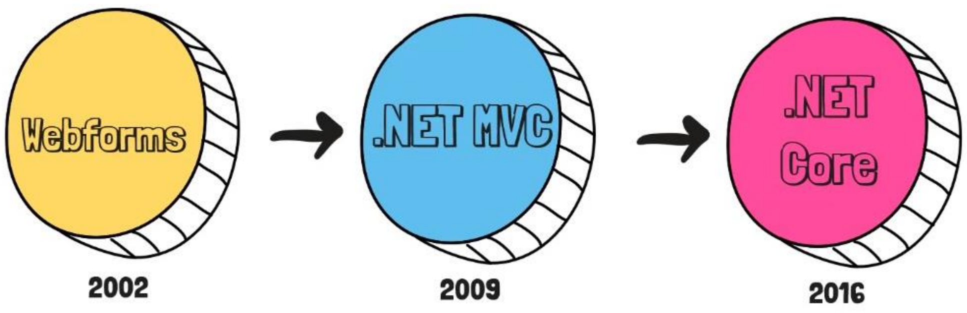
# Introduction to .Net Core

## Road Map

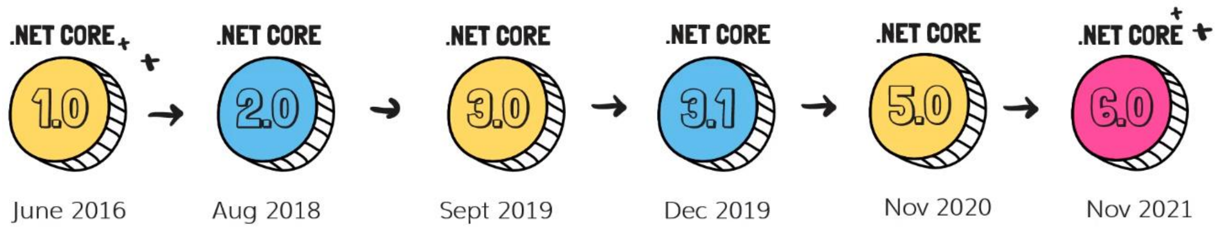


**In *2002*, Microsoft introduced web forms, which was revolution at that time.**

- Web form has its drawbacks and there was a need to overcome all of them.
- Because of that Dot Net Team came up with a new architecture which is Dot Net MVC.

**In *2009*, Microsoft introduced .Net MVC and it also have some flaws like it was created on the top of the components of Web Forms**

**In June of *2016*, Microsoft released ASP.NET Core.**



**.Net Core is built on the top the new Dot Net Core framework**

- It is completely rewritten and it is cross-platform version, hence it is not tied with Windows.
- It is also build with cloud in mind, so it is extremely robust with that.

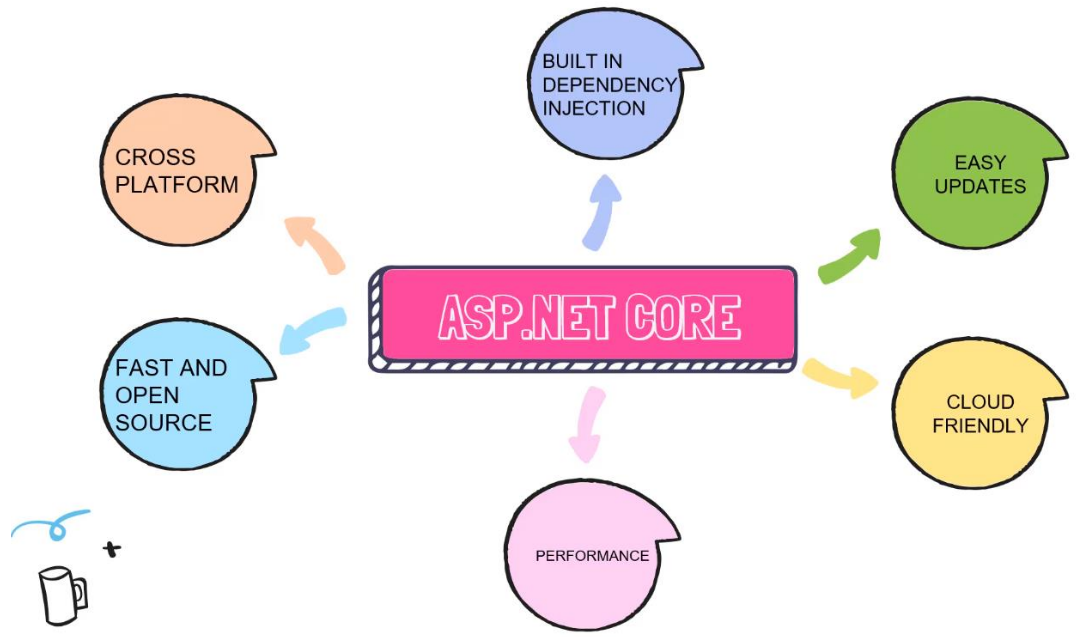
**In August of 2018, Microsoft released a report that .Net Core 2 and team has been active in releasing new version.**

**There was a big change from 2.1 to 2.2 because we had to update quite few libraries and there were few challenges.**

**But since then, Dot Net team has been releasing new versions with 3.0, 3.1, 5.0 which was released in November of 2020.**

**In November of 2021, .Net Core 6 has been released.**

## Why should we use .net core compared to the classic .net?



- **Fast and Open source**
  - If we compared that to the traditional Dot Net Applications that have been quite a few benchmarks and it is very fast when we compare that to the web forms or even Dot Net Applications.
- **Cross Platform**
  - The classic .Net was tied to IIS and Windows, but since the .Net Core code is rewritten, it has removed that dependency with .Net Core.
- **Built in Dependency Injection**
- **Easy Updates**
  - It is critical that the new updates or the new version that are released, they should be easily upgradable and that is one of the future with .NET.
  - When a new version is released, updating to that new version does not have ground breaking changes. Because of that, we can always keep up with the new versions.
- **Cloud friendly**
  - It is completely compatible with all of the cloud components.
- **Performance**
  - It exceeds all the version of the previous versions and even the new version in .Net core that are being released, they are supersede (replace) the previous version.
  - The code actually get more optimized that results into improved performance.
  - The ASP.Net Core compiler will eventually optimize the entire code whenever the code is recomposed using .Net Core Framework.

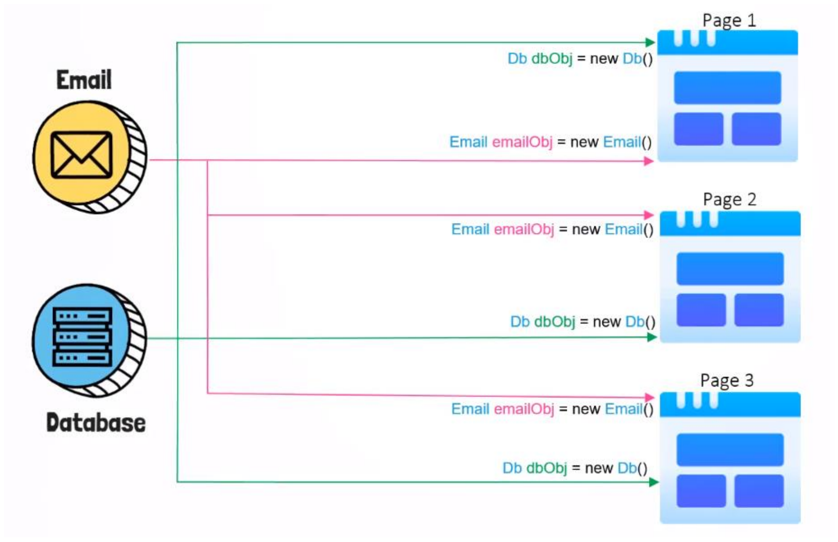
## Dependency Injection

**ASP.Net Core implements a simple patent dependency injection.**

**Container dependency injection is an integral part of the ASP.Net Core architecture.**

**.Net core inject objects of dependency classes through constructor by using the built in container.**

### Without Dependency Injection



**In a typical application, let's say we have 3 pages and we have some common functionality that we want to use across all the 3 pages like, send emails and to access Database.**

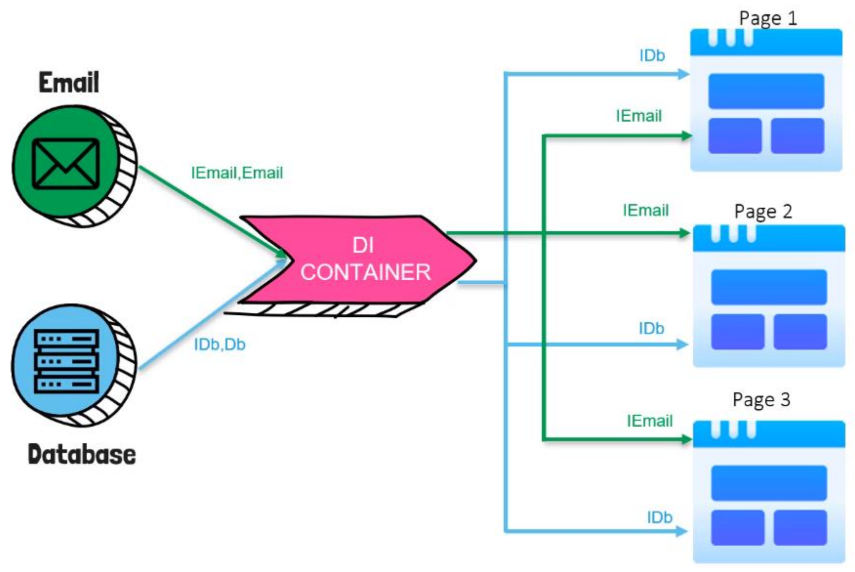
1. Now imagine that on these *three pages*, *we need to access the database first.*
   - So we will ***create the object for database classes on all the three pages***.
   - Now we would have to open the connection and do the database operations and then we would have to close the connection in all places to do the same and create object or email implementation in all the 3 pages.

**There are lot of duplicate code.**

**On the top of it if we change the implementation of how we access the database or email, based on the current configuration.**

- ✦ We might have to make that changes in all the three pages, which is a big mess which may be more pages in future.
- ✦ Another issue here is that on each page, we will deal with creating the object, managing them, as well as disposing that. (Time consuming task).

## With Dependency Injection



**Again, we would have all 3 pages and we could have email and database functionality.**

1. **Now we would have something special, which is dependency injection container that will have an *IEmail* and *IDb* interface and its implementation.**
   - So inside that container, we have the implementation of the ***IEmail*** interface and the ***IDb*** interface.
   - When any page would need access to these functionalities, it will just ask the dependency injection container to create an object of this functionality and directly give page and object to use.
   - So instead, the page will actually be using an interface and then dependency injection and pass the object when the website needs it.
   - That way, we don't have to deal with creating the object, disposing or managing that object.

**Pages will look very clean with just three interfaces alterations and implementation will be done by dependency injection container.**

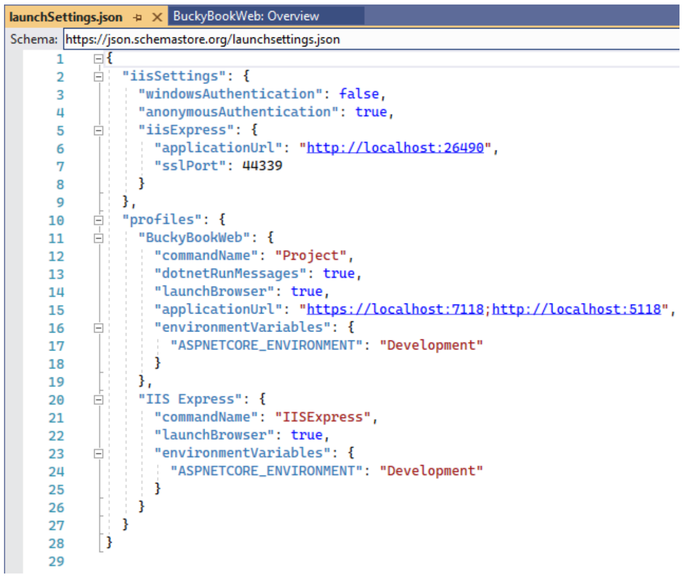
- ✦ In future, if we want to change or replace the email class, we don't have to do changes in the pages, all we have to do is just change the implementation inside the email class.
- ✦ Next time, we build the application, we are registering the changes that in the container.

