTime = DateTime.Now.Ticks,

RequestParameters = model.ToParams()

};

//model==null - не может, быть нулевой, если придет нуль, средствами ASp будет отказано в операции

if (String.IsNullOrEmpty(model.Slug) ||

String.IsNullOrEmpty(model.Name) ||

String.IsNullOrEmpty(model.Description))

{

restResponce.Status = new()

{

Code = StatusCodes.Status422UnprocessableEntity,

Message = "Missing required data",

IsOk = false

};

}

else

{

try

{

\_dataAccessor.ShopDao.AddCategory(

name: model.Name,

slug: model.Slug,

description: model.Description,

imageUrl: \_uploadService.SaveFormFile(model.Image, "wwwroot/img/shop"));

restResponce.Status = new()

{

Code = StatusCodes.Status201Created,

Message = "Ok",

IsOk = false

};

}

catch (Exception ex)

{

restResponce.Status = new()

{

Code = StatusCodes.Status422UnprocessableEntity,

Message = "Missing required data", //ex.Message

IsOk = false

};

}

}

return restResponce;

}

}

}

Меняем возвращаемое значение в AddCategory, для этого нажием Ctrl и правой кнопкой на методе – попадаем в ShopDao

public Category AddCategory(String name, String slug, String description, String imageUrl)

{

Category category = new()

{

Name = name,

Slug = slug,

Description = description,

ImageUrl = imageUrl,

IsActive = true,

Id = Guid.NewGuid()

};

return AddCategory(category);

}

public Category AddCategory(Category category)

{

lock (\_dbLocker)

{

\_context.Categories.Add(category);

\_context.SaveChanges();

}

return category;

}

Возвращаемся в APIController и возвращаем категорию

try

{

var category=\_dataAccessor.ShopDao.AddCategory(….

//Сохраняем эту категорию, которую мы добавили и в Response можем в Data добавить

restResponce.Data= category;

Теперь возвращаем то, что мы добавлии в БД

Что касается исключений. Передавать Excetion на клиент – очень плохая идея, как с точки зрения безопасности, так и вообще. Если там какие-то проблемы с SQL-запросом или с подключением к БД, то в сообщении стоят технические детали типа SQL-сервер, типа таблица такая-то и это все может дать злоумышленнику какую-то модель атаки, чтобы он уже понимает уже диалекты SQL, уязвимости, названия таблиц, полей и т.д. Короче, пользователю нельзя отправлять какие-то детали этих исключений категорически.

Чтобы сохранить эту информацию, используем специальный инструмент – логгер. Поэтому добавляем себе логгер.

При создании нового проекта нам дается HomeController, в котором автоматически добавляется логгер прямо на старте:

private readonly ILogger<HomeController> \_logger;

Добавляем в наш APIController

private readonly ILogger<ShopAPIController> \_logger;

(информация из логгера по умолчанию выводится на консоль, можно настроить, чтобы выводилось в файл или БД – отдельная тема)

Правой кнопкой QuickAction=>Add parameter to constructor, присваиваем полю значение из конструктора

private readonly ILogger<ShopAPIController> \_logger=logger;

Тема логирования – отдельная большая тема, не стартовая. На самостоятельное обучение. Есть механизм логгера и когда мы создаем логгер, мы добавляем этот класс. Это позволяет разделять сообщения от разных классов в разные файлы. Логгеры можно настраивать на сохранение не в файлы, а в БД, можно, чтобы они сразу по e-mail сразу отправляли. Смысл один: в логгер мы что-то пишем, оно куда-то отправляет сообщения. По умолчанию – на консоль

Теперь этим логгером логируем наш Exception

Логгер подчергивает, что желательно поддерживать единую форму сообщений.

//посмотреть информацию из логгера:

\_logger.LogInformation(String.Join(',',restResponce.Meta.RequestParameters.Values));

Бывают традиции, когда логируется все, все обращения к API. Если обращений много – выходит большая нагрузка. **По условиям безопасности обязательно сохранять данные (запросы) по внесению изменений.** То, что не меняет систему – не обязательно.

using ASP1.Data.Dal;

using ASP1.Models.Rest;

using ASP1.Models.Shop;

using ASP1.Services.Upload;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

namespace ASP1.Controllers

{

[Route("api/shop")]

[ApiController]

public class ShopAPIController(DataAccessor dataAccessor, IUploadService uploadService, ILogger<ShopAPIController> logger) : ControllerBase

