**Integration of database into c# using ADO.NET**

First we have our “connection string” that has all the necessary information for our application to connect to our database:

Server=myServerAddress;Database=myDataBase;Trusted\_Connection=True;

"Server=WIN-DNH1G34J36A;Database=Students;Trusted\_Conntection=True"

Let’s store our connection string in a field.

private static readonly string \_connectionString = "Server=WIN-DNH1G34J36A;Database=Students;Trusted\_Connection=True”;

Now we are gonna connect to our database using SqlConnetion and send info(model) to our view(razor page). We use SQLConnection, SqlCommand, SqlDataReader

public IActionResult Index()

{

List<Student> students = new List<Student>();

using (SqlConnection conn=new SqlConnection(\_connectionString))

{

SqlCommand command = new SqlCommand("select \* from students", conn);

conn.Open();

var reader = command.ExecuteReader();

while (reader.Read()) //as long as theres code to read

{

// in Models we have got a student class🡪

Student student = new Student();

student.id = (int)reader["student\_id"];

student.gpa = (int)reader["gpa"];

student.name = reader["name"].ToString();

student.major = reader["major"].ToString();

students.Add(student);

}

}

return View(students);

}

In our view we receive the model…

@model List<MVCPractise.Models.Student>

@{

ViewData["Title"] = "Home Page";

}

<div class="text-center">

<ul>

@foreach(Student student in Model )

{

<li>ID:@student.id Name:@student.name Major:@student.major Gpa:@student.gpa</li>

}

</ul>

</div>

First we create a SqlConnection object, then we open the connetion by using Open() method. After that, we need a command and then we use our SqlCommand reader to read our databse record by record.

**Route**

A route is a URL pattern that is mapped to a handler called a controller

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

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

});

For instance, if we have a products controller and action message inside it that takes name and num parameters then it will be:

Products/message?name&num

**ViewBag**

ViewBag is a way to pass data from the controller to the view.

ViewBag is a dynamic type meaning that the type is checked at the end.

When we write🡪

ViewBag.propName=”Hey”;

ViewBag becomes an object which generates a propName property for us.

To display these properties in view, you would need to use the same property names.

**TempData**

TempData is used to transfer data from view to controller, controller to view, or from one action method to another action method of the same or a different controller.

The main difference between ViewBag and TempData is that when we transfer data from one action to another using RedirectToAction() method and ViewBag is going to lose that data whatever we stored in it but TempData saves that data and then we can pass that TempData to our View. TempData implements Idictionary.

TempData helps us to transfer data between controllers or between actions.

public IActionResult NewsList(int? id)

{

if (id==null)

{

ViewBag.Info = "Id yoxdursa xeberde yoxdur";

return View();

}

TempData["info"]= "Sene teze xeber getirmishem";

return RedirectToAction("NewsDetail");

}

public IActionResult NewsDetail()

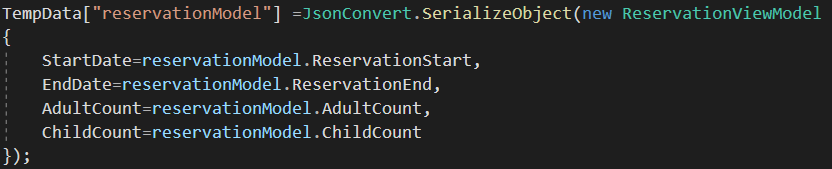
{

return View("Index");

}

When we send an object such as a model to another controller via TempData then we need to serialize that object/model when we send it.

TempData uses Session, which itself uses IDistributedCache. IDistributedCache doesn't have the capability to accept objects or to serialize objects. As a result, you need to do this yourself, i.e.:



And when of course we receive it we need to deserialize it 🡪



**ViewData**

ViewData is a dictionary. It is for less comlex information.Usually, ViewBag we use for objects. But ViewData we use for storing single pieces of information like strings.

ViewData["Name"] = "vahid";

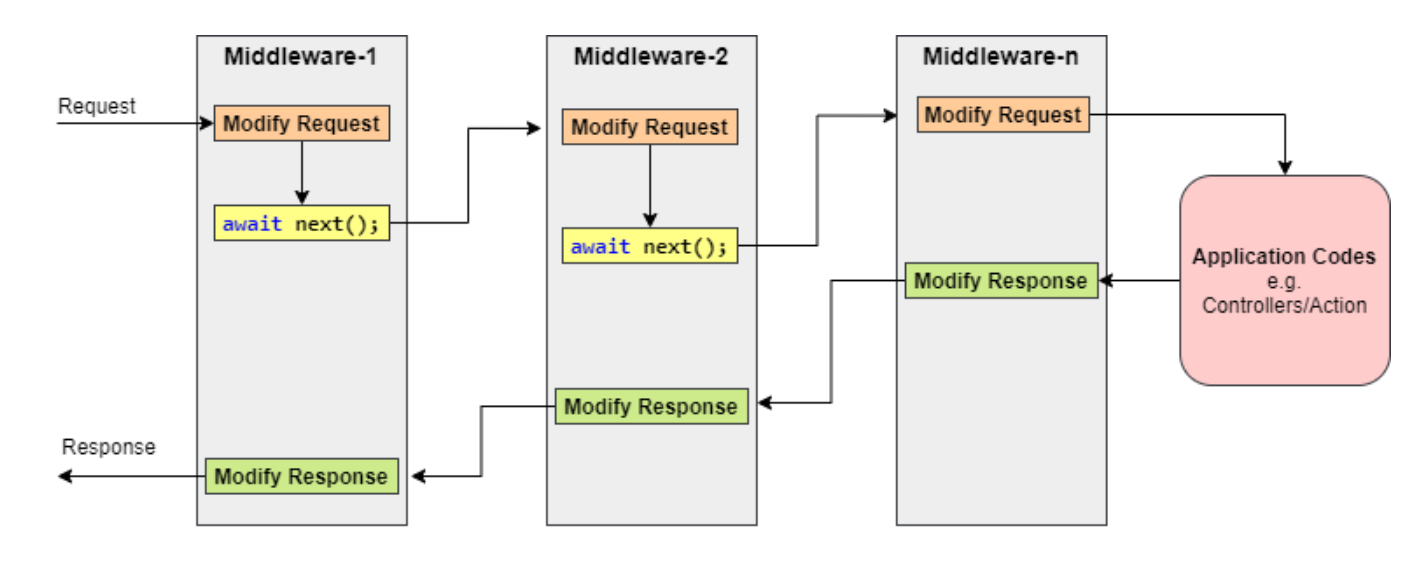
<h1> @ViewData["Name"]</h1>

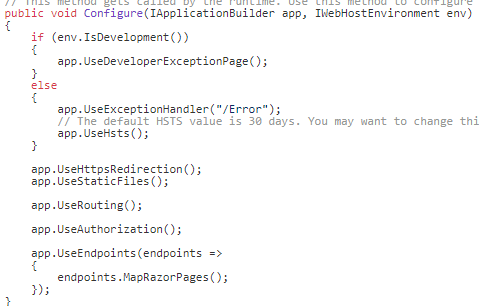
**MiddleWare**

The middleware is the pipeline between the request and our code. A middleware is nothing but a component (method) which is executed on every request in ASP.NET Core application. Middleware is a piece of code in an application pipeline used to handle requests and responses. For example, we may have a middleware component to authenticate a user, another piece of middleware to handle errors, and another middleware to serve static files such as JavaScript files, CSS files, images, etc. The middleware has 3 main responsibilities

1. It receives
2. It changes
3. It activates the next middleware if there is any. (by using Use method)

At this point, it is important to note that the middleware pipeline is bi-directional. When the request enters the pipeline, it will flow through the different components of the pipeline until one of them can produce a response, or an exception is thrown. Once this happens, the response will go through the pipeline again in the opposite direction, being inspected by the same components that the request went through on the way in.





**DeveloperExceptionPage**: the middleware pipeline is bi-directional. This means that the first component of the pipeline will be both the first to inspect the incoming HTTP request and the last to inspect the response. This will condition what we decide to place at the very beginning of the pipeline (or at the very end). In this case, DeveloperExceptionPage has been placed at the top because it cares about being the last to inspect the response. Its task is to handle exceptions being thrown by any other component in the pipeline.

When DeveloperExceptionPage receives an HTTP request, it will simply pass it on to the next component in the pipeline. If any subsequent component in the pipeline throws an exception, it will be passed back to the previous components of the pipeline and will finally reach DeveloperExceptionPage. DeveloperExceptionPage is in charge of rendering HTML containing detailed information on the exception, which will be useful for developers when debugging the problem.

**ExceptionHandler**: Note that DeveloperExceptionPage is only used in development. When running in production, the level of detail rendered would be not only unnecessary, but could also expose sensitive information to users. When in production, ExceptionHandler will be used instead. ExceptionHandler operates in the same way as DeveloperExceptionPage, but the information displayed when an exception has been thrown is less detailed and more user friendly.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World From 1st Middleware");

});

// the following will never be executed

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World From 2nd Middleware");

});

}

To configure multiple middleware, use Use() extension method. It is similar to Run() method except that it includes next parameter to invoke next middleware in the sequence. Consider the following example.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

app.Use(async (context, next) =>

{

Await context.Response.WriteAsync("Hello World From 1st Middleware!");

await next();

});

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World From 2nd Middleware");

});

}





**View**

An explicit view to return:

return View("Orders");

public ViewResult Index(Nullable<int> id)

{

ViewResult view = new ViewResult();

view.ViewName = "~/Views/Student/Index.cshtml";

return view;

}

We can also pass an object to a view

public ViewResult Index(int? id)

{

int x = 3;

return View(x);

}

@model int

<h1> @Model</h1>

The Model always has to be capitalized.