**In order to use dependency injection, we can use many third party tools.**

**However, in .NET Core, we have a built-in dependency injection container and that has its own advantages.**

## File Structure

1. **Project File**
2. **Connected Services**
   - It contains the details about all the service references added to the project.
   - A new service can be added here, for example, if you want to add access to Cloud Storage of Azure Storage you can add the service here.
3. **Dependencies**
   - The **Dependencies** node contains all the references of the NuGet packages used in the project.
   - Here the **Frameworks** node contains reference two most important *dotnet core runtime* and asp.net core runtime libraries.
   - Project contains all the installed server-side NuGet packages.
4. **Properties**



launchSettings.json   BuckyBookWeb: Overview

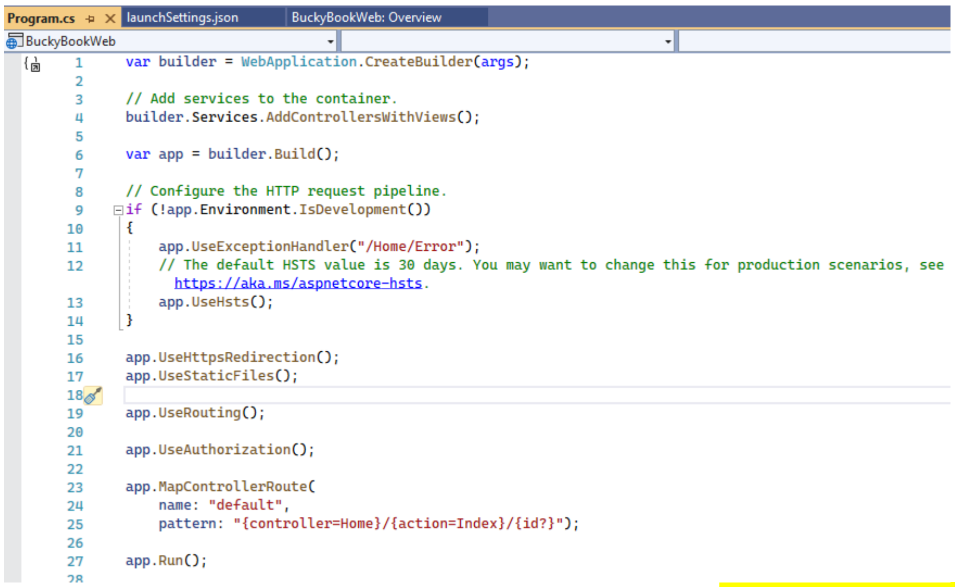
Schema: https://json.schemastore.org/launchsettings.json

```json
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:26490",
      "sslPort": 44339
    }
  },
  "profiles": {
    "BuckyBookWeb": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7118;http://localhost:5118",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

   - Properties folder contains a **launchSettings.json** file, which containing all the information required to lunch the application.
   - Configuration details about what action to perform when the application is executed and contains details like IIS settings, application URLs, authentication, SSL port details, etc.
5. **WWWroot**
   - This is the web root folder and all the static files required by the project are stored and served from here.
   - The web root folder contains a sub-folder to categorize the static file types, like all the Cascading Stylesheet files, are stored in the CSS folder, all the JavaScript files are stored in the folder and the external libraries like bootstrap, JQuery are kept in the library folder.
   - Generally, there should be separate folders for the different types of static files such as JavaScript, CSS, Images, library scripts, etc. in the wwwroot folder.
6. **appsettings.json**
   - This file contains the application settings, for example, configuration details like logging details, database connection details.

## 7. Program.cs

**This class is the entry point of the web application. It builds the host and executes the run method.**


Program.cs | launchSettings.json | BuckyBookWeb: Overview

BuckyBookWeb

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see
      https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

**Once we open the *Project.cs* file, we have a variable builder where the *webApplication.CreateBuilder(args)* is passed with build-in arguments.**

- When we run the DOTNET command, we can pass custom argument here if we want.
- With that, it will configure the application and it will create the web application builder object.

**Previously we have seen that we can use dependency injection with DOTNET Core.**

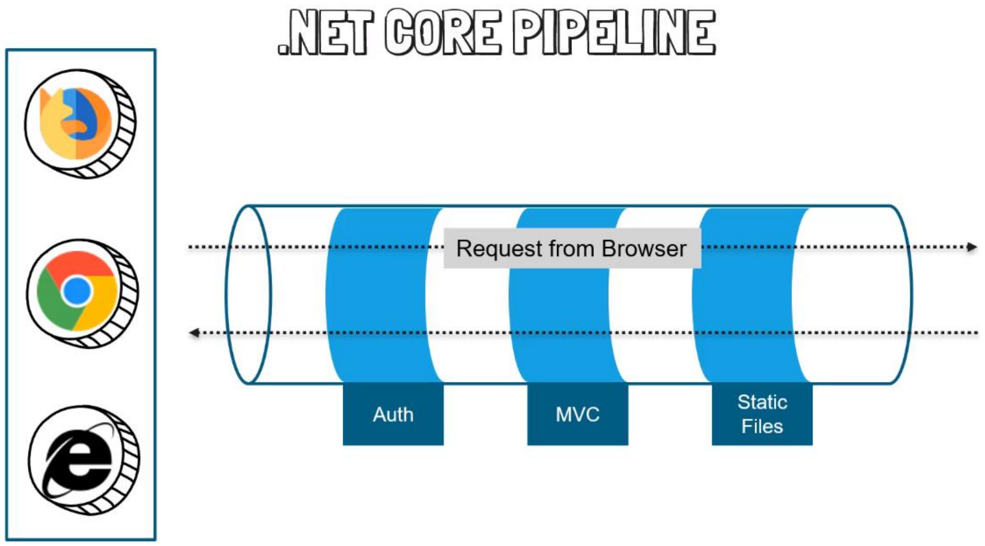
- When we want to register anything with our dependency injection container, we will doing in this file.
- Let's say we want to register our database or email or anything, we have to do between the ***builder*** and before we build the builder object.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the Dependency Injection container here

var app = builder.Build();

// configure the HTTP request Pipeline here to last line app.Run().
```

So we are adding services in Dependency Injection container then we need to configure the request pipeline.

- That pipeline will be configure.

*What is Pipeline?*



**It specify how application should respond to a web request from the browser**

**The request goes back and forth through pipeline.**

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

**We have different browser here and then we have a pipeline.**

**In Pipeline we can add items that we want.**

**Pipeline is made up of different middleware and MVC is a type of middleware that itself.**

**So if we want an application built using MVC, we have to add at middleware.**

- Other example: Authentication middleware, authorization Middleware, etc.

- ✦ What exactly happens is when our request will go through each of middleware, its gets modified by them and eventually it is passed to the next meeting there.
- ✦ If it is the last middleware in the pipeline, the response is return back to the server.

### Few Middleware in our applications:

1. **First we are checking if it is development or not in the environment.**
   - If it is then we are adding the ***UseDevelopmentExceptionPages()*** that will show us user friendly exceptions so that we can debug and solve them.
   - If it is ***not***, then we are just redirecting them to an Error Page.
2. ***UseHttpsRedirection***:
3. ***UseStaticFiles():*** Use the static files defined in ***WWWRoot*** folder.
4. `UseRouting() : Routing middleware`
5. MapControllerRoute(): we map the different patterns that we have for MVC based on this routing, it will be able to redirect a request to the corresponding controller and action.

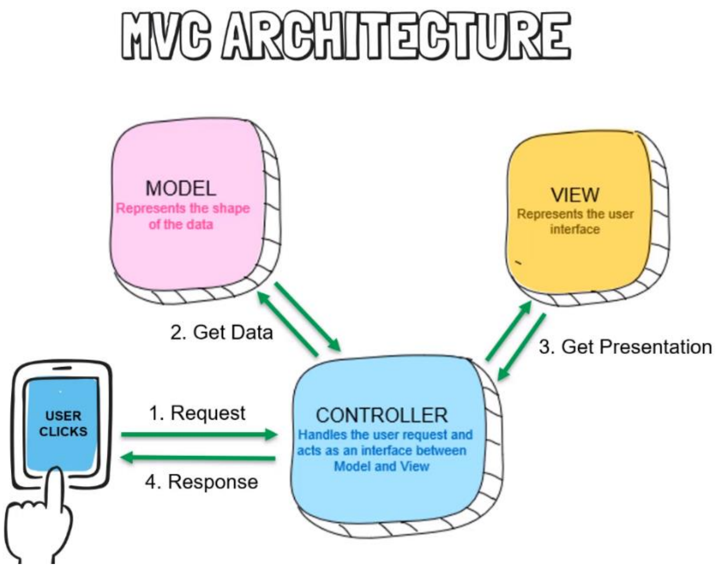
**Order of the pipeline is extremely important.**

- The way we write the middleware in the pipeline, that is exactly how the request is processed.
- So, first routing will be done then it checks for authorization and so on.

**If we place the pipeline in some different order, it will break things inside the endpoints.**

## MVC Architecture

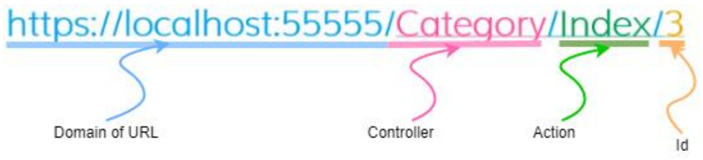
1. **Model: Represent the shape of the data.**
   - A class in C# is used to describe a model, the model component corresponds to all the data related logic that user works with.
   - Let's say, we have a table that store all the category, all the product details then that product will be model itself
   - It represent all the data in our application.
   - It can be a table that we are storing inside SQL server or it can be a model which will be a combination of multiple tables and so on.
   - This model can be either represent the data that is being transferred between views and Controllers any business related data model that could represent all the tables of the database.
2. **View: Represent the user interface.**
   - It can be part of HTML and CSS.
   - Whatever we see on the website with our eye, it's basically the view that is being displayed to us.
   - 
3. **Controller: handles the user request and act as an interface between Model and View.**
   - It act as interface between model and view to process all the data business logic and incoming request.


MVC ARCHITECTURE
MODEL
Represents the shape of the data
VIEW
Represents the user interface
2. Get Data
3. Get Presentation
USER CLICKS
1. Request
4. Response
CONTROLLER
Handles the user request and acts as an interface between Model and View

1. If a user click on a button, controller is first thing that will receive that request.
   - Controller will have lots of action methods based on those action method controller will redirect this request to one of the action method.
2. Controller will use the model, it will fetch all the data that it need to display.
3. Once the view is rendered, it will pass all of that to the controller and it will then pass a response which will be sent back and the user will finally be able to see page.

## Routing

The URL pattern for routing is considered after the domain name.