{

private readonly DataAccessor \_dataAccessor = dataAccessor;

private readonly IUploadService \_uploadService = uploadService;

private readonly ILogger<ShopAPIController> \_logger=logger;

[HttpPost("category")]

public RestResponce DoPost(ShopCategoryFormModel model)

{

RestResponce restResponce = new();

restResponce.Meta = new()

{

Service = "/api/shop/category",

ServerTime = DateTime.Now.Ticks,

RequestParameters = model.ToParams()

};

\_logger.LogInformation(String.Join(',',restResponce.Meta.RequestParameters.Values));

//model==null - не может, быть нулевой, если придет нуль, средствами ASp будет отказано в операции

if (String.IsNullOrEmpty(model.Slug) ||

String.IsNullOrEmpty(model.Name) ||

String.IsNullOrEmpty(model.Description))

{

restResponce.Status = new()

{

Code = StatusCodes.Status422UnprocessableEntity,

Message = "Missing required data",

IsOk = false

};

}

else

{

try

{

var category = \_dataAccessor.ShopDao.AddCategory(

name: model.Name,

slug: model.Slug,

description: model.Description,

imageUrl: \_uploadService.SaveFormFile(model.Image, "wwwroot/img/shop"));

restResponce.Status = new()

{

Code = StatusCodes.Status201Created,

Message = "Ok",

IsOk = false

};

restResponce.Data = category;

}

catch (Exception ex)

{

\_logger.LogError(ex.Message);

restResponce.Status = new()

{

Code = StatusCodes.Status500InternalServerError,

Message = "Server error. Details in logs",

IsOk = false

};

}

}

return restResponce;

}

}

}

Как тестировать бэкэнд?

Для этого можно или на фронтэнде сделать или использовать сторонний инструмент – PostMan

Устанавливаем его: <https://www.postman.com/downloads/?utm_source=postman-home>

Эта программка позволяет создавать запросы. Get-запросы можно легко сделать и через браузер, а вот POST – уже проблема, либо писать свои средства для отправки запросов, или использовать другие средства типа Postman

Запускаем бэкэнд, чтобы он работал

Postman=>new=>HTTP

Если не отправлять никакие данные, а контроллер у нас ждем FormModel

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Отправляем реальные данные (в обход клиента)

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

**То есть кто угодно через API может добавить данные в БД, поэтому нужно реализовывать авторизацию. Принципы авторизации по API**

**Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание Подсказки, о чем гуглить!**

**Postman позволяет тестировать запросы, если авторизироваться, их можно сохранять в так называемые Postman-коллекции (папочки), которые сохраняются где-то в облаке**

Делаем метод Get ShopApiController

[HttpGet("category")]

public RestResponce GetCategory()

{

return new()

{

Status = new()

{

Code = StatusCodes.Status200OK,

Message = "Ok",

IsOk = true

},

Data = \_dataAccessor.ShopDao.GetCategories(),

Meta = new()

{

Service = "/api/shop/category",

ServerTime = DateTime.Now.Ticks

}

};

}

Когда мы сделали метод Get, уже ответ приходит в виде RestResponce

**Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание**

**REST очень упрощает разработку клиентских приложений.**

По сессиям: можно ли повременно оставить, или как оно работает на сайтах обычных



Сессии настраиваются с таймаутом бездеятельности (если 10 секунд ничего не делается на сайте, то сессия срывается), бывает абсолютный таймаут, т.е. от выдачи

Сессия настраивается по связке браузер-бэкэнд. Сами данные сохраняются в памяти ПК (MemoryCache), а стучимся к этим данным через куки (метаданные, которые браузер сохраняет и использует, но еще и пересылает, точнее включает в каждый запрос, который выходит из браузера на соответствующий сайт). Если наш сайт установил куки на день, то в течение дня все запросы на сайт будут автоматически браузером без нашего вмешательства и разрешения будут включать эту куку в запрос.

Cookie

Это часть HTTP-стандарта, согласно которой клиент обязан передавать все полученные от сервера Cookie в запросах к этому серверу.

Приблизительный таймпас взаимодействия

|  |  |
| --- | --- |
| Client(Browser) | Server |
| Get /shop ==🡺 | 200 OK |
|  | Заголовки Set-Cookie: x=10; expires=Tie,24Mar (дата, когда куки будет уничтожена) … |
| Сохранение Cookie (x=10) для<===== данного сайта | … |
| Другой запрос/обновление страницы |  |
| Get /shop  Cookie: x=10 ==🡺  (в запрос автоматически добавляются заголовки из сохраненными Cookie) |  |
|  |  |