**ViewModels**

ViewModel in ASP.NET Core MVC is a model that contains more than one model data required for a particular view. Combining multiple model objects into a single view model object provides us better optimization.

If we want to pass a complex data like 2 lists of objects then we gotta wrap them in another class, namely in ViewModel class.

Student student1 = new Student("Vahid", "Rashidli");

Student student2 = new Student("Nihad", "Rashidli");

Group group1 = new Group() { Name = "P319", NumberOfStudents = 2 };

Group group2 = new Group() { Name = "P320", NumberOfStudents = 4 };

List<Student> students = new List<Student>() { student1,student2};

List<Group> groups= new List<Group>() { group1,group2};

Now if we are to pass these two Lists(students and groups) then we have to create a ViewModels folder and create a ViewModel for this Controller of this action (Index) and the naming convention for ViewModel class is as follows🡪

ControllerNameActionNameViewModel in our case:

StudentIndexViewModel

StudentIndexViewModel model = new StudentIndexViewModel() { groups = groups, students = students };

return View(model);

@model MVC.ViewModels.StudentIndexViewModel;

@using MVC.Models

@{

foreach(Student student in Model.students)

{

<li>Name:@student.Name SurName: @student.Surname</li>

}

}

**ConfigureServices Method**

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews();

}

Here we tell the app to use MVC. We could have also specified AddMvc() method but it is more complex and it also includes apis as well which we don’t need if we want to use just MVC pattern.

**Shared folder for views**

If we have repetitive parts (header and footer) in our website then we can use a shared razor page (\_Layout.cshtml).

In views we create a shared folder then \_Layout.cshtml (Razor View Start) for our shared razor pages.

Then if we don’t want to repeat Layout part in every razor page we can create a ViewStart razor page so that it automatically targets that \_Layout.cshtml file we don’t need to specify it. It has to be directly in the view folder.

**ViewImports**

If we have multiple using statements in our razor pages then we can just create a ViewImports file (Razor View Import) and put all of our usings there. It also has to be directly in the view folder.

**appsettings.json**

In this file we store various information about our app like credentials or ConnectionStrings:

{

"ConnectionStrings": {

"DefaultConnection": "Server=WIN-DNH1G34J36A;Database=Academy;Trusted\_Connection=true"

}

}

**Integration of database into c# using EntityFramework**

EntityFramework is an orm (object relational mapping).

In order to integrate our database with ASP.NET Core using entity framework Core first we need to install a few packages using NuGet package manager.

Developers describe npm as "The package manager for JavaScript". npm is the command-line interface to the npm ecosystem. On the other hand, NuGet is detailed as "**The package manager for .** **NET**".

The packages tha we need to install are: EntityFrameworkCore itself, EntityFrameworkCore.SqlServer, EntityFrameworkCore.Tools (for commands like add-migration and etc.).

In order to interact with our database with EntityFrameworkCore we need a class that inherits from DbContext.So normally we create a DAL folder (Data Access Layer) and inside that we create a AppDbContext class that inherits from DbContext.

public class AppDbContext:DbContext

{

Public AppDbContext ( DbContextOptions<AppDbContext> options) :base(options)

{

}

public DbSet<Basket> Baskets { get; set; }

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

public DbSet<Favourite>Favourites {get; set;}

public DbSet<Order> Orders { get; set; }

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

public DbSet<Role> Roles { get; set; }

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

}

We then pass DbContextOptions to the base class(DbContext) to do any kind of task with the database.

**DbContextOptions** are the options that we set in the ConfigureServices method of StartUp to use either SQL or in memory connection and etc. In the ConfigureServices method, we then use services.AddDbContext<AppDbContext>(options) method to add those options to the base class (DbContext).

DbSets are our tables. If we want a many-to-many relationship then we just create two ICollections (Lists) of each other in them. For example:

public class Product

{

public List<Basket> Baskets { get; set; }

}

public class Basket

{

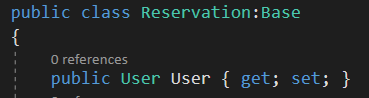
public List<Product> Products { get; set; }

[Required] // A Basket may not have orders but it must always have a customer

public User User { get; set; }

}

If we want a one-to-one relationship then only the dependent one’s model will have the model that it depends upon. For example, between reservation and user 🡪 one user can have one reservation and one reservation can be applied to only one user. So the reservation model is the dependent one and it will have the user property but the user model will not have the reservation property🡪



There are some constraints that EF does it by default and if we don’t want that then 🡪

[Column(TypeName = "varchar(200)")]

We add to the top of the property or [Required], [MaxLength(20)]

Lastly, in our Startup class, in order for us to interact with our appsetting.json file ,which we need to get the ConnectionString, we need to add a IConfiguration property ,which is in Microsoft.Extensions.Configuration namespace, to inject it to our Startup class so that it is available everywhere.

private IConfiguration \_configuration;

public Startup(IConfiguration configuration)

{

\_configuration = configuration;

}

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews();

string connetionString = \_configuration.GetConnectionString("Default");

services.AddDbContext<AppDbContext>(options =>

{

options.UseSqlServer(connetionString);

});

}

After this we add-migration the we update-database.

Now if we want to access our database in a controller we need to add that context field and inject it to that class 🡪

**(It has to be readonly)**

private readonly AppDbContext \_context;

public HomeController(AppDbContext context)

{

\_context = context;

}

HomeIndexViewModel homeIndexViewModel = new HomeIndexViewModel()

{

Baskets = \_context.Baskets.ToList(),

Categories = \_context.Categories.ToList(),

Favourites=\_context.Favourites.ToList(),

Orders=\_context.Orders.ToList(),

Products=\_context.Products.ToList(),

Roles=\_context.Roles.ToList(),

Users=\_context.User.ToList(),

};

return View(homeIndexViewModel);

**DbContext class**

Translator between our model classes and the database.



public class AppDbContext:DbContext

{

Public AppDbContext (DbContextOptions <AppDbContext> options ) : base(options){}

}

This is just a way to tell the app about our options. By options, I mean whether I want to use SQL Server or SqlLite and etc.

**Html.Raw(source) Method**

If we have an html element in our database such as <h1>Send <span>flowers</span> like</h1> then after we receive this we gotta convert it from a string into an html element. We do it by using Html.Raw(source) method

@Html.Raw(Model.Slider.Title)

**Wwwroot directory**

wwwroot folder is where we place our static files such as js, css, image files. When we use tilde sign (~) the place where our app is gonna look at is the root folder (~/img/…). To use static files from the root folder we have to add a middleware ,namely:

app.UseStaticFiles();

**Linq Query**

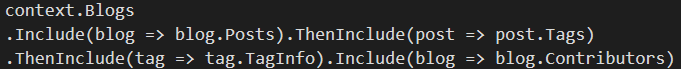
\_context.Categories.OrderBy(c=>c.Order).ThenBy(c=>c.id)

\_context.Products.Include(p=>p.Category)- Joining tables

Order it first by Order field then by their ids if their order is identical.

**ThenInclude( ) Method**

ThenInclude( ) takes the descendant and Include always takes the parent🡪



Include goes back to the parent. And ThenInclude keeps going into child models.

**Partial Views and TagHelpers**

If we have a component that gets repeatead in some parts of our web application, then it is reasonable to use partial views. We create partial view file in our shared folder and the naming convention is🡪

\_ComponentNamePartial for example \_ProductPartial or \_Header

Then in our partial if we receive data from a controller then we need to include our viewmodel.

Another case where we would consider using partials views is when we used ajax (where we needed to send data to a partial view from ajax).

To use partials, first we gotta import partial tag from Taghelpers so we need to import TagHelpers 🡪in our ViewImport Razor Page

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

Then we include:

<**partial** **name**="\_ProductPartial" />

Wherever we want to use the product component.

Our partial pages are dependent on the called view’s page controller because the model that partial views can use is the model that is passed from called view’s model which is received from its controller.

**Cache Busting**

When we make changes in our js file which is inside our wwwroot directory we have to clean cache so that we can see changes or **ctrl+shift+R** or we cann add the following TagHelper to tell our app to use the latest version of the js file.

**asp-append-version**="true"

This is used for <link>, <script> and <img> to ascertain that the browser loads the latest version of the resource.

<script **src**="~/js/script.js" **asp-append-version** ="true"></script>

In the browser it will look like this🡪

<script src ="/js/script.js?v= sevhZ70HL0FUVxPoCap6XWbt47gbwfqsiCG77tXNnKk">

</script>

The framework generates a random string for us and when the browser sees a change in the string it loads the latest version of the file and the string then is changed.