***Id*** **is optional.**

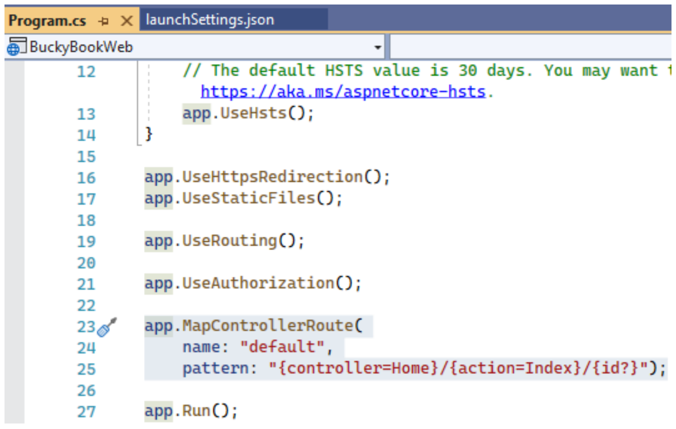
**Remember**

**If controller is not defined.**

- **Default route will be used in our application**

**If action is not defined.**

- **Index action will be used in the same controller.**



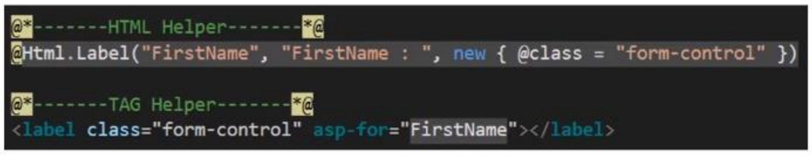
```
    // The default HSTS value is 30 days. You may want t
      https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

| URL | Controller | Action | Id |
|---|---|---|---|
| **https://localhost:55555/** | **Home** | **Index** | **Null** |
| **https://localhost:55555/Category/Index** | **Category** | **Index** | **Null** |
| **https://localhost:55555/Category/** | **Category** | **Index** | **Null** |
| **https://localhost:55555/Category/Index/3** | **Category** | **Index** | **3** |
| **https://localhost:55555/Product/Edit/3** | **Product** | **Edit** | **3** |

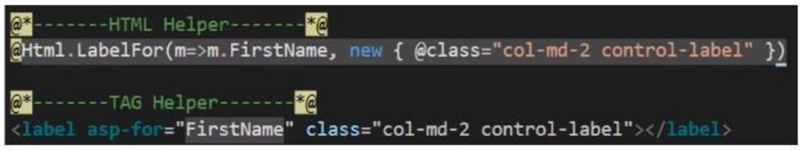
## Tag Helper

It enable server side code to participate in the creating and rendering HTML elements in Razor files.

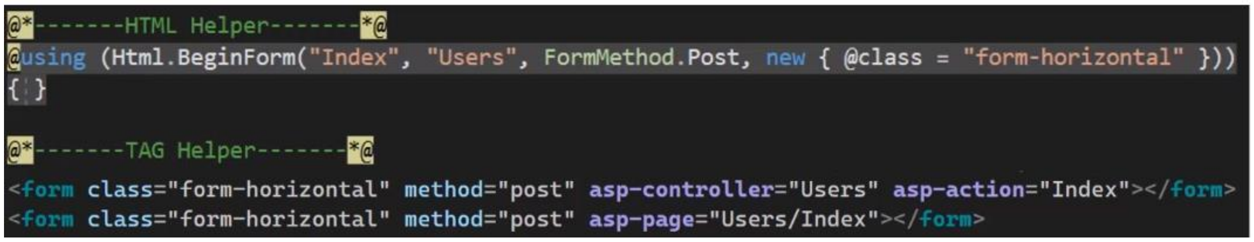
It is very focused around the html elements and much more natural to use.



```
@*-------HTML Helper-------*@
@Html.Label("FirstName", "FirstName : ", new { @class = "form-control" })

@*-------TAG Helper-------*@
<label class="form-control" asp-for="FirstName"></label>
```



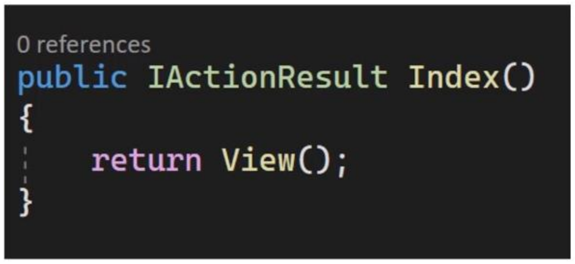
```
@*-------HTML Helper-------*@
@Html.LabelFor(m=>m.FirstName, new { @class="col-md-2 control-label" })

@*-------TAG Helper-------*@
<label asp-for="FirstName" class="col-md-2 control-label"></label>
```



```
@*-------HTML Helper-------*@
@using (Html.BeginForm("Index", "Users", FormMethod.Post, new { @class = "form-horizontal" }))
{ }

@*-------TAG Helper-------*@
<form class="form-horizontal" method="post" asp-controller="Users" asp-action="Index"></form>
<form class="form-horizontal" method="post" asp-page="Users/Index"></form>
```

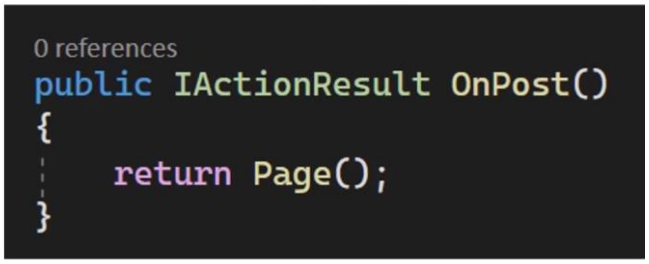
## Action Result

MVC Application



```
0 references
public IActionResult Index()
{
    return View();
}
```

Razor Page Application

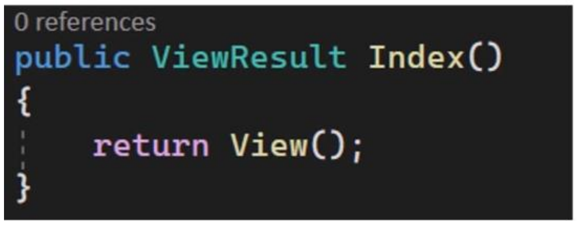


```
0 references
public IActionResult OnPost()
{
    return Page();
}
```

Action Result is a generic type that implement all type of other return types.

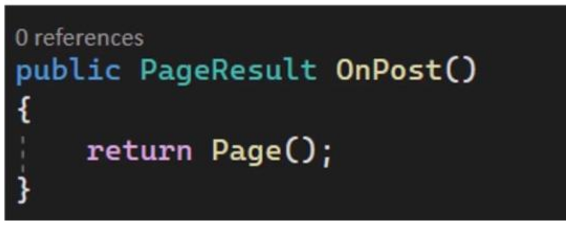
Now if we want to be explicit about the return type in the both of below cases.

MVC Application



```
0 references
public ViewResult Index()
{
    return View();
}
```

Razor Page Application



```
0 references
public PageResult OnPost()
{
    return Page();
}
```

- ***ActionResult*** is a result of action method/pages or return type of action methods/page handlers.
- ***ActionResult*** is a parent class for many of the derived classes that have associated helpers.
- **The *IActionResult*** return type is appropriate when multiple ActionResult return types are possible in an action.

| ActionResult | Helper | Description |
|---|---|---|
| **ContentResult** | Content | Takes a string and returns it with a text/plaincontent-type header by default. Overloads enable you to specify the content-type to return other formats such as text/html or application/json, for example. |
| **FileContentResult** | File | Returns a file from a byte array, stream or virtual path. |
| **NotFoundResult** | NotFound | Returns an HTTP 404 (Not Found) status code indicating that the requested resource could not be found |
| **PageResult** | Page | Will process and return the result of the current page. |
| **PartialResult** | Partial2 | Returns a Partial Page. L |
| **RedirectToPageResult** | RedirectToPage<br>RedirectToPagePermanent<br>RedirectToPagePreserveMethod<br>RedirectToPagePreserveMethodPermanent | Redirects the user to the specified page. |
| | | |
| **ViewComponentResult** | | Returns the result of executing a ViewComponent. |

## 22. Create Category Model

Model will basically resemble a table in database.

It is not always the case, but whatever tables we have in our database, we will need a corresponding model for the code First Migration.

1. Create a class with name “***Category.cs***” in Models folder.

```
public class Category
    {
        [Key]
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        public int DisplayOrder { get; set; }
        public DateTime CreatedDateTime { get; set; } = DateTime.Now;

    }
```

## 23. Data Annotation

## 24. Connection String for Database

1. Add the connection string AppSetting.json file.

```
{
  "ConnectionStrings": {

"DefaultConnection":"server:localhost;Database=BulkyDb;TrustedConnection:True;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

We can create different ***appSetting*** for different Environment like if we have development, staging, preview production, we can create all those ***appSettings*** and we can configure to use that appSetting.

Based on environment name, we can use different app settings or different databases and so on.

## 25. Add ApplicationDbContext

1. Create a Folder and name it "***Data***".
2. Create a class and name it "***ApplicationDbContext***".
3. Add a package "***Microsoft.EntityFrameworkCore***" using NuGet Package manager or console.
4. Inherit the class with "DbContext" class and add namespace "*using Microsoft.EntityFrameworkCore;*"
5. Create a constructor and
   - In constructor we will receive some options and those options we just have to pass to the base class, which is DbContext.
6. Whatever models that we have to create inside the database, we will have to create a Dbset inside the ApplicationDbContext.

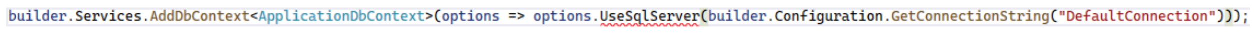
```
public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> 
options): base(options)
        {
        }

        // create db sets which we want to create
        public DbSet<Category> Categories { get; set; } = default!;
    }
```

## 26. Setup Program.cs to use DbContext

We just need to tell our application that it has to use the DbContext,and then it hase to use a SQL Server using connection string that we defined.

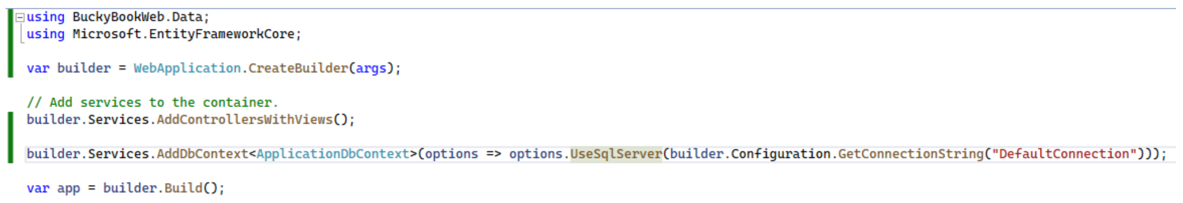
We configure the services that are application would use.

**We add a new service.**



```
builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

- UseSqlServer is not defined because we need to add a package "***Microsoft.EntityFrameworkCore.SqlServer***".
- Always ensure to install same version of above package to ***Microsoft.EntityFrameworkCore*.**



```
using BuckyBookWeb.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();
```

## 27. Create Database

We create database using EntityFramework Code first approach.

1. Install a package "***Microsoft.EntityFrameworkCore.Tools***" using Package manager.
2. Open the package manager console.
3. 
4. Add migration using this command: ***add-migration AddCategoryToDb***
   - Migration is basically keeping the track of all the changes that are needed.
   - Now Migration folder has been created and it has 2 files,
5. Push the migration to database using this command: ***update-database***
   - It will connect to database server and check is there any database in server we want to connect.
   - There won't be anything like that, so it will create that database first and then migration here convert them to SQL and execute them on our database.

**.Net Core CLI**

**// add migration**

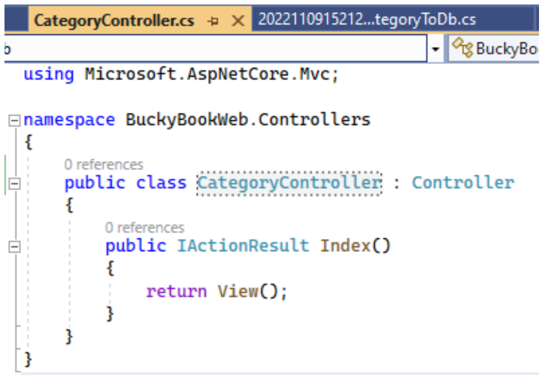
**dotnet ef migrations add InitialCreate**

**// push the migration to database**

**dotnet ef database update**

## 28. Create Category Controller

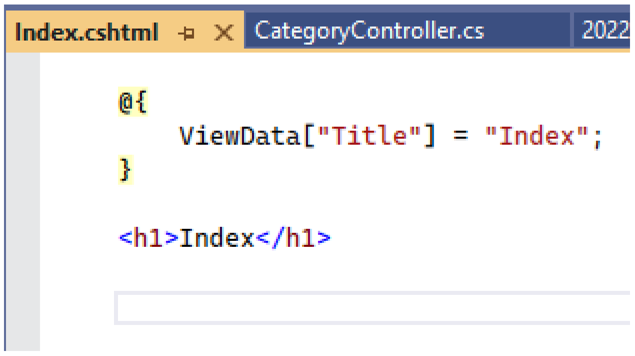
1. Create a controller by Right click on Controller folder > select Add > select controller.
   - Name the controller class with ***CategoryController*.**



CategoryController.cs    2022110915212...tegoryToDb.cs

```
using Microsoft.AspNetCore.Mvc;

namespace BuckyBookWeb.Controllers
{
    0 references
    public class CategoryController : Controller
    {
        0 references
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

2. Create an index view from Index action by right click on Index *action* method > select **add view** > select **Razor View**.



