**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.

**Using statement**

Using statement is used to create a scope for a resource and once the scope ends it is going to dispose of that object (database and etc.)

If the type implements IDisposable, it automatically disposes that type.

public class SomeDisposableType : IDisposable

{

...implmentation details...

}

If it inherits from IDisposable then it will have Dispose() method to clean up

SomeDisposableType t = new SomeDisposableType();

try {

OperateOnType(t);

}

finally {

if (t != null) {

((IDisposable)t).Dispose();

}

}

This is the same as🡪

using (SomeDisposableType u = new SomeDisposableType()) {

OperateOnType(u);

}

Using calls Dispose() after the using-block is left, even if the code throws an exception. So you usually use using for classes that require cleaning up after them. In our example above(database) it will close our database and SqlDataReader as well.

**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");

}

**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 our request and our code. Authorization process is implemented in the middleware.

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**

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.

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; }

}

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 🡪

private 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.

**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 in 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.

**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 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🡪