**View Components**

View components are similar to view partials but they are more powerful. Firstly, view components have their own controller and they are not dependent on the called view’s controller. Their model is passed from their own controller.

To create view components, first we create a ViewComponents folder in our project and create all of our viewcomponents there.



The classed have to inherit from ViewComponent class. They all will have just one action namely, Invoke( ) method to call the view component and It has to return IViewComponentResult

public IViewComponentResult Invoke()

{

return View (

new FooterInvokeViewModel {

Footer=\_context.Footers.FirstOrDefault(),

FooterCategories=\_context.FooterCategories.Include(fc=>fc.footerCategorySections).ToList(),

FooterCategorySections=\_context.FooterCategorySections.ToList()

}

);

}

This view will return:



Then we create a components folder in the shared folder and create all of our component folders



In component folders we name them just Default.cshtml because the IViewComponentResult will open the view page which is named like this in its corresponding folder. Basically, we always have just one action (Invoke( ) method) in our view components so that’s why we it is named Default.

The corresponding name for the components is selected from the ViewComponent class!

Finally, in the place where we wanna render view components we place the following🡪

@await Component.InvokeAsync("Footer")

**AJAX to request to our action**

So we have a load more button in our product page and as we click at that it loads products 2 by 2.

So we have a GetPartial( ) action in our Product controller which returns a JSON🡪

public IActionResult GetPartial(int count)

{

ICollection<Product> products = \_context.Products.Include(p => p.Category) .Skip(count).Take(2).ToList();

return Json(products);

}

Then in our js file🡪

let count = 4;

$(document).on("click", "#load", function () {

$.ajax({

method: "GET",

url: "GetPartial/" + count,

success: function (response) {

count += 2;

for (let item of response) {

var content = `

<div class="col-sm-6 col-md-4 col-lg-3 mt-3">

<div class="product-item text-center" data-id="@product.Category.Name.ToLower()">

<div class="img">

<a href="">

<img src="/img/${item.imageName}" class="img-fluid" alt="">

</a>

</div>

<div class="title mt-3">

<h6>${item.name}</h6>

</div>

<div class="price">

<span class="text-black-50">Add to cart</span>

<span class="text-black-50">$${item.price}</span>

</div>

</div>

</div>

`;

$("#product").append(content);

}

}

})

})

In the URL Product part is already included that’s why it is exluded here so that it is Product/GetPartial/count. If the data that we send to our action ,via ajax, is an integer then we can just use slash + our integer. If it is a string then we need to put a question mark there🡪

“Product/GetPartial?str=”+str

str vairable name must match the paramater name of the action(GetPartial’s).

**Ajax for Search input**

$(document).on("keyup", "#input-search", function ()

{

let str = $(this).val();

$.ajax({

method: "Get",

url: "Product/Search?searchedStr=" + str,

success: function (res) {

$(".search\_\_container li:not(:first-child)").detach()

$(".search\_\_container").append(res);

}

}

)

});

When we receive the response it is just some html elements.

Our Search Action🡪

public async Task<IActionResult> Search(string searchedStr)

{

if (string.IsNullOrWhiteSpace(searchedStr))

{

return PartialView("\_ProductSearchPartial",

new List<Product>());

}

List<Product> products = await \_context.Products.Where(p => p.Name.ToLower().StartsWith(searchedStr.ToLower()))

.Take(5).ToListAsync();

if (products.Count==0)

{

ViewBag.ProductCount = 0;

}

return PartialView("\_ProductSearchPartial",products);

}

And our Partial View🡪

@model List<Product>

@{ if (ViewBag.ProductCount==0)

{

<li>Not found</li>

}else{

foreach(Product product in Model)

{

<li>

<div class="container-fluid">

<div class="row">

<div class="col-4"><img class="w-100" src="~/img/@product.ImageName" /></div>

<div class="col">@product.Name</div>

</div>

</div>

</li>

}

}

}

**Data Annotations**

Data Annotation provides attribute classes that are used to define metadata for ASP.NET MVC and ASP.NET data controls. We also use data anotations in front end validations such as [EmailAddressed]. In itself in includes regex for emails.

[Required]

And in this namespace

using System.ComponentModel.DataAnnotations.Schema;

we have 🡪

[Column(TypeName = "varchar(200)")]

public User User {get;set;}



The Name is the name in the database, order and etc.

public class BaseEntity

{

[Key]

public int StudentId { get; set; }

}

Sets the key of the table. (Primary Key)

[NotMapped] – the property is ignored by the database.

EntityFramework will throw an exception if it doesn’t find Id named property so that it puts that as an identifier. So in this case we can use [Key] data annotation.

public class DrivingLicense{

[Key, Column(Order = 1)]

public int LicenseNumber { get; set; }

[Key, Column(Order = 2)]

public string IssuingCountry { get; set; }

public DateTime Issued { get; set; }

public DateTime Expires { get; set; }

}

If we want to change the default Table Name that EF takes from the name of the class, then we can specify it🡪

[Table("StudentsInfo")]

public class Student{

[Key]

public int StdntID { get; set; }

[Required]

public string LastName { get; set; }

[Required]

public string FirstMidName { get; set; }

public DateTime EnrollmentDate { get; set; }

public virtual ICollection<Enrollment> Enrollments { get; set; }

}

Now, we have have something like this in our poco class🡪

public Category Cat { get; set; }

Then it will not understand that Cat, which is short for Category, is actually our foreign key because it knows that from the name of it and since the name is diiferent EF is going to get confused. So to fix this, we use ForeignKey Annotation which is inside the following namespace🡪

using System.ComponentModel.DataAnnotations.Schema;

[ForeignKey("CategoryId")]

public Category Cat {get; set; }

Now here, it will assign Cat to a foreign key and the name in the database will be CategoryId.

We can also add an error message for our data annotations🡪



If we don’t specify the error messages then it sends the default error messages.

**Dependency injection vs Inverion of Control**

The Inversion-of-Control (IoC) pattern, is about providing any kind of callback (which "implements" and/or controls reaction), instead of acting ourself directly (in other words, inversion and/or redirecting control to external handler/controller).

The Dependency-Injection (DI) pattern is a more specific version of IoC pattern, where implementations are passed into an object through constructors/setters/service lookups, which the object will 'depend' on in order to behave correctly.

The Dependency Injection pattern involves 3 types of classes.

**1)Client Class:** The client class (dependent class) is a class which depends on the service class

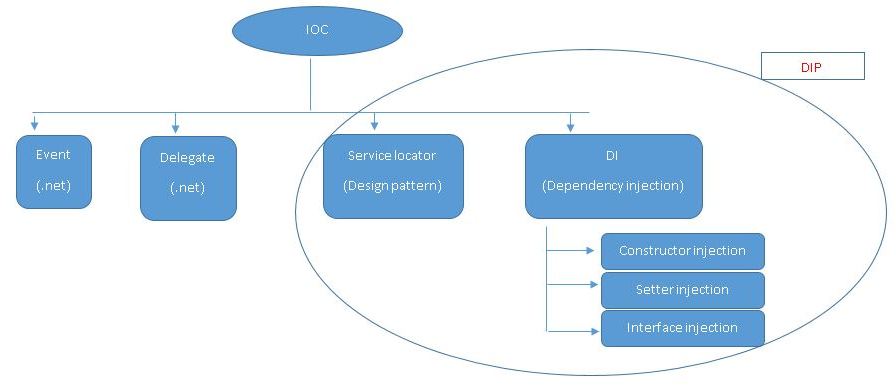
**2)Service Class**: The service class (dependency) is a class that provides service to the client class.

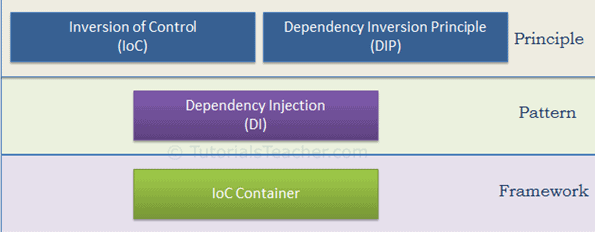
**3)Injector Class:** injects the service class object into the client class.

The injector class creates an object of the service class, and injects that object to a client object. In this way, the DI pattern separates the responsibility of creating an object of the service class out of the client class.

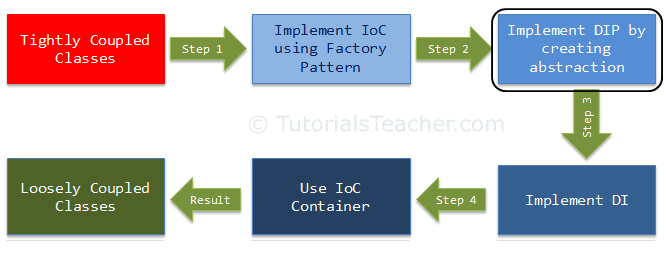
**Dependency injection is a pattern through which IoC principle is implemented and the act of connecting objects with other objects or injecting objects into objects is done by container rather than by the object themselves.** **IoC Container (a.k.a. DI Container) is a framework for implementing automatic dependency injection. It manages object creation and it's life-time, and also injects dependencies to the class.**

**DI,DIP,IOC are aimed to achieve loose coupling.**





The IoC container is a framework used to manage automatic dependency injection throughout the application, so that we as programmers do not need to put more time and effort into it. There are various IoC Containers for .NET, such as Unity, Ninject, StructureMap, Autofac, etc. We will learn more about this in the IoC Container chapter.