Index.cshtml    CategoryController.cs    2022

```
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>
```

## 29. Retrieve all categories

We need to retrieve all categories list from data base in Index action method.

1. Create an object of ***ApplicationDbContext*** using Dependency injection.
   - Create a private readonly field of ***ApplicationDbContext*** named ***_db***.
   - Create class constructor and in parameter we pass the whatever is registered inside the Dependency injection container in service which is implementation of ApplicationDbContext object.
   - Using _db, we can retrieve categories list from database.

```
using BuckyBookWeb.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore.Query.Internal;

namespace BuckyBookWeb.Controllers
{
    public class CategoryController : Controller
    {
        private readonly ApplicationDbContext _db;

       // inject ApplicationDbContext implementation DJ container from service
        public CategoryController(ApplicationDbContext dbContext)
        {
            _db = dbContext;
        }

        public IActionResult Index()
        {
            IEnumerable<Category> list =_db.Categories.ToList();
            return View(list);
        }
    }
}
```

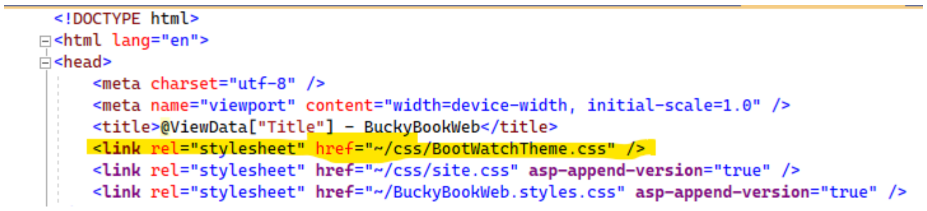
## 30. Display all categories

1. In view, we need to capture the model that we are passing from the controller.
   - Declare at top @model IEnumerable<Catergory>
   - We declare the same thing we pass from controller.
2. Using razor syntax, we iterate using foreach loop.

```
@model IEnumerable<Category>
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>
<table class="table-bordered table-strapped" style="width:100%">
    <thead>
        <tr>
            <th>Category Name</th>

            <th>Display Order</th>
        </tr>
    </thead>
    <tbody>
        @foreach(var obj in Model){
            <tr>
                <td width="50%">
                    @obj.Name
                </td>
                <td width="30%">
                    @obj.DisplayOrder
                </td>
            </tr>
        }
    </tbody>
</table>
```

## 31. Add BootWatch Theme

1. Go to Bootwatch.com and select the theme.
2. Download the Bootstrap.css file and copy code from file.
   - Create a CSS file in WWWroot > CSS with name “bootWatchTheme”.
3. Go to View > Shared > _Layout.
   - Replace the link of old bootstrap theme CSS to new.



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - BuckyBookWeb</title>
    <link rel="stylesheet" href="~/css/BootWatchTheme.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    <link rel="stylesheet" href="~/BuckyBookWeb.styles.css" asp-append-version="true" />
```

4. Install Bootstrap 5 using NuGet Package Manager.

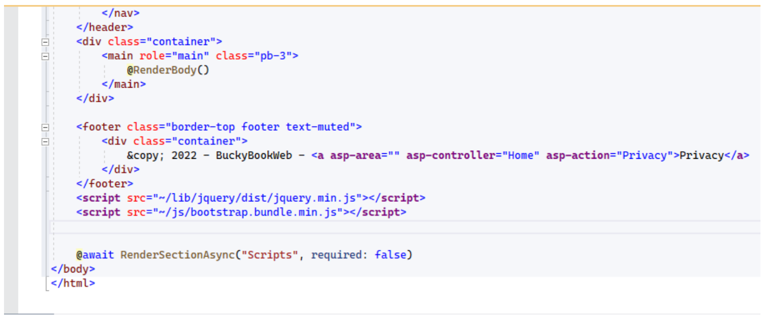
**Problem for referencing Bootstrap when installed via nuget**

The client-side library NuGet packages are for ASP.NET MVC, *not* ASP.NET Core. Core uses an entirely different methodology for managing client-side libraries.

In Core, you need to use either LibMan (the default client-side library manager provided by Microsoft) or npm (the Node.js package manager). While npm requires a bit more effort, as you'll need to combine it with something like webpack or gulp, it is the preferred approach and also more complete and extensible. LibMan, while easy, is not complete. It relies entirely on cdnjs.cloudflare.com, so if the library doesn't exist there, you're out of luck. While there's a very large amount of libraries covered, it is not exhaustive, and you'll likely run into one thing or another that you can't obtain. At that point, you'll have to fallback to npm, anyways, so you might as well just use it for everything.

   - ***Manually install the bootstrap files in the wwwroot folders and link it in _layout file.***

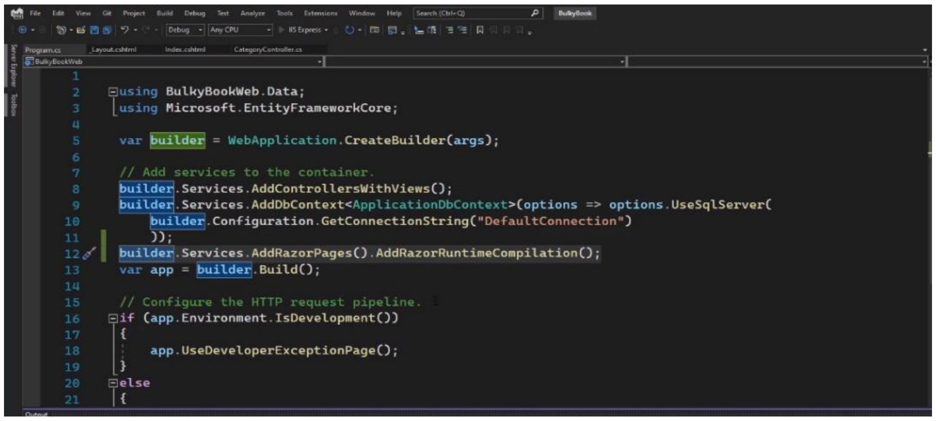
5. In _Layout file, replace the JS file with Bootstrap.bundle.min.js.



```
            </nav>
        </header>
        <div class="container">
            <main role="main" class="pb-3">
                @RenderBody()
            </main>
        </div>

        <footer class="border-top footer text-muted">
            <div class="container">
                &copy; 2022 - BuckyBookWeb - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
            </div>
        </footer>
        <script src="~/lib/jquery/dist/jquery.min.js"></script>
        <script src="~/js/bootstrap.bundle.min.js"></script>


        @await RenderSectionAsync("Scripts", required: false)
    </body>
</html>
```

6. To get Runtime Compliation of Razor Pages we need to install a Package “***Microsoft.AspNetCore.Mvc.Razor.RuntimeCom***”.
   -

- Add service in ***Program.cs***



```
using BulkyBookWeb.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(
    builder.Configuration.GetConnectionString("DefaultConnection")
    ));
builder.Services.AddRazorPages().AddRazorRuntimeCompilation();
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
```

## 32. Add Bootstrap Icon

1. Go to ***https://icons.getbootstrap.com/***
2. Download zip and save in wwwroot folder and link it in _Layout file.
3. Add Heading and a button in ***index.cshtml*** of Category controller.

```
<div class="col-6 text-end">
                <a asp-controller="Category" asp-action="Create" class="btn btn-primary" >
                                Create a New Category
                </a>
</div>
```

4. Add the + icon in button.

```
<a asp-controller="Category" asp-action="Create" class="btn btn-primary" >
        <i class="bi bi-plus-circle"></i> &nbsp; Create a New Category
</a>
```

## 33. Create category View

1. Create an action method for create category.
   - The model we are writing in view is not always the model we passed from the controller.
   - But if we are passing a model from the controller, it must match what we have inside the model in the view.
   - If we are not passing anything in the controller like in ***create***, then we can find with a model based on the data that we are collecting on the page.

```
@model Category
@{
    ViewData["Title"] = "Create";
}


<form method="post">
    <div class="border p-3 mt-4">
        <div class="row pb-2">
            <h2 class="text-primary">Create Cateogry</h2>
            <hr />
        </div>
        <div class="mb-3">
            <label asp-for="Name"></label>
            <input asp-for="Name" class="form-control"/>
        </div>
        <div class="mb-3">
            <label asp-for="DisplayOrder"></label>
            <input asp-for="DisplayOrder" class="form-control" />
        </div>
        <button type="submit" class="btn btn-primary" style="width:150px"> Create</button>
        <a asp-controller="Category" asp-action="Index" class="btn btn-secondary" style="width:150px">Back to 
List</a>
     </div>
</form>
```

## 34. Create Category Post Action Method

```
//POST
[ValidateAntiForgeryToken]
public IActionResult Create(Category obj)
{
  // add the entry to db
  _db.Categories.Add(obj);

  // push the changes to db
  _db.SaveChanges();

  return RedirectToAction("Index");
}
```

## 35. Server Side Validation

**When we receive a model in controller, we can check whether the model is valid or not.**

**Valid model defined by the data annotation in the model.**

- Example: Name have a required property in model.

**We have *ModelState* that determine if the model is valid or not.**

```
//POST
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Category obj)
{
    if (ModelState.IsValid)
    {
        // add the entry to db
        _db.Categories.Add(obj);

        // push the changes to db
        _db.SaveChanges();

        return RedirectToAction("Index");
    }
    return View(obj);
}
```

**In view, we have a tag helper for validation 'asp-validation-for'.**

- This helps us to display the error on the view from the server-side validation.
- We did not have to write any JavaScript or any other complex code.

```
<div class="mb-3">
            <label asp-for="Name"></label>
            <input asp-for="Name" class="form-control"/>
            <span asp-validation-for="Name" class="text-danger"></span>
</div>
```

## 36. Custom Server-Side Validation

**What if we want to display a summary at the top with all the error messages**.

- We can use helper tag ‘asp-validation-summary’ and provide value ‘All’ to display all validation at the top of the page.

```
<div asp-validation-summary="All"></div>
```

**We want to make sure that we don’t add any category which has the same name and display order**

- .Before we confirm if the model state is valid, we can check if same category name and display order is same.

```
//POST
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Category obj)
{
    if(obj.Name == obj.DisplayOrder.ToString())
    {
        ModelState.AddModelError("CustomError", "The display order cannot exactly match the name");
    }

    // check if model is valid
    if (ModelState.IsValid)
    {
        // add the entry to db
        _db.Categories.Add(obj);

        // push the changes to db
        _db.SaveChanges();

        return RedirectToAction("Index");
    }
    return View(obj);
}
```

All the validations that we have so far are done on the server side.

Every time we submit the button, the request hit the server even if we don’t enter any value in any field.

We need to do some client side validation to prevent reload of page for every validations.

## 37. Add Client-Side validation

If we want to do the basic model validation, we have created a partial view in shared folder with name ***_ValidationScriptsPartial.***

- Added two scripts in it.

