**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);