**SingleOrDefault vs FirstOrDefault**

So we have got:

\_context.AboutSections.Single()

Returns the only element of a sequence, and thows an exception if there is not exatcly one element in the sequence.

\_context.AboutSections.First()

It just returns the first element of a sequence.

\_context.AboutSections.SingleOrDefault()

Returns the only element of a sequence or if there is none then it gives a default value, this method thows an exception if there is not exatcly one element in the sequence.

\_context.AboutSections.FirstOrDefault()

Returns the first element of a sequence or if there is none then it gives a default value.

**Preventing the loop via NewtonSoftJson**

When ajax receives data (data, of course, is in the json format) from a controller, if there is a loop it gives the 500 status code which is the loop problem. For example, we have got a product object and inside this object for relationship porposes we have category property which itself is also an object and it includes a list of products and this continues, leading to an infinite loop. 🡪

{

Product{

Id:3,

Name:”Majecty”,

Category:{

Id:1,

Name:”Flowers”,

Products:{

……

}

}

}

}

So in order to avoid this we use NewtonSoftJson. First, we need to download it via Nuget package manager🡪



Then in our StartUp class, we add the following service🡪



**Session storage and Cookies**

Sessions live throughout a specific period. We need to both specify it in our services and middleware🡪



Here we are okay with just AddSession( ) but if we leave it like this then IdleTImeout is gonna be 20 mins by default.

Also we specify it in our middleware🡪

app.UseSession();

Then in our controller🡪

HttpContext.Session.SetString("test", "hey");

And to get a Session we just use🡪

HttpContext.Session GetString(key)

**Cookies**

Most of the time we use cookies.



ProductCardViewModel🡪



To append an item into cookies🡪



Cookies are hashted as sessions but they are available even after the browser is closed.

**Asp-route-data**

To send data from our view to our action we can use this taghelper from a tag🡪 note that “data” can be anything (id,name and etc.)

<a **asp-controller**="Product" **asp-action**="AddToBasket" **asp-route-id**="@product.Id">Add to cart</a>

We will receive id from our view.

**Admin Panel**

Admin panel is a different project. So we can either create a new one or create our admin panel project inside our existing project as an area.

So we click at New Scaffolding item and then we type Admin.

Inside our wwwroot directory we create an Admin folder and put all the Admin’s static files there.

After that we need to add endpoints middleware for our Admin page. So we take this🡪

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "areas",

pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}"

);

});

And put it before our main endpoint middleware.

The controllers of Admin don’t know that they are in another area, so we gotta tell them that they are in the Admin area with the data Annotation🡪



**Asp-area**



With this we tell the a tag it is going to be the Admin area.

**Implementing DI for AppDbContext**

Once we create an IAppDbContext interface then we implement this interface in our AppDbContext class. Then we put IAppDbContext everywhere. Then in our services we gotta add the following so that our app knows when IAppDbContext is required, create an AppDbContext and pass that in.



**Request types**

All requests are sent using an "HTTP method". The method designates the type of request being made to the web server. The most common types of request methods are GET and POST but there are many others, including HEAD, PUT, DELETE, CONNECT, and OPTIONS.

Post request is for sending new data, put is for updating there is also patch which is for updating one field.

In Dot Net we can specify requests for the same named actions 🡪



Or



So if we have the same named functions that do different things like one is for get request and the other is for post then we can use this data annotaion.

By default if we don’t specify it, it is a get request.

**Post request via Form**

In our input element’s atributes we can specify for which field this input’s values is going to be for intsance🡪



When we send a model in to our razor page, we can view all the available fields that we can put for the inputs like Name, Description and etc.

In the form itself we gotta specify the contoller name and the action name not in the submit button element!

Important! For our form action, above that action we must add one data annotation against attackers. Since, it is possible to send a post request outside of our website, we gotta prevent it.



Basically, every form has a request verification token which gets updated each time. So we need to validate the request through our request verification token.

The verification process explained🡪 so when the post request is submitted our action receives the token and then it validates it to see if it is from our application.

We can have multiple parameters for a post request and it is not a good practice to pass them all. Instead, we can just put a model as an argument there🡪



Our razor page takes all the data from our form and puts that in our model. Then in our action if the model that is passed matches the action’s model it takes it as a parameter.

Finally, in order to force our constraints to be taken into account we gotta use ModelState object.



So if the model state is not valid then return the view. When it returns the view it returns it with error messages and to display these error messages we gotta have an html element with an attribute of asp-validation-for 🡪



**Partial scripts (RenderSection)**

To use partial scripts, at the end of our layout page we add the following🡪



The name of our partial script is gonna be “Scripts”. The second parameter specifies whether this partial is gonna be forced to be used by all the pages or not. So it is false here which means it’s optional.

Then in the page which we will render🡪



We put everything inside those curly braces.

**Validations cdns**

When we submit a form, if the requirements are not met the request gets sent anyways. If we want a front-end checker so that it doesn’t send a request for nothing we gotta add two cdns. First one is jquery validation cdn the second one is jquery-validation-unobtrusive cdn for asp.net core. They are gonna be repetitive so we create a partial view for them.

**Note that the order also matters!** First the standard jquery validation cdn has to be placed then the jquery-validation-unobtrusive cdn. The main page’s script has to be the last. So we need both of them so that unobtrusive cdn works fine. But the standard jquery cdn has to be on top of everything.







**CK editor**

CKEditor 5 provides every type of WYSIWYG (what you see is what you get) editing solution imaginable. From editors similar to Google Docs and Medium, to Slack or Twitter like applications, all is possible within a single editing framework.

So if we want to add bold words or sth like this we can use this framework to quickly add editors to our text inputs (text areas).



It will get converted into html elements such as <b></b> and etc. Then we will store it in our database and when we take this from our database we will pass it through HTML.Raw( ) method.



ClassicEditor.create(element).then() and etc.



Asp-for is gonna put the id there for us.

**Update**



When we click at the update button it creates a view for us🡪



When we submit this,first jquery validations will check the requirements if they are not met then it will return a view with error msgs. Then the request will get sent and once this model is in our action, our action will check the model state of the category that we passed in whether the requirements are met or not. After that, we also check the id whether the id is equal to the passed category’s id. If not the badRequest( ) is returned. Finally, we check whether this category exists in our database. If not the NotFound( ) is returned. If we pass all of these then we just Update our category



Or just \_context.Update(category)

**ActionName attributre**



If we want to have a different name for our action we can use ActionName( ) attribute. We use this if we have a conflict with names in our controller (method overload problems).

**File Uploads**

If we are to upload a file to our server, we first have to upload the file to our img folder in wwwroot directory. Then we save the name of the file (image) in our database. First we create a IFormFile property in our model so that we can interact with the file but we gotta specify the attribute [NotMapped] so that it doesn’t get created in our database. We are not gonna store the file itself, but the name of it.

When Uploading an image we have to check several things:is the content type contains an image word? So that we know that it is an image (it can be jpeg,png and etc.) it also has to be specified in the front-end as accept🡪



The accept attribute specifies a filter for what file types the user can pick from the file input dialog box. Image/\* means that all of the types of the image(jpeg,png etc.). accept="image/png, image/gif, image/jpeg".

Then we can check the size of the file to be a particular size. If it is not of the specified size then we can create a model error and send it to our File property in the model



Our form has to contain enctype as well. The enctype attribute specifies how the form-data should be encoded when submitting it to the server.



application/x-www-form-urlencoded - Default. All characters are encoded before sent (spaces are converted to "+" symbols, and special characters are converted to ASCII HEX values)

multipart/form-data - This value is necessary if the user will upload a file through the form

text/plain- Sends data without any encoding at all. Not recommended

After all the steps meantioned befored, we set our image name. Then finally we gotta upload our file to our img folder in the wwwroot directory. First we find the path to our wwroot folder(through IWebHostEnvironment) and combine it with the img word and the file name so that we can create a filestream. Then we copy the file to that fileStream and save the image in our database (with the name of course)



Note that Path.Combine works for all the platforms (in windows the combination is via a single slash / but on MacOs it can be diffenet)

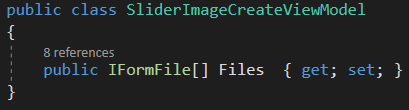
**Guid (globally unique identifier)**

We also have to add a random string to our image name every time we create one. We do this so that in the next time when we create a file with the same name it doesn’t give an error they are gonna have random strings at the beginning of them. 🡪