```
<script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
<script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js"></script>
```

- If we want to do client side validations, we just have to include this partial view in our view.

```
@section Scripts
{
    <partial name="_ValidationScriptPartial"></partial>
}
```

Now we get the validations, but we are not going to server this time.

On the client side, we can only validate the following rules we defined in the Model using data annotation.

When the name and display order are the same, then the client-side validation are valid but server-side validation is invalid.

## 39. Display Name and Range Validation

In client-side validation, the field name is same as model property name and default message is used for error message.

To display customized Field name, we use data annotation ***[DisplayName()]*** property in the model class.

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace BuckyBookWeb.Models
{
    public class Category
    {
        [Key]
        public int Id { get; set; }

        [DisplayName("Display Order")]
        [Required]
        public string Name { get; set; } = string.Empty;

        [Range(1,100, ErrorMessage ="Display order must be between 1 to 100!")]
        public int DisplayOrder { get; set; }
        public DateTime CreatedDateTime { get; set; } = DateTime.Now;

    }
}
```

## 40. Edit Category (GET)

1. Create an action method with name ‘Edit’ and request type of GET.

```
[HttpGet]
public IActionResult Edit(int id)
{
    // check if id is valid or not
    if (id == null || id == 0)
        return NotFound();


    // fetch the category details
    var CategoryFromDb = _db.Categories.Find(id);
    //var CategoryFromDbSingle = _db.Categories.SingleOrDefault(u => u.Id == id);
    //var CategoryFromDbFirst = _db.Categories.FirstOrDefault(u => u.Id == id);

    // check if fetch data is null
    if (CategoryFromDb == null)
        return NotFound();


    // if fetch data is not null then return data to view
    return View(CategoryFromDb);
}
```

2. Create an edit view to bind the data from action method.

```
@model Category
@{
    ViewData["Title"] = "Create";
}


<form method="post">
    <div class="border p-3 mt-4">
        <div class="row pb-2">
            <h2 class="text-primary">Edit Cateogry</h2>
            <hr />
        </div>
        <!-- show validation summary -->
        <div asp-validation-summary="All"></div>
        <div class="mb-3">
            <label asp-for="Name"></label>
            <input asp-for="Name" class="form-control"/>
            <!-- show validation error -->
            <span asp-validation-for="Name" class="text-danger"></span>
        </div>
        <div class="mb-3">
            <label asp-for="DisplayOrder"></label>
            <input asp-for="DisplayOrder" class="form-control" />
            <!-- show validation error -->
            <span asp-validation-for="DisplayOrder" class="text-danger"></span>
        </div>
        <button type="submit" class="btn btn-primary" style="width:150px"> Update</button>
        <a asp-controller="Category" asp-action="Index" class="btn btn-secondary"
style="width:150px">Back to List</a>
     </div>
</form>

@section Scripts
{
    <partial name="_ValidationScriptPartial"></partial>
}
```

If we don’t provide the submit action method in form, it would by default submit to the same action method. We can also explicitly mention the action method in the form tag.

3. Add an edit link button in the Index view to redirect to the view page of Category.

```
<td width="30%">
          <div class="w-75 btn-group" role="group">
                    <a asp-controller="Category" asp-action="Edit" asp-route-id="@obj.Id" class="btn btn-primary mx-2">
                              <i class="bi bi-pencil-square"></i>
                              Edit
                    </a>
          </div>
</td>
```

## 41. Edit Category (POST)

1. Create an action method with name ‘Edit’ and request type of POST.

```
[HttpPost]
 [ValidateAntiForgeryToken]
public IActionResult Edit(Category obj)
{

        if (obj.Name == obj.DisplayOrder.ToString())
        {
                ModelState.AddModelError("CustomError", "The display order cannot exactly match the name");
        }

                        // check if model is valid
        if (ModelState.IsValid)
        {
                // update the record based on primary key to db
                _db.Categories.Update(obj);

                // push the changes to db
                _db.SaveChanges();

                return RedirectToAction("Index");
        }
        return View(obj);
}
```

***Update method*** would automatically update all of the properties and it will look at the passed object and it will find its primary key and check all the other properties where the values have changed and it will update all properties.

## 42. Delete Category (POST)

### 1. Create an action method with name ‘Delete and request type of POST.

```
[HttpGet]
public IActionResult Delete(int? id)
{
          // check if id is valid or not
          if (id == null || id == 0)
                    return NotFound();


          // fetch the category details
          var CategoryFromDb = _db.Categories.Find(id);
          //var CategoryFromDbSingle = _db.Categories.SingleOrDefault(u => u.Id == id);
          //var CategoryFromDbFirst = _db.Categories.FirstOrDefault(u => u.Id == id);

          // check if fetch data is null
          if (CategoryFromDb == null)
                    return NotFound();


          // if fetch data is not null then return data to view
          return View(CategoryFromDb);
}

//Now we can use Delete as Action method for post request.
 [HttpPost,ActionName("Delete")]
 [ValidateAntiForgeryToken]
public IActionResult DeletePost(int? id)
{

          var obj = _db.Categories.Find(id);

          if(obj == null)
          {
              return NotFound();
          }


          // remove the record based on primary key to db
          _db.Categories.Remove(obj);

          // push the changes to db
          _db.SaveChanges();

          return RedirectToAction("Index");
}
```

### 2. Create a view to show data we want to delete.

```
@model Category
@{
    ViewData["Title"] = "Delete";
}


<form method="post" asp-action="Delete">
    <!--Hidden Id for delete a category-->
    <input asp-for="Id" hidden/>
    <div class="border p-3 mt-4">
        <div class="row pb-2">
            <h2 class="text-primary">Delete Cateogry</h2>
            <hr />
        </div>
        <div class="mb-3">
            <label asp-for="Name"></label>
            <input asp-for="Name" disabled class="form-control"/>
        </div>
        <div class="mb-3">
            <label asp-for="DisplayOrder"></label>
            <input asp-for="DisplayOrder" disabled class="form-control" />
        </div>
        <button type="submit" class="btn btn-primary" style="width:150px"> Delete</button>
        <a asp-controller="Category" asp-action="Index" class="btn btn-secondary" style="width:150px">Back to List</a>
     </div>
</form>

@section Scripts
{
    <partial name="_ValidationScriptsPartial"></partial>
}
```

## 3. Added the delete button in the index page.

```
<a asp-controller="Category" asp-action="Delete" asp-route-id="@obj.Id" class="btn btn-danger mx-2">
        <i class="bi bi-trash-fill"></i>
        Delete
a>
```

## 43. Tempdata

Now we want to display some alerts when a user delete or edit a category.

***Tempdata stores data for only one request.***

- If we refresh the same page, that would be gone.
- That is perfect for displaying alerts of successful or failure notification.

**When we create or edit or delete anything, we will add something to a temp data of success.**

- So once the create is successful, before redirect, we use Tempdata to store success message.

```
// for success alert
TempData["Success"] = "Category created successfully";
```

**It is great to display notification on some of the action that are performed.**

**It can be used throughout the application and not just one page.**

***Why not we are checking this temp data success and displaying that at a global level?***

- ***Partial view*** *would be a perfect choice because it is possible that this code will increase drastically when we do something fancy or notification.*

## 44. Partial View

Alert code is applicable on almost all the pages because if in future we added more pages, we don't want to copy and paste the same code in the other places.

1. **Create a partial view in the Shared folder with name "_notification".**

- Right click on View folder > Add View > Razor View > select 'Create as a partial view'
- Copy the Tempdata check code into the partial view.

```
@if (TempData["Success"] != null)
{
        <h2>@TempData["Success"]</h2>
}
@if (TempData["Failure"] != null)
{
        <h2>@TempData["Failure"]</h2>
}
```

2. **In index, we call the partial view.**

```
<!--Call the partial view-->
<partial name="_Notification" />
```

## 45. Toastr Alerts

***ToastrJS* is a JavaScript library for Gnome / Growl type non-blocking notifications.**

**jQuery is required.**

**The goal is to create a simple core library that can be customized and extended.**

1. **Copy the minified JS and CSS files and add it to project.**
2. **Link the CSS in _Layout view (master page).**
3. **Link the JS of jQuery and toastr in partial view '_Notification'**

```
@if (TempData["Success"] != null)
{
        <script src="~/lib/jquery/dist/jquery.min.js"></script>
        <script src="~/js/Toastr.js"></script>
        <script type="text/javascript">
                toastr.success('@TempData["Success"]')
        </script>

}
@if (TempData["error"] != null)
{
        <script src="~/lib/jquery/dist/jquery.min.js"></script>
        <script src="~/js/Toastr.js"></script>
        <script type="text/javascript">
                toastr.error('@TempData["error"]')
        </script>
}
```

4. **Call the partial view in master page (_Layout.chtml)**

```
<div class="container">
        <main role="main" class="pb-3">
            <!-- add parrial view to show alert on all pages -->
            <partial name="_Notification"/>
            @RenderBody()
        </main>
</div>
```

We add this partial view in the master page to prevent duplication of code in other pages,

It is a common feature all pages should have.

## 46.Scaffold CRUD

**Scaffolding is used to define the code-generation framework used in web applications.**

**It uses T4 templates to generate basic controllers and views for the models.**

**It generates instances for the mapped domain model and code for all CRUD operations.**

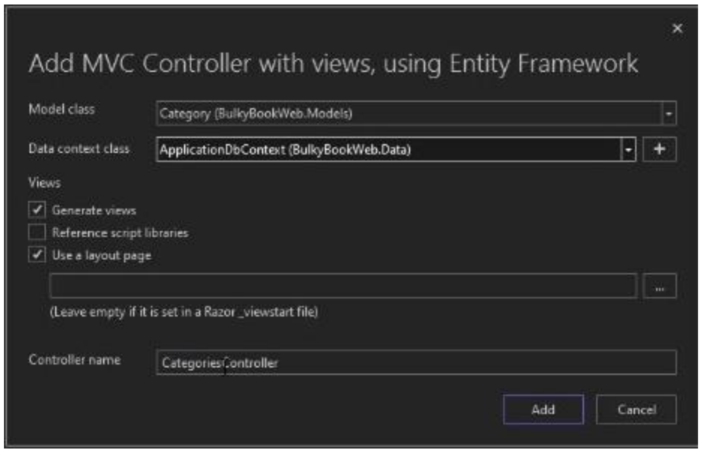
**It also reduces the amount of time for developing a standard data operation in the application.**

**Basically, it is an automated code generation framework, it generates code for CRUD operations based on the provided domain model classes and DB connections.**

**You can add scaffolding to your project when you want to add code that interacts with the data model in the shortest amount of time.**

### Creating a new controller using EF framework (Scaffold)

1. Right click on controller folder > Add > Controller > MVC controller with view, using EntityFramework.



Add MVC Controller with views, using Entity Framework

Model class: Category (BulkyBookWeb.Models)

Data context class: ApplicationDbContext (BulkyBookWeb.Data)

Views

- [✓] Generate views
- [ ] Reference script libraries
- [✓] Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

Controller name: CategoriesController

- Select Model on which we want to create CRUD operation.
- Select data Context (DBContext) which will be used to access the database.
- Add controller name for this controller.

**It will create a new controller with create, edit, delete, details and index.**

- All the correct functionalities with view are ready for us to use.
- But we need to spend time on customizing those based on our requirement.

**To remove the scaffolded controller, just remove the controller and Views from their respective folders.**

# N-Tier Architecture

## 79. Creating more Project

Application will be growing drastically and having everything in one project is not a good idea.

- It makes sense to divide the project into different project based on tier architecture.

1. **Add new project of type class library, with name '*BuckyBookDataAccess*', *BuckyBookModels*', '*BuckyBookUtility*'.**
    - In this project ***BuckyBookDataAccess*'**, everything related to data will be stored.
    - ***BuckyBookModels*** responsible for all models in our project.
    - ***BuckyBookUtility*** project is responsible to have all the generic utility for out project like email options, session extensions, static details, etc.

## 80. N-Tier Architecture

Now we have different projects (libraries), lets move the corresponding folders inside that individual project.

1. Move data folder inside the ***BuckyBookDataAccess*** and remove from the main project.
2. Move Models folder inside the ***BuckyBookModels*** project and remove from the main project.
3. Create a static class in utility with name '***SD'***
    - We don't have to create an object of this class and it will hold all the static details.
4. Change the namespace of the *Models* to "`BuckyBookWeb.Models`".
5. Change the namespace of the *DataAccess* to "`BuckyBookWeb.DataAccess`".
6. Change the namespace of the *Utility* to "`BuckyBookWeb.Utility`".

**In DataAccess project, we need to add EntityFramwork package to prevent compile time error for migration.**

1. Install ***Microsoft.EntityFrameworkCore*** (Version: 7.0 stable) using the NuGet package manager.
2. Move the **migration** folder to DataAccess project and remove this folder from main project.
3. Install ***Microsoft.EntityFrameworkCore.Tools*** (Version: 7.0 stable) using the NuGet package manager.
4. Install ***Microsoft.EntityFrameworkCore.Design*** (Version: 7.0 stable) using the NuGet package manager.

## 81. How to handle corrupted migrations

If migration get corrupted, while the project is new and for fresh application.

1. Delete migration folder from the project.
2. Drop the project database from SQL Server.
3. Add new migration using package manager console.
    - Add-migration addCategory

**This is not an approach for production but only for learning.**

**This approach is not for applied migration on production.**

# Repository Pattern

## 83. Introduction

**It is introduced as a part of domain driven design in 2004.**

**It provides abstraction of data so that our application can work in a simple abstraction that has an interface approximating of the collection.**

**Adding, removing updating and selecting items form this collection id done through a series of straightforward method without the need to deal with database concerns like connections, commands, cursor or readers.**

**Using this pattern can help achieve loose coupling and keep the domain objects, persistence.**

**Benefits of Repository pattern.**

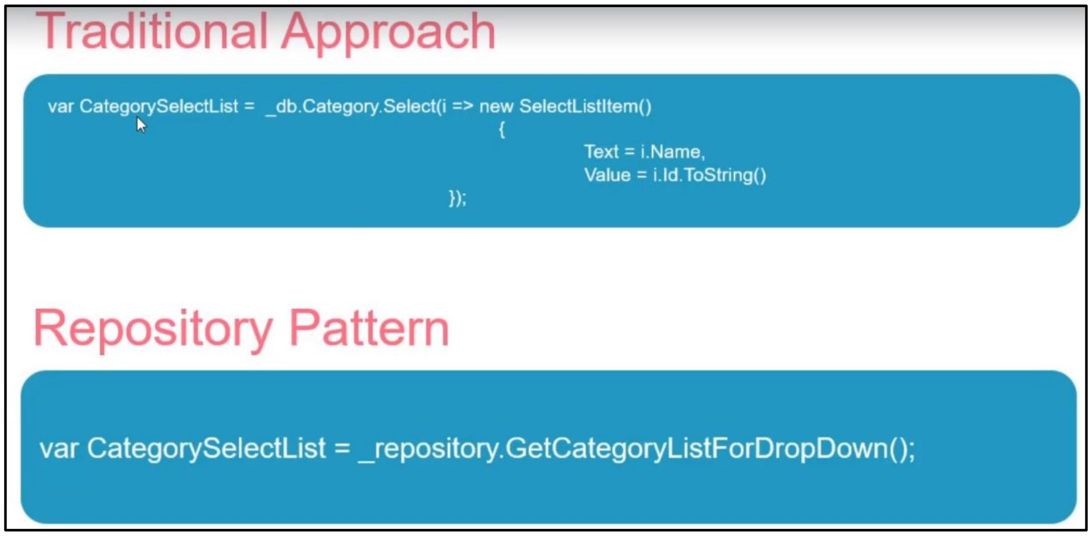
1. Minimize the duplication logic.
2. It ***decouples*** our application from persistence frameworks.
    - In the future, if we want to switch to a different persistence framework, so we can do with minimum impact on the rest of the application.
    - If we want to have freedom to explore the different persistence framework with minimal impact on our application.

***Coupling*** describes the degree of dependency between one entities to another entity

- When ClassA depends heavily on ClassB, the chances of ClassA being affected when ClassB is changed are high. This is strong coupling.
- However if ClassA depends lightly on ClassB, than the chances of ClassA being affected in any way by a change in the code of ClassB, are low. T***his is loose coupling, or a 'decoupled' relationship.***

Loose coupling is good because we don't want the components of our system to heavily depend on each other. We want to keep our system modular, ***where we can safely change one part without affecting the other.***

When two parts are loosely coupled, they are more independent of each other and are less likely to break when the other changes.



## Traditional Approach

```
var CategorySelectList =  _db.Category.Select(i => new SelectListItem()
                                                {
                                                        Text = i.Name,
                                                        Value = i.Id.ToString()
                                            });
```

## Repository Pattern

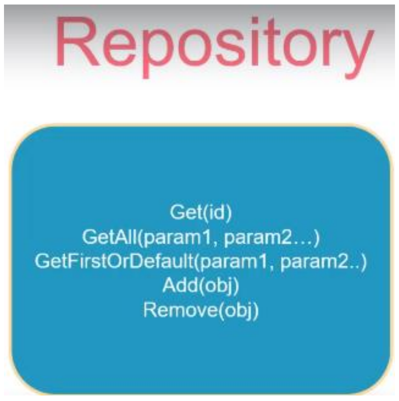
```
var CategorySelectList = _repository.GetCategoryListForDropDown();
```

***Let's understand with the help of example, let's say we have a requirement in the website to display a list of categories in a dropdown in multiple places in our website.***

- We will end up writing something like traditional method in the five places.
- It would be better if we just had a function that would retrieve us the same result every five times.
- Because of this, that will be much less duplicate code and the actual logic would be resides in the one location. If we want to change it, we can have to update in only one location.

## 84. Repository

***Repository*** should act like a collection of objects in the memory, so we have methods to add or remove an object or get an object by it or get all of the object based on certain parameters.



**We don't have an update method in the repository because many times update logic is not common for all the objects or entities.**

- Hence, we should always implement that separately and not inside our common repository.
- One common mistake by many developers is to implement an update method in the repository. But ***it should not be included.***
    - As the project grows in the complexity and increase the different update logic.
    - It will not work and we will end up overriding it almost all of the places.

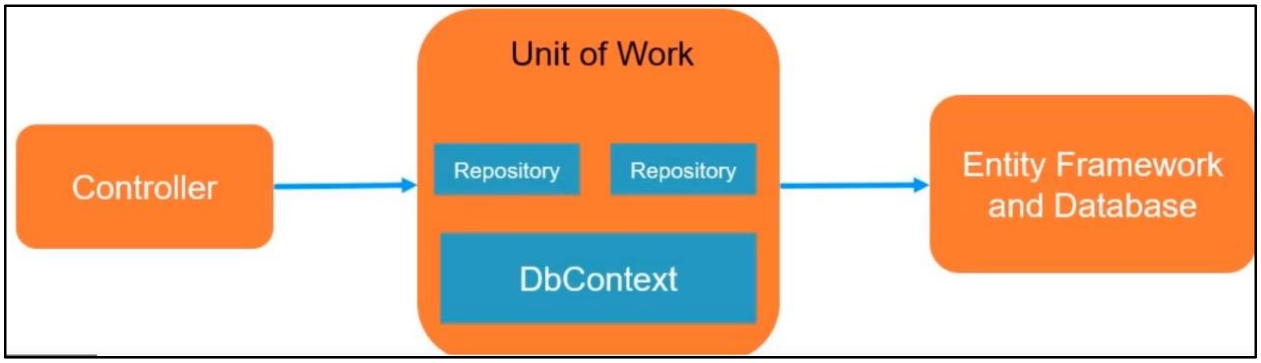
**A repository should not have semantics of database.**

- It should be like a collection of objects in memory and should not have method like update or save.
- Now if we cannot save or update inside the repository, then how are we supposed to do those things?

To update and save operation should be done using ***UNIT OF WORK.***

## 85. UNIT OF WORK Pattern

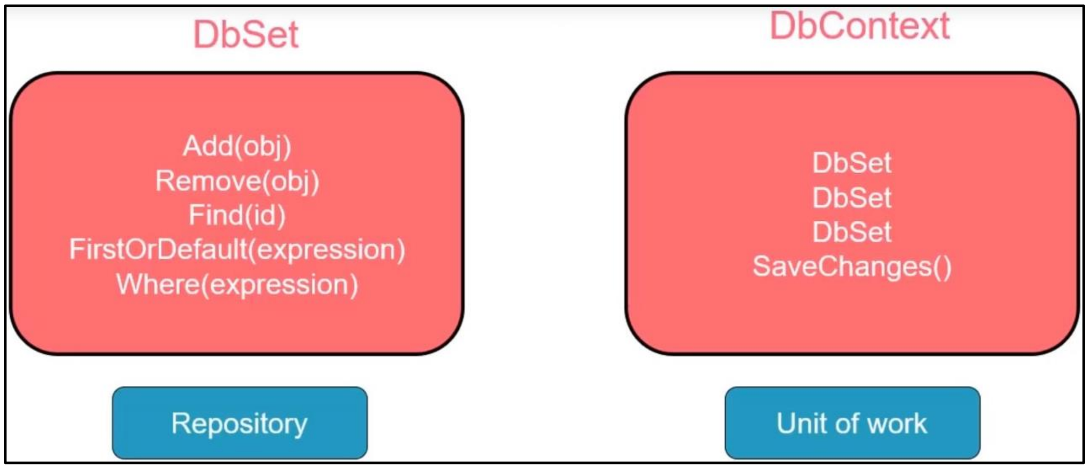
**The unit of work class coordinates the work of multiple repositories by creating a single database context class shared by all of them.**



**It maintains a list of objects affected by the business transaction and coordinates the writing out of the changes.**

- The controller interacts with the unit of work, which has all the repositories and it will ultimately interact with database using entity framework.
- The benefit we can see with it is that ***there is a common dBContext for our repository.***
- Without unit of work, we will be separated by the context object for each repository.

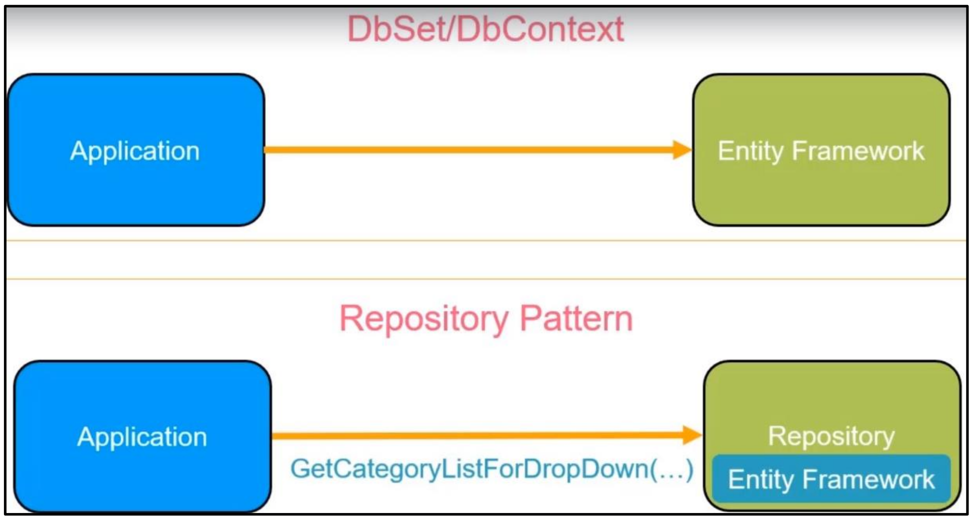
**Common argument that the repository and Unit of work pattern are already implemented in the entity framework and there is no need to recreate them, this would lead to unnecessary complexity.**



- ***DbSet*** has a collection like interface, so it has methods like add, remove, find, etc., it does not have methods like update and save.
- Then ***DBContext*** act as unit of work and keep tracks of changes and save them to the database by using ***SaveChanges*** method.
- ***Save*** method should not be in the repository.
- In business transaction, we may work with more than one repository.
  - For example, we have category, order, header, details repositories and so on.
  - Hence, it is better to have save method inside the unit of work so it coordinate persistent changes across multiple repositories in a single transaction.