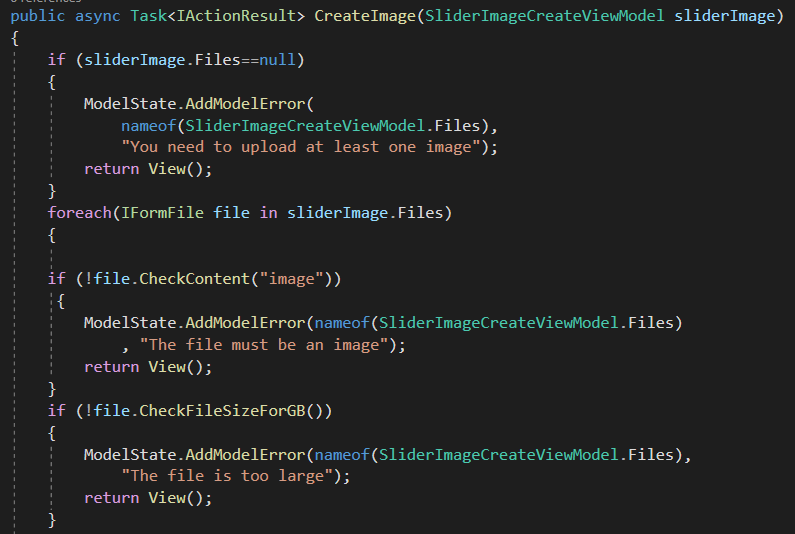


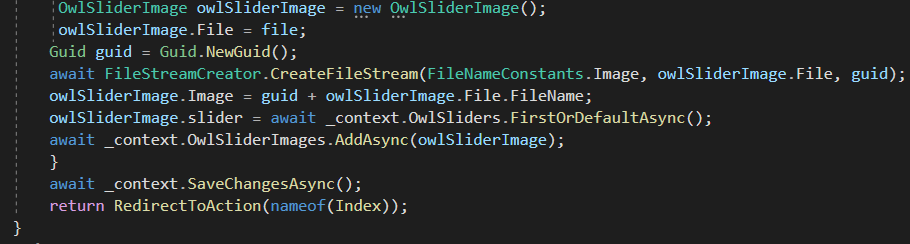
**Multiple File upload**

In order to upload multiple files and send it to our action, needles to say we will need an array of IformFiles so the best way to do this is to create a viewmodel for this particular actions🡪



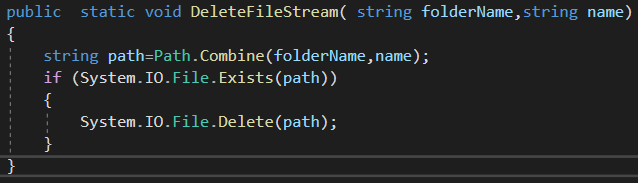
Then when we receive this model we are gonna loop through each file 🡪





**Delete a file**

First we need to get the path of the file along with the name of it to delete it from the img folder.



folderName is the combination of IWebHostEnvironment.WebRootPath the folder name (img in wwwroot) and the name of the file which is guid+FileName. So then the value of the path variable that is mentioned in the above example is 🡪 “~/img/wwj23picofme.jpeg”

then we check whether it exists in the img folder or not to make sure that it is there before we remove it. We use System.IO.File.Exists(path) method here.

**Note that** every controller has its File class so to avoid the conflict we gotta use the entire namespace (System.IO.File).

**Model Binding**

Controllers and Razor pages work with data that comes from HTTP requests. For example, route data may provide a record key, and posted (submited) form fields may provide values for the properties of the model. Writing code to retrieve each of these values and convert them from strings to .NET types would be tedious and error-prone. Model binding automates this process. The model binding system:

* Retrieves data from various sources such as route data, form fields, and query strings.
* Provides the data to controllers and Razor pages in method parameters and public properties.
* Converts string data to .NET types.
* Updates properties of complex types.



https://contoso.com/api/pets/2?DogsOnly=true

Model binding goes through the following steps after the routing system selects the action method:

* Finds the first parameter of GetById, an integer named id.
* Looks through the available sources in the HTTP request and finds id = "2" in route data.
* Converts the string "2" into integer 2.
* Finds the next parameter of GetById, a boolean named dogsOnly.
* Looks through the sources and finds "DogsOnly=true" in the query string. Name matching is not case-sensitive.
* Converts the string "true" into boolean true.

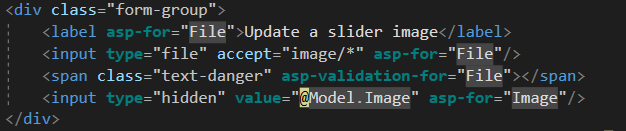
The framework then calls the GetById method, passing in 2 for the id parameter, and true for the dogsOnly parameter.

Model binding maps the incoming HTTP request data to the respected controller action method parameters by their names.

When we submit a form, the model’s properties that we get in our action are actually collected from the inputs. So if we don’t pass any property from the input that property is gonna be null even though it wasn’t in the view. **The model binder binds everything from the input values to our model if, of course, that input has asp-for taghelper**.

That’s why we need to path Model.Image manually because Model binder is not gonna bind it to our model if it doesn’t have it in hand.

So for it we create a hidden input and pass the value from our model to it🡪



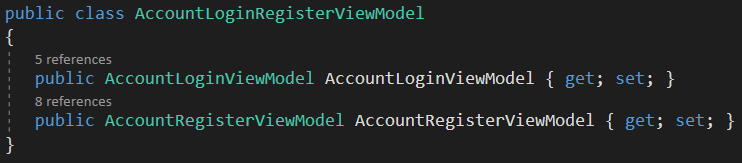
**Model binder only binds within one form tag!** So if we try to bind something outside of the form element then that model will not be bound for instance 🡪

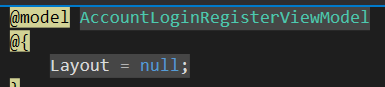


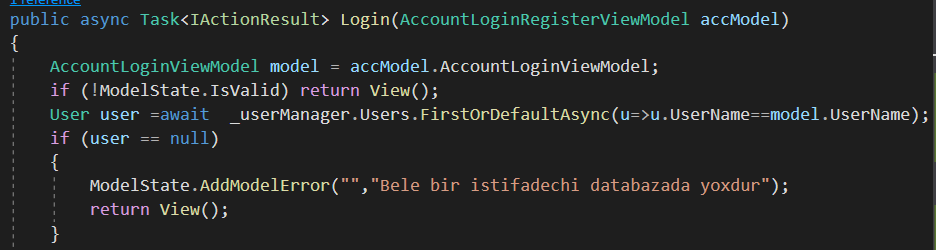
It will not be bound because it is outside of the form element. So there is no point in using asp-for attribute for that input.

**Multiple ViewModels in one ViewModel**

If we have one view for two different actions (login and register in our case) then we can exclude the @model from the view and assign the corresponding model to our view, meaning that we don’t use strongly typed view here so that we don’t get compile-time error when assigning values for asp-for attributes. The second way is to use 2 viewmodels in 1 viewmodel 🡪





As we receive AccountLoginRegisterViewModel from our view we need that viewModel to take the needed viewModel (AccountViewModel).

**ModelState Error Binder**

Suppose we have Exist bool in our model. The way the name of it is in the asp-for attribute must be the same as in the AddModelStateError method 🡪

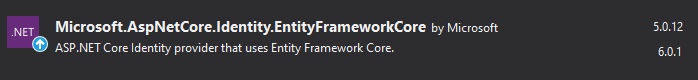




So we should be careful when we use nameof expression.

**Identity**

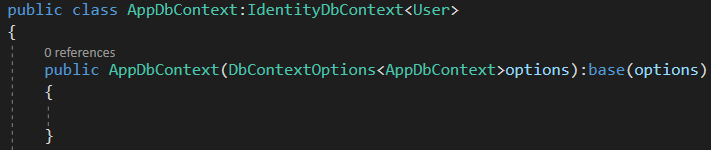
Normally, we ourselves can create users, roles, claims tables but instead we can just extend IdentityUser class so that we have all the properties of this class and the methods are virtual meaning that we can override them. So in order to do this we have to install one package, namely EF identity



After this we extend IdentityUser to have all the methods and properties of it.

Other than IdentityUser we can also extend IdentityRole, IdentityRoleClaim, IdentityUserClaim classes as well.

After these steps, we need to also extend AppDbContext so that DbSets are placed there and since IdentityDbContext inherits from DbContext we will not have any problems.



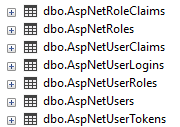
If we don’t pass anything for IdenitityDbContext then the default values🡪



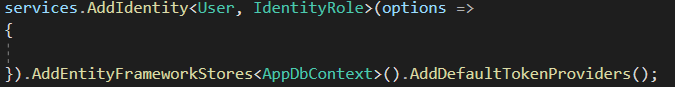
It will create a default IdentityUser but if we put our user there then it will be Tuser which we passed.

If we don’t pass IdentityRole in IdentityDbContext then the default is IdentityRole.