**We can agree that framework *DbSet* and *DbContext* look like repositories and unit of work, but there is a problem in this implementation.**

- *Repository pattern minimize the duplicate query logic*.
- The ***problem with DbSet*** is that to get the dropdown for category, we will end up repeating the same code in the different places in our application and a change in one place will have to be updated in all the places as Traditional approach.
- So, the implementation of DbSet not really help with minimizing duplicate query logic.
- In the case, we want a repository that has a method like ***getCategoryList*** for dropdown and all the logic should be wrapped within that method.



**The second benefit for repository pattern is that it decouple our application from persistence framework.**

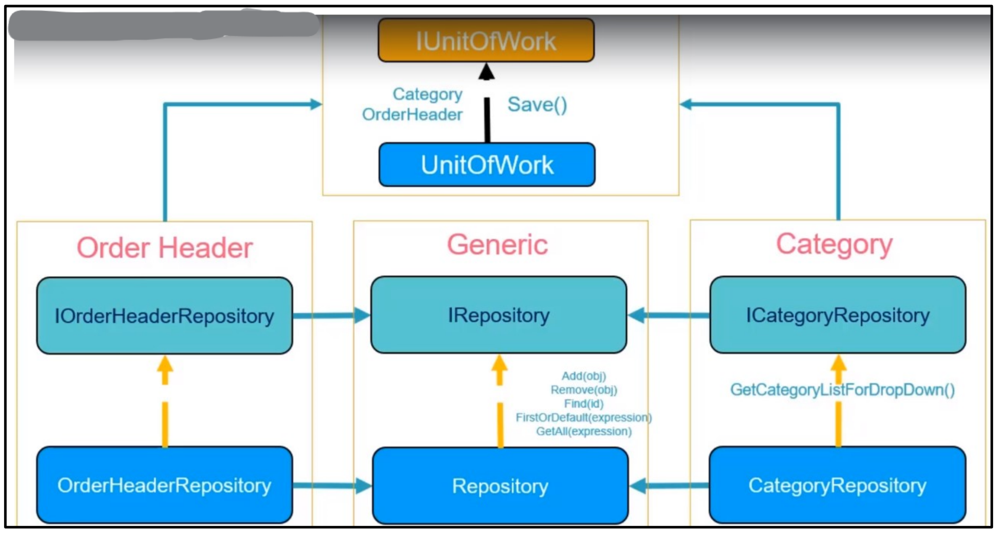
- When we use DbContext and DbSet directly in our application, our application is tightly coupled to entity framework.
- If we want to upgrade to different ORM, we need to modify our application code directly.
- However, if all this is behind the repository and our application code relies on the repository to return the right data, it doesn't matter what is inside the repository.
- The application code will not be affected in the future when we decide to change the **ORM**, and that is because repository does its job of retrieving and saving data.
- It doesn't matter what inside a repository and how it's done because of this difference, even though DbSet and DbContext seem like implementation of repository and unit of work but they don't bring all the advantages as repository pattern.

**Remember that not all project would need such sophisticated architecture.**

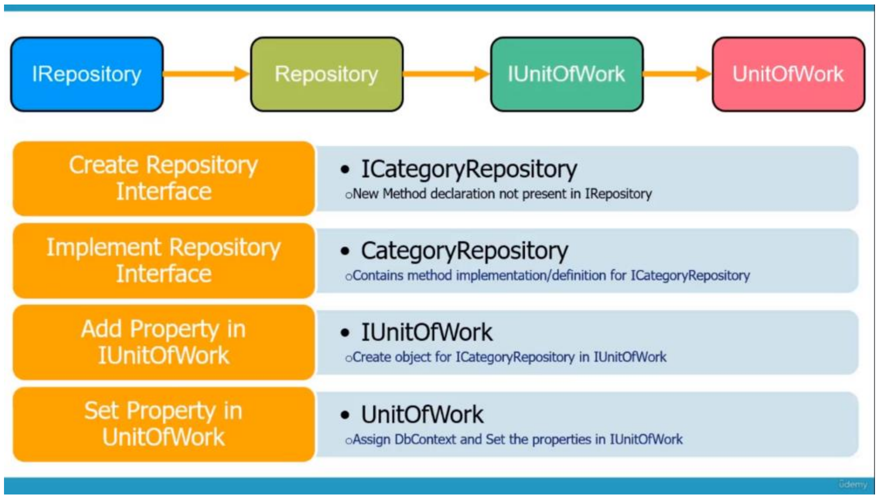
**Many time if we need a basic application or maybe *an application where requirement are changing very frequently*, it is recommended to not use this pattern because using patterns would always increase the time to implement the project.**

- For example, a basic CRUD application website for about 50 tables which were reference tables for an SISS package and business is constantly changing that requirement and project was on a strict timeline. To complete the project in time with changing requirement, we should not use repository pattern.

## 86. Repository Pattern Flow



1. **We implement the interface for a repository with name IRepository.**
   - This will have ***generic method*** definitions for get, getAll, add, remove, firstOrDefault or any other common methods that we want to implement.
   - The implementation of these methods will be lie inside the repository class.
   - Once we complete the generic methods for the repository and its interface that have consumed this repositories.
2. **We implement an interface for Category repository and then we will define any custom methods that we want like GetCategoryList for dropdown.**
   - Implementation will lie inside the category repository class.
   - If we have another model for order, header and maybe other details, and much more so they will create their repositories then the class for implementation that interface.
3. Create interface for unit of work and then the class which implements the interface with the same methods.
   - We create our DbContext inside unit of work as well.



| | |
|---|---|
| Create Repository Interface | • ICategoryRepository<br>o New Method declaration not present in IRepository |
| Implement Repository Interface | • CategoryRepository<br>o Contains method implementation/definition for ICategoryRepository |
| Add Property in IUnitOfWork | • IUnitOfWork<br>o Create object for ICategoryRepository in IUnitOfWork |
| Set Property in UnitOfWork | • UnitOfWork<br>o Assign DbContext and Set the properties in IUnitOfWork |

## 87. Define IRepository Interface

Since it is a big project, it is always good to have a repository pattern rather than accessing database directly.

1. **Create a folder named with IRepository inside the DataAccess library.**
    - It holds all the interface and the implementation will be inside the repository.
    - Implement the interface and define all the generic methods.
2. **Create a folder named with Repository inside the IRepository folder.**

```
namespace BuckyBookDataAccess.IRepository
{
    public interface IRepository<T> where T : class
    {
       // it is a generic repository
       // assume T is Category class
       // define All the common methods that we want to implement

        // Expression of type Function of T class, boolean for linq expression
        T GetFirstOrDefault(Expression<Func<T,bool>> filter );
        IEnumerable<T> GetAll();
        void Add(T entity);
        void Remove(T entity);
        void RemoveRange(IEnumerable<T> entities);

    }
}
```

## 87. Repository class

1. Create a class with named “Repository” inside Repository folder.

- Inject the DbContext using the dependency injection and it will be the final place to interact with the database.

```
namespace BuckyBookDataAccess.IRepository.Repository
{
    // it is a generic class
    public class Repository<T> : IRepository<T> where T : class
    {
        private readonly ApplicationDbContext _db;
        internal DbSet<T> dbSet;

        // constructor to inject DbContext
        public Repository(ApplicationDbContext dbContext)
        {
            _db = dbContext;
              // From DbContext, we will get the DbSet and work on that directly.
            // set our dbSet to the particular instance of the class
            this.dbSet = _db.Set<T>();
        }

        public void Add(T entity)
        {

            // dbset is basically working same as _db.Categories.Add(obj)
            dbSet.Add(entity);
        }

        public IEnumerable<T> GetAll()
        {

            IQueryable<T> query = dbSet;

            return query.ToList();
        }

        public T GetFirstOrDefault(Expression<Func<T, bool>> filter)
        {
            IQueryable<T> query = dbSet;

            // return a filter data from which we get first record in next statement.
            query = query.Where(filter);

            return query.FirstOrDefault();
        }

        public void Remove(T entity)
        {
            dbSet.Remove(entity);
        }

        public void RemoveRange(IEnumerable<T> entities)
        {
            dbSet.RemoveRange(entities);
        }
    }
}
```

## Data Annotation

**Data Annotation Namespace**: ***System.ComponentModel.DataAnnotations*** Namespace

- ***Provides attribute classes that are used to define metadata for ASP.NET MVC and ASP.NET data controls.***

***Reference***: System.ComponentModel.DataAnnotations Namespace | Microsoft Learn

| Classes | Description |
|---|---|
| **AssociatedMetadataTypeTypeDescriptionProvider** | Extends the metadata information for a class by adding attributes and property information that is defined in an associated class. |
| **AssociationAttribute** | Specifies that an entity member represents a data relationship, such as a foreign key relationship. |
| **CompareAttribute** | Provides an attribute that compares two properties. |
| **ConcurrencyCheckAttribute** | Specifies that a property participates in optimistic concurrency checks. |
| **CreditCardAttribute** | Specifies that a data field value is a credit card number. |
| **CustomValidationAttribute** | Specifies a custom validation method that is used to validate a property or class instance. |
| **DataTypeAttribute** | Specifies the name of an additional type to associate with a data field. |
| **DisplayAttribute** | Provides a general-purpose attribute that lets you specify localizable strings for types and members of entity partial classes. |
| **DisplayColumnAttribute** | Specifies the column that is displayed in the referred table as a foreign-key column. |
| **DisplayFormatAttribute** | Specifies how data fields are displayed and formatted by ASP.NET Dynamic Data. |
| **EditableAttribute** | Indicates whether a data field is editable. |
| **EmailAddressAttribute** | Validates an email address. |
| **EnumDataTypeAttribute** | Enables a .NET enumeration to be mapped to a data column. |
| **FileExtensionsAttribute** | Validates file name extensions. |
| **FilterUIHintAttribute** | Represents an attribute that is used to specify the filtering behavior for a column. |
| **KeyAttribute** | Denotes one or more properties that uniquely identify an entity. |
| **MaxLengthAttribute** | Specifies the maximum length of array or string data allowed in a property. |
| **MetadataTypeAttribute** | Specifies the metadata class to associate with a data model class. |
| **MinLengthAttribute** | Specifies the minimum length of array or string data allowed in a property. |
| **PhoneAttribute** | Specifies that a data field value is a well-formed phone number. |
| **RangeAttribute** | Specifies the numeric range constraints for the value of a data field. |
| **RegularExpressionAttribute** | Specifies that a data field value in ASP.NET Dynamic Data must match the specified regular expression. |
| **RequiredAttribute** | Specifies that a data field value is required. |
| **ScaffoldColumnAttribute** | Specifies whether a class or data column uses scaffolding. |
| **StringLengthAttribute** | Specifies the minimum and maximum length of characters that are allowed in a data field. |
| **TimestampAttribute** | Specifies the data type of the column as a row version. |
| **UIHintAttribute** | Specifies the template or user control that Dynamic Data uses to display a data field. |
| **UrlAttribute** | Provides URL validation. |

| | |
|---|---|
| **ValidationAttribute** | Serves as the base class for all validation attributes. |
| **ValidationContext** | Describes the context in which a validation check is performed. |
| **ValidationException** | Represents the exception that occurs during validation of a data field when the ValidationAttribute class is used. |
| **ValidationResult** | Represents a container for the results of a validation request. |
| **Validator** | Defines a helper class that can be used to validate objects, properties, and methods when it is included in their associated ValidationAttribute attributes. |
| | |
| ***Interfaces*** | |
| | |
| **IValidatableObject** | Provides a way for an object to be validated. |
| | |
| ***Enums*** | |
| | |
| **DataType** | Represents an enumeration of the data types associated with data fields and parameters. |