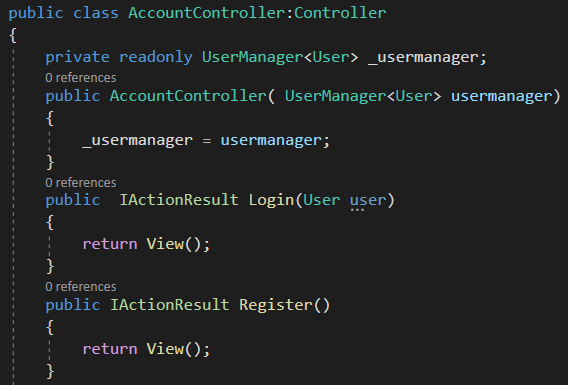
NormalizedName – for quick checkings names will be stored in uppercase.

 will be created

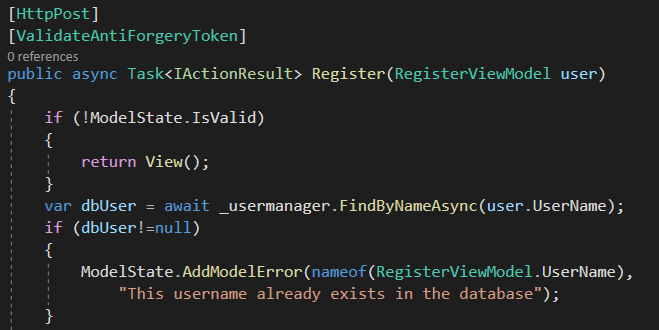
Our IServiceCollection 🡪



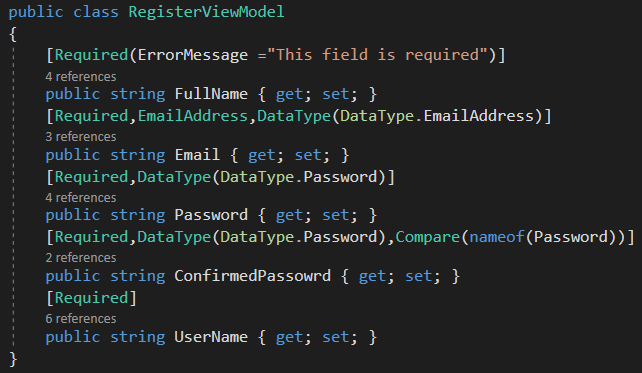
After this we create a AccountController for all the operations with users. This controller will have to inject UserManager for operations with the user 🡪



Then we create HttpPost Register 🡪



It is always better to have a viewmodel in each view. So we will receive RegisterViewModel 🡪



For email property, we have to specify EmailAddress attribute for regex and validation. We need to specify the datatype so that the input in the view has the type of EmailAddress. Datatype(Datatype.EmailAddress)

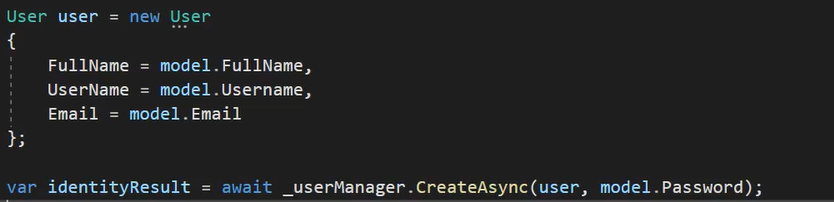
And for password we also have DataType attribute so that the input type is password and therefore it will display the input values with starts. And also we have to have ConfirmedPassword property to compare the passwords for the equality.

Once we receive the RegisterViewModel in our Register post action, we need to check the username to see if there is a user with this username. If there is then we will add a ModelState error to UserName property of RegisterViewModel.

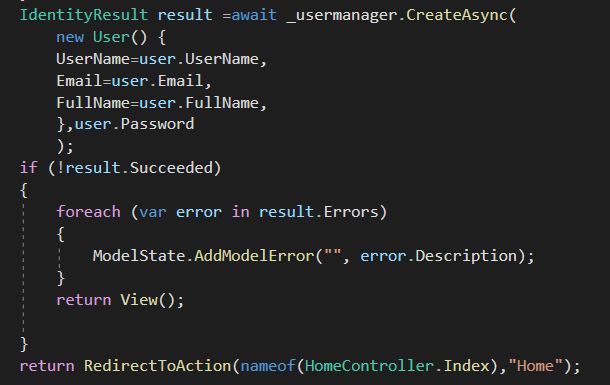
As we can see, we make requests to the database with the help of \_userManager field which we inject to our Account controller.

As the name implies, UserManager class manages the users. It help to get the user information and create user information.

Then we add the following🡪



Here, we create a user and add it to the database via \_userManager. \_userManagaer.CreateAsync(user,password) – password parameter hashes the password and stores it.



We also check if it creates the user successfully. And if it has errors then we loop through the errors and we display them in our view by having asp-validation-summary taghelper🡪



**Get the signedIn user**

We can get the current signedIn user using \_usermanager 🡪



**Get the signedIn user’s info**



It will give the user's userId



It will give the user's userName

**ConcurrencyStamp and SecutiryStamp methods**

**ConcurrencyStamp-** As the name states, it's used to prevent concurrency update conflict. For example, there's a UserA named Peter in the database 2 admins open the editor page of UserA, want to update this user.

Admin\_1 opened the page, and saw user called Peter.

Admin\_2 opened the page, and saw user called Peter (obviously). Admin\_1 updated user name to Tom, and save data. Now UserA in the db named Tom.

Admin\_2 updated user name to Thomas, and try to save it.

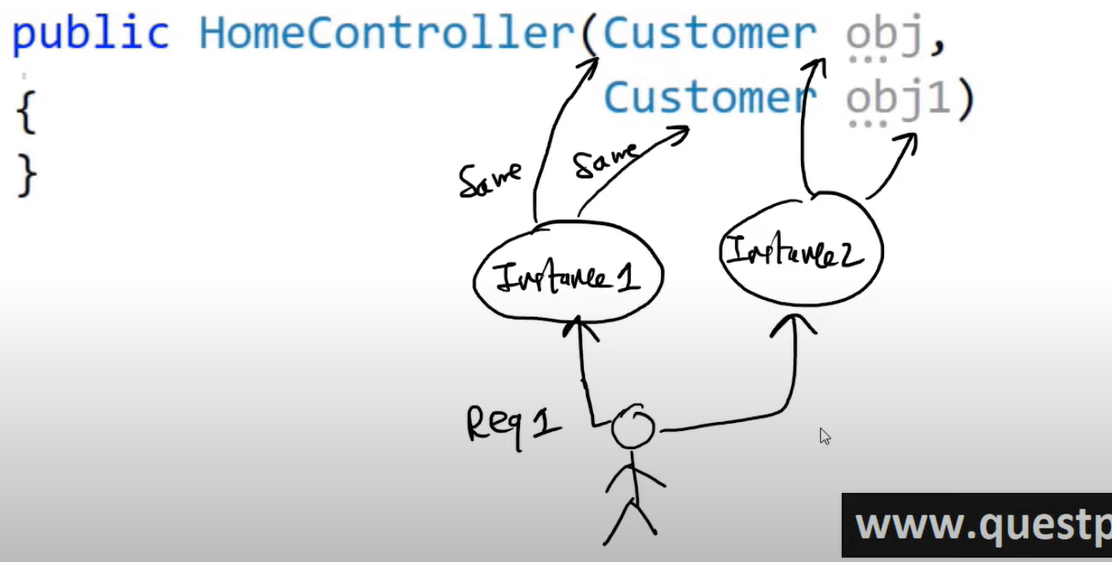
What would happen if there's no ConcurrencyStamp is Admin\_1's update will be overwritten by Admin\_2's update. But since we have ConcurrencyStamp, when Admin\_1/Admin\_2 loads the page, the stamp is loaded. When updating data this stamp will be changed too. So now step 5 would be system throw exception telling Admin\_2 that this user has already been updated, since he ConcurrencyStamp is different from the one he loaded.

**SecurityStamp** does the same thing but with information related to the user's credentials. If it displaces or changes the password SecurityStamp changes, invalidating old cookies and other possible security issues. So when user’s credentials change it also changes.

**Scoped, Transient**

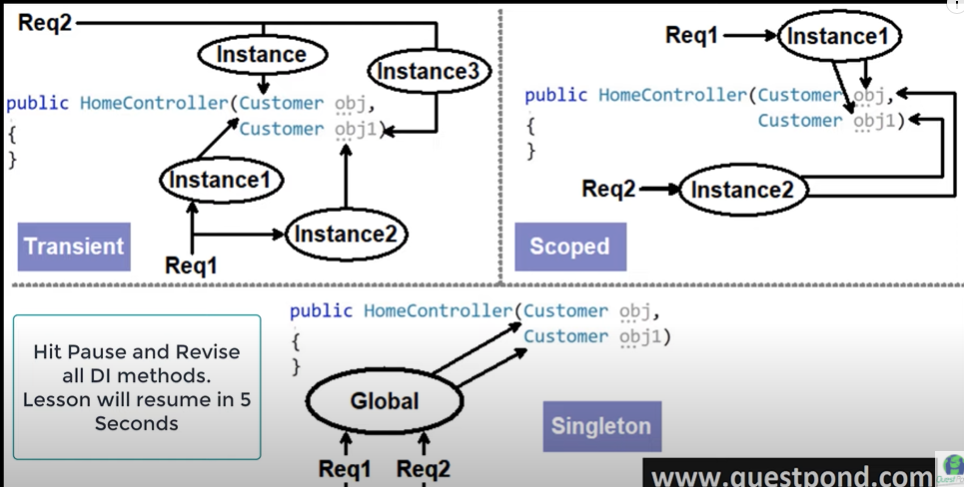


AddScoped – for example, we are in the category controller we have delete action and in this action if AddScoped service is added then no matter however requests we make in the delete action the same instance of AppDbContext is gonna be used.



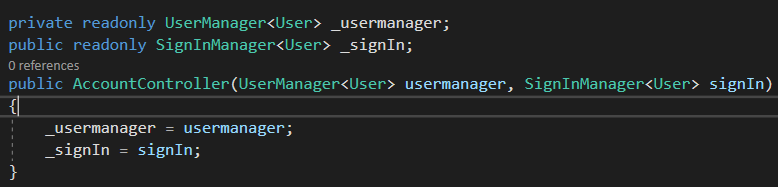
Here when we send a request tho this controller, if it is addScoped then it will create one instance and assign that insance to both of these objects (obj and obj1 are the same).

However, in Transient, it will create a new instance for each object. Therefore in transient objects are always different.

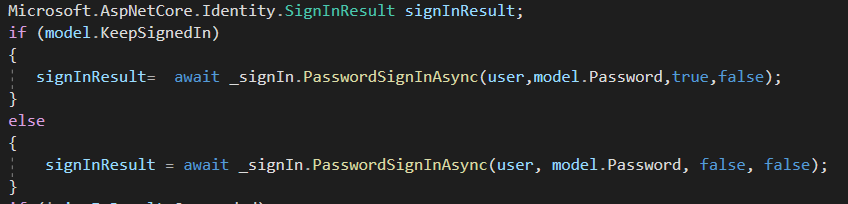


**SignInManager**

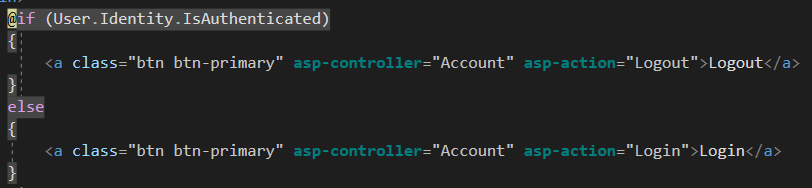
To sign in, we need to then inject SignInManager to our Account controller🡪



Now, when we log in to our account we use PasswordSignInAsync method of SignInManager that takes the following parameters🡪user, password, isPersisten(for keeping singed in), lockoutOnFailure



If isPersisten is true then it is gonna keep the token in our cookies otherwise it creates a session and once the session ends it gets deleted.



If the user is authenticated meaning that he is logged in then Logout button will be displayed. But for this to happen we need to first add our UseAuthentication( ) method to our middleware so that it authenticates the user.



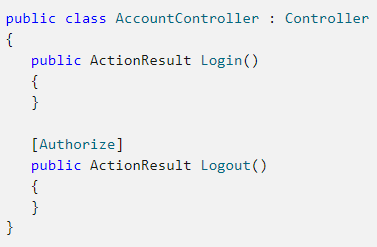
**Authorization**

We can add authorize attributes to our actions so that only authorized users access the specific action. First we add our middleware, namely UseAuthorization() mehod to our configure method (where all the middleware goes).

**Note that** this method has to be placed between UseRouting() and UseEndpoints() methods so that it works properly 🡪

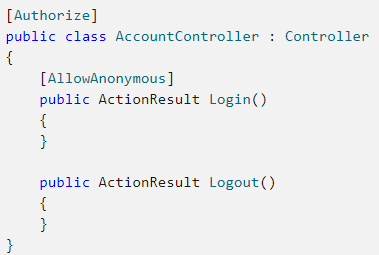


So now we can add [Authorize] attribute wherever we want.



Now only authenticated users can access the Logout function which is normal.

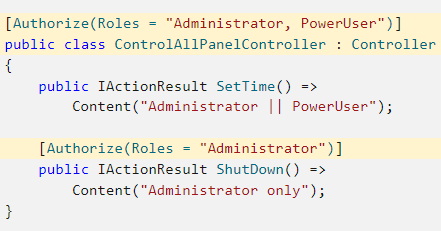
We can also use the AllowAnonymous attribute to allow access by non-authenticated users to individual actions. For example:



This would allow only authenticated users to the AccountController, except for the Login action, which is accessible by everyone, regardless of their authenticated or unauthenticated / anonymous status.

By default, the authorization is AllowAnonymous.

Access to an action can be limited by applying additional role authorization attributes at the action level:

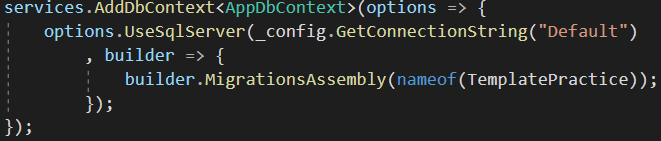


**Data Seeding**

Suppose that we don’t have any data in our database so our app is gonna be ugly without any data. In order to display our app we need mock data.

Our first step is to add the second parameter of UseSqlServer( ) method, namely we need to add SqlServerDbContextOptionsBuilder for the auto migration to know which assembly to use for migrations.

And then we add MigrationsAssembly( ) method So that it knows which assembly it has to use for our migrations.



After this, we need a place where we can call this method. Obviously, this place has to be the one which always gets used when our app runs.

Mainly, 2 places are used for this purpose. The first one is right in the program class. And the second one is in our middleware.

Middleware is the most appropriate one where we can initialize our InitializeData method to seed our database with data.

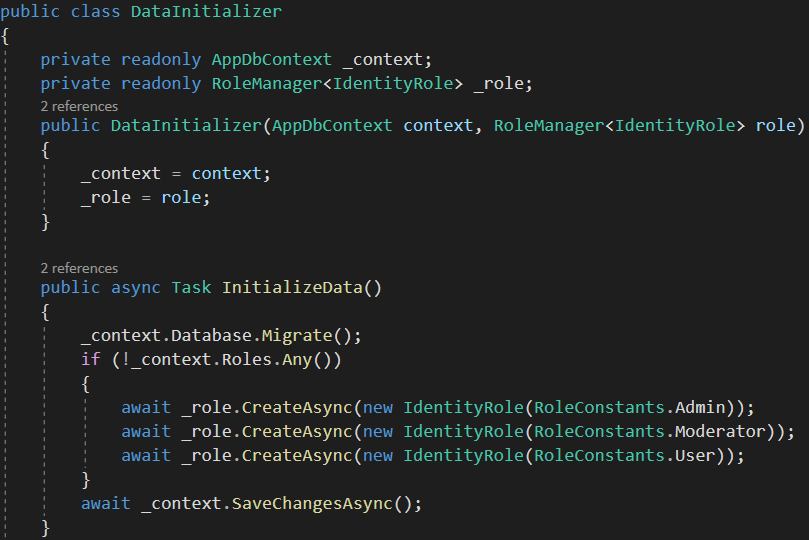
We also need to check tables to see whether they have something or not. If any of the tables is empty, then we add data to it. So to do this, we need to first create a DataInitializer class with a InitializeData method in the DAL folder that adds dummy data to our database if we have nothing in the corresponding table.

We also add \_context.Database.Migrate( ) method so that if our databse doesn’t exist it is automatically added along with our migrations so that late we can use migrations to update our database. Not to be confused with \_context.Database.EnsureCreated( ) 🡪

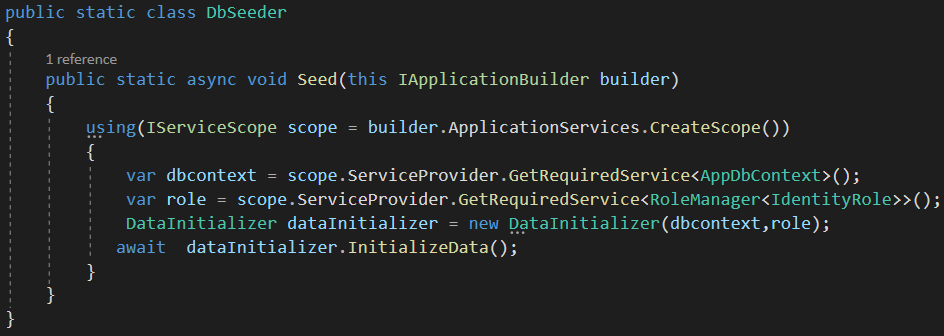
**context.Database.EnsureCreated()** is new EF core method which ensures that the database for the context exists. If it exists, no action is taken. If it does not exist then the database and all its schema are created and also it ensures it is compatible with the model for this context.

**Note**: This method does not use migrations to create the database. In addition, the database that is created cannot later be updated using migrations. If you are targeting a relational database and using migrations, we can use the DbContext.Database.Migrate() method to ensure the database is created and all migrations are applied.