## 87. Create IRepository Interface

**For big project, it is always good to have a repository pattern rather than accessing database directly.**

1. **Create a folder with name '*IRepository*' in DataAccess Library folder which holds all the interfaces.**
2. **Create a folder with name '*Repository*' in DataAccess Library> IRepository folder which holds all the implementation class..**
3. **Create an interface in IRepository named with '' inside '*IRepository*' *folder.***
    - This interface should hold generic definitions..
    - This should be able to handle all of the classes.

```
namespace BuckyBookDataAccess.IRepository
{
    public interface IRepository<T> where T : class
    {
       // it is a generic repository
       // assume T is Category class
       // define All the common methods that we want to implement

        // Expression of type Function of T class, boolean for LINQ expression
        T GetFirstOrDefault(Expression<Func<T,bool>> filter );
        IEnumerable<T> GetAll();
        void Add(T entity);
        void Remove(T entity);
        void RemoveRange(IEnumerable<T> entities);

    }
}
```

## 88. Implement Repository class

**The repository will be the the final place where we are interacting with the database.**

1. **Create a class named with 'Repository' inside the folder DataAccess > IRepository > Repository.**
   - Repository class implement the IRepository<T> interface.
   - Implement all the defined method in Interface into the class.
2. Inject the DbContext using dependency injection to interact with the database.

```
private readonly ApplicationDbContext _db;

// constructor to inject DbContext
public Repository(ApplicationDbContext dbContext)
{
        _db = dbContext;
}
```

3. Add generic ***DbSet*** of type T (where T is a class) variable to store the returned from DBContext.
   - In constructor, we initialize the dbset from dbcontext of particular class.

**From our Application DbContext, DbSet are doing all the transactions.**

- From our DbContext, we can get the DbSet and work on that directly.

```
namespace BuckyBookDataAccess.IRepository.Repository
{
    // it is a generic class
    public class Repository<T> : IRepository<T> where T : class
    {
        private readonly ApplicationDbContext _db;
        internal DbSet<T> dbSet;

        // constructor to inject DbContext
        public Repository(ApplicationDbContext dbContext)
        {
            _db = dbContext;
            // set our dbSet to the particular instance of the class
            this,dbSet = _db.Set<T>();
        }

        public void Add(T entity)
        {

            // dbset is basically working same as _db.Categories.Add(obj)
            dbSet.Add(entity);
        }

        public IEnumerable<T> GetAll()
        {
           //
            IQueryable<T> query = dbSet;

            return query.ToList();
        }

        public T GetFirstOrDefault(Expression<Func<T, bool>> filter)
        {
            IQueryable<T> query = dbSet;

            // return a filter data from which we get first record in next statement.
            query = query.Where(filter);

            return query.FirstOrDefault();
        }

        public void Remove(T entity)
        {
            dbSet.Remove(entity);
        }

        public void RemoveRange(IEnumerable<T> entities)
        {
            dbSet.RemoveRange(entities);
        }
    }
}
```

## 89. Category Repository Interface

**Now that we have our generic repository, we will be consuming this generic repository inside the model specific repository.**

1. **Create an interface with name "*ICategory*" Repository.**
    - It will implements the generic IRepository interface of type Category.
    - We are specifying that when we are using ***Category repository***, the model will be ***category***.
    - On the implementation of category repository, if required, it will retrieve all the categories.
    - So that a category repository will get all the methods that are implemented inside the generic repository.
2. **Create a method declaration for update function.**
3. **Create a save method.**
    - The save method implemented inside the category repository, so we have to explicitly call when we want to save the changes.

**Method like update or some other custom message that have different implementation in the different repository will be implemented inside the specific interface.**

```
internal interface ICategoryRespository : IRepository<Category>
{
    // to update the category we implemnt the this method.
    void Update(Category obj);
    //to explicitly call when we want to save the changes to DB.
    void Save();
}
```

## 90. Implement Category Repository

1. **1. Create a class with name *CategoryRepository* which implements the interface *ICategoryRepository*.**
    - We need to inherit ***Repository*** class and interface because implementation is inside the repository class for IRepository
2. **Inject the application DBContext using dependency injection in class constructor.**
    - We need to pass the dB context object to base class because the Repository class also expects the dBContext object.

```
namespace BuckyBookDataAccess.IRepository.Repository
{
    // use to implement ICategory
    public class CategoryRepository : Repository<Category>, ICategoryRespository
    {
        // inherit the class Repository<Category> due to it contains the implementation of
interface IRepository which is generic functionality.
        // ICategoryRespository interface inherits IRepository interface

        //inject the application DBContext using dependency injection
        private ApplicationDbContext _db;

        public CategoryRepository(ApplicationDbContext db) : base(db)
        {
            // pass the db object to the base class
            // because the Respository expects db context object
            _db = db;
        }

        // now we need to implements the remaining Methods which we declared in
ICategoryRespository
        public void Save()
        {
            _db.SaveChanges();
        }

        public void Update(Category obj)
        {
            _db.Categories.Update(obj);
        }
    }
}
```

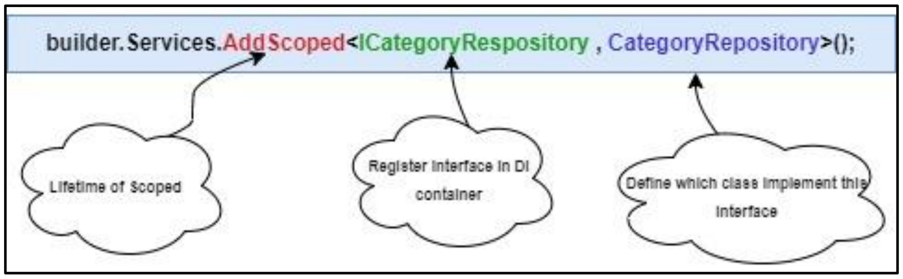
## 91. Replace DbContext with Category Repository

Now we must replace ***ApplicationDbContext*** with the category repository class because we use ***ApplicationDbContext*** in the same class as we get the services we are registered.

1. **Register the Application ApplicationDbContext service in the Dependency Injection container in program.cs file.**

```
    builder.Services.AddScoped<ICategoryRespository, CategoryRepository>();
```

- Whenever we request an object of a category repository, it will give us the implementation that we have define inside the category repository.


builder.Services.AddScoped<ICategoryRespository , CategoryRepository>();
Lifetime of Scoped
Register interface in DI container
Define which class implement this interface

**There are Three types of lifetime while register a service.**

1. **Singleton**: there will be one object of the service which will be used throughout the lifecycle unless we restart the application.
2. **Scoped**: Means when we hit the button and the request is received, that will be the scope of that object.
   - For the next request, a new object instance where we created.
3. **Transient**: It is the smallest of the lifetime. It means every time it will create a new object.
   - Let say, we hit a button to request data from database three times, transient will create three instance of the DbContext.

2. In **Category** Controller, rather than ***ApplicationDbContext***, we want an ***ICategoryRepository*** object through dependency injection.

```
private readonly ICategoryRespository _db;

public CategoryController(ICategoryRespository dbContext)
{
    _db = dbContext;
}
```

3. Modify all method in Category **Controller**.

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Category obj)
{
    if (obj.Name == obj.DisplayOrder.ToString())
    {
        ModelState.AddModelError("CustomError", "The display order cannot exactly match the name");
    }

    // check if model is valid
    if (ModelState.IsValid)
    {
         // add the entry to db by using Repository implemented method
       _db.Add(obj);

       // push the changes to db using CategoryRepository implemented method
       _db.Save();

       TempData["Success"] = "Category created successfully";
       return RedirectToAction("Index");
     }
       return View(obj);
}

public IActionResult Edit(int? id)
{
    // check if id is valid or not
    if (id == null || id == 0)
        return NotFound();


    // fetch the category details
    //var CategoryFromDb = _db.Find(id);
    //var CategoryFromDbSingle = _db.Categories.SingleOrDefault(u => u.Id == id);
    var CategoryFromDbFirst = _db.GetFirstOrDefault(u => u.Id == id);

    // check if fetch data is null
    if (CategoryFromDbFirst == null)
        return NotFound();


    // if fetch data is not null then return data to view
    return View(CategoryFromDbFirst);
}

[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(Category obj)
{

         if (obj.Name == obj.DisplayOrder.ToString())
         {
         ModelState.AddModelError("CustomError", "The display order cannot exactly match the name");
                  }

                  // check if model is valid
                  if (ModelState.IsValid)
                  {
                           // update the record based on primary key to db
                           _db.Update(obj);

                           // push the changes to db
                           _db.Save();
                           TempData["Success"] = "Category updated successfully";
                           return RedirectToAction("Index");
                  }
                  return View(obj);
         }
}
```

```
[HttpGet]
public IActionResult Delete(int? id)
{
         // check if id is valid or not
         if (id == null || id == 0)
                  return NotFound();


         // fetch the category details
         //var CategoryFromDb = _db.Categories.Find(id);
         //var CategoryFromDbSingle = _db.Categories.SingleOrDefault(u => u.Id == id);
         var CategoryFromDbFirst = _db.GetFirstOrDefault(u => u.Id == id);

         // check if fetch data is null
         if (CategoryFromDbFirst == null)
                  return NotFound();


         // if fetch data is not null then return data to view
         return View(CategoryFromDbFirst);
}

        //Now we can use Delete as Action method for post request.
[HttpPost,ActionName("Delete")]
[ValidateAntiForgeryToken]
public IActionResult DeletePost(int? id)
{

      var obj = _db.GetFirstOrDefault(u => u.Id == id);

      if (obj == null)
      {
          return NotFound();
      }


                  // remove the record based on primary key to db
                  _db.Remove(obj);

                  // push the changes to db
                  _db.Save();
      // for success alert
                  TempData["Success"] = "Category deleted successfully";

      return RedirectToAction("Index");

}
```

## 92. Implementing Unit of Work

**Let’s image in a page we are accessing 10 models.**

- In that case, there is a repository for each model, we will have to inject all of them inside the dependency injection.
- Also, we have save method that is implemented at the repository level, but when we see the implementation, we don’t apply that at the any category level, like in update *_db*.
- Same is applicable at a global level. Because of that, this is not as manful as it should be.

**The correct way of doing things is to have this save complimented in something called as unit of work.**

1. Add an interface named with ***IUnitOfWork***.
   - Create all of the repositories, we have ***ICategoryRepository*** that will only have a category.
   - The global method that we want will also be implemented here (Save method).

```
public interface IUnitOfWork
{
    // Create all repository
    ICategoryRespository CategoryRepo { get; }

    // Global method
    void Save();


}
```

2. Create a class with name ***UnitOfWork***.
   - It is a public class and implements the IUnitOfWork interface.
   - On implementing interface, we will have a property (`CategoryRepo`) and a ***Save*** method.
   - Inside the ***CategoryRepository***, we were getting DbContext code with Dependency injection. The same we do in the class. Replicate the constructor of ***CategoryRepository*** *to this class.*
   - Inside constructor, we can create a ***CategoryRepository*** object and pass the object of ***_db*** .

```
public class UnitOfWork : IUnitOfWork
    {
        //inject the application DBContext using dependency injection
        private ApplicationDbContext _db;

        public UnitOfWork(ApplicationDbContext db)
        {
            // pass the db object to the base class
            // because the Respository expects db context object
            _db = db;
            // create object of CategoryRepository
            CategoryRepo = new CategoryRepository(_db);

        }

        // property
        public ICategoryRespository CategoryRepo { get; private set; }

        //method
        public void Save()
        {
            _db.SaveChanges();
        }
    }
```

93.