Our DataInitializer class 🡪



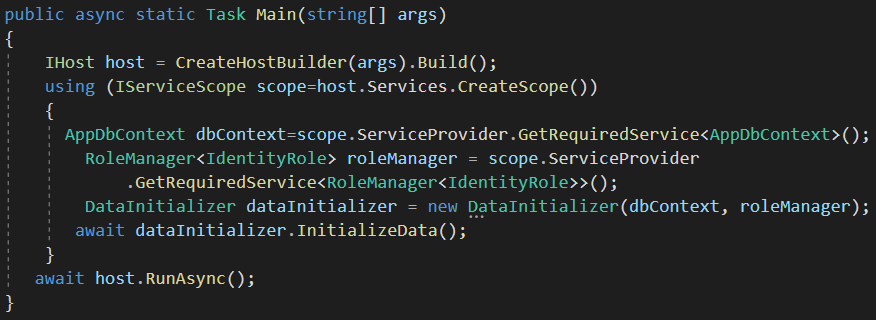
To use this method in our **middleware**, we have to create an extension method for IApplicationBuilder and in this extension method we will create an instance of this DataInitializer class to use this method.



As we know, DataInitializer class’s constructor wants 2 parameters which are our dbcontext and rolemanager. It is wrong to create an instance of these classes here. As we know, our controllers have access to the IOC container, however this outside classed don’t know about the IOC container so we need to first create that container and get our dbcontext and rolemanager from our IOC (inversion of control) container. To do this we need to first create a scope (IOC container). Then we take our dbcontext and rolemanager from the IOC container via GetRequiredService<T>( ) method.

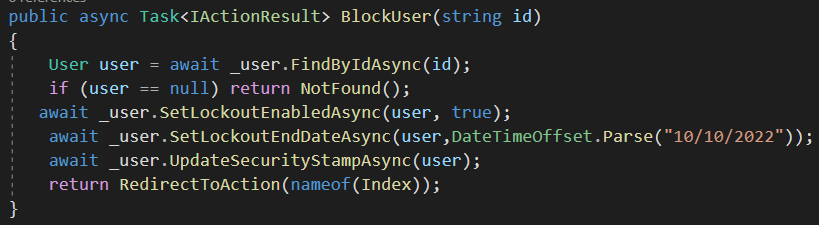


To use InitializeData in our Program class 🡪



**Blocking the user (Lockout the user)**

To lockout the user first we need to use the following methods🡪

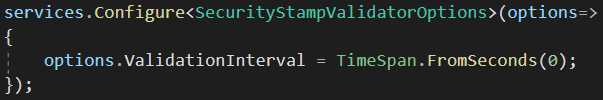


The LockoutEnabled property indicates, that a user CAN be locked out. It does NOT mean, that the user is currently banned. This way we could setup "normal" users with LockoutEnabled=true and some "special admin users" that have LockoutEnabled=false. (Would be bad if "evil people" lockout your admins by simply trying to login a few times - just as an example, where this could make sense.).

It is actually the SetLockoutEndDateAsync( ) method that locks out the user.

After this we need to update the security stamp of that user so that he gets prompted to the login view.

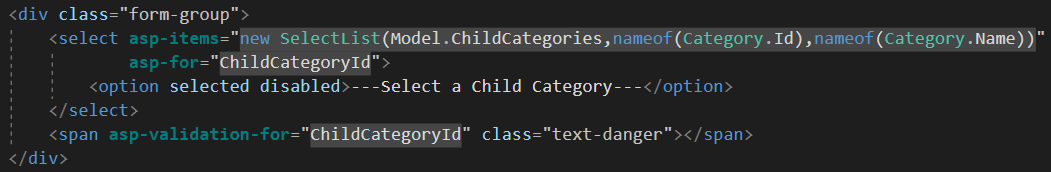
Additionally, we also need to zero ValidationInterval so that when the security stamp is updated then take that user to the login page. set it to Zero for the immediate logout🡪



The Security Stamp is checked at a frequency set in configuration. The default is the same as the auth cookie timeout if not set in the startup. The default frequency at which it checks the security stamp is 30 minutes.

**SelectList and asp-items**

Normally, if we have a select tag in our razor page then we have to have a loop. To make our lives easier with select tags, .NET provides SelectList class to automatically put the options of the select tag. 🡪





The first argument to be passed is the collection. The second argument is the value for each option and the third parameter is the text field for each option.

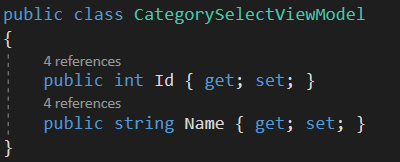
**Select query**

If we don’t need all the properties of a model then we can create a view model for that, namely select view model.

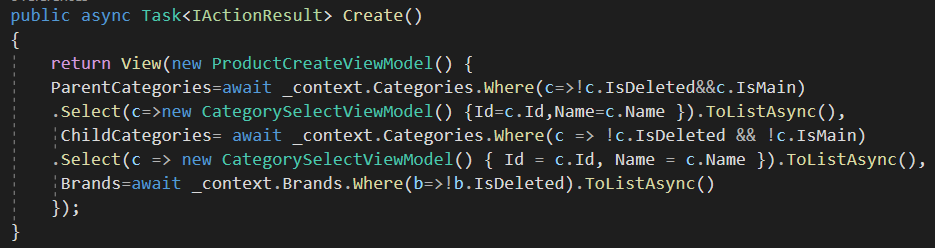
For example, let’s say that we need a collection of categories. And we only need ids and names of the categories so 🡪



🡪



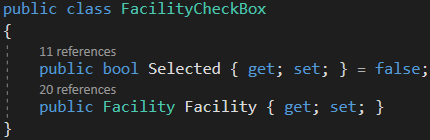
And to select a model from the database with the specific fields we will use Select query 🡪

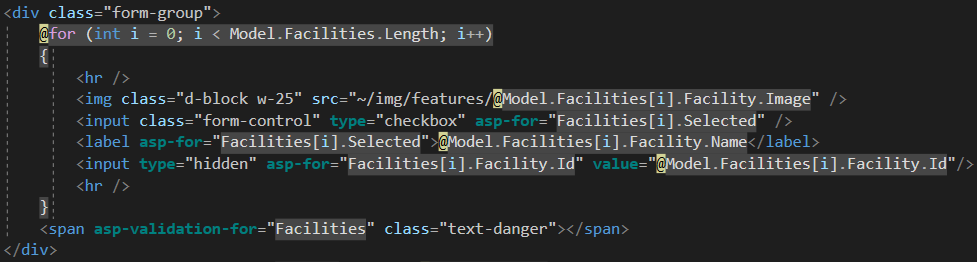


**CheckBox (multiple values)**

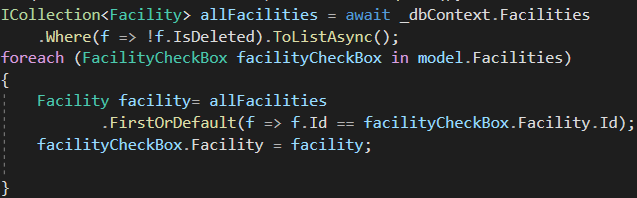
If we want to receive multiple values from one property, we will need an array.

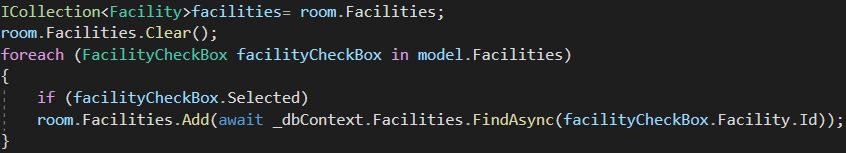
We use checkboxes! We have to have an array inside which we will have the model itself and selected boolean. With selected boolean we will indicate wheter it is checked or not 🡪





Once we submit the form, The Facility model will only have an id so we are going to need to request our database and take the facilities from the database 🡪

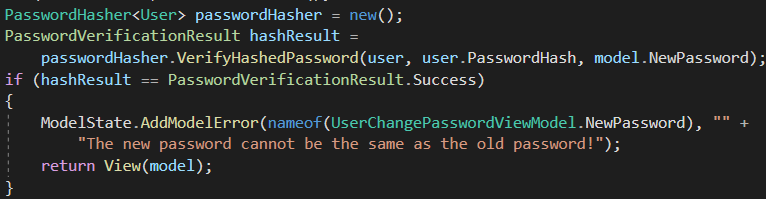




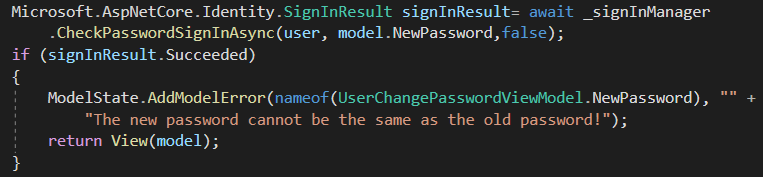
**Validation for Change Password (new and old passwords must be different)**

To validate the new password so that it is not equal to the old one we can first create a new table for old passwords as every time when we change a password we can check if it eqauls to one of the passwords in the database.

The second method is to hash our new password and take our old hashed password and compare these two hashed passwords with the PasswordHasher.VerifyHashedPassword. It will return a PasswordVerificationResult so we store that in a variable and then we check if it was successful. If it is then we will ad a model error. 🡪



The third method is to sign in with the new password and if it signs in then it means that the old password is equal to the new one 🡪



The third parameter is the lockoutOnFaulure it is set to false.