Construction document

Contents

[Summary of build 3](#_Toc83633579)

[Getting database to show up 3](#_Toc83633580)

[Introduction to Identity on ASP.NET Core 5](#_Toc83633581)

[Create a Web app with authentication 6](#_Toc83633582)

[Apply migrations 6](#_Toc83633583)

[Test Register and Login 6](#_Toc83633584)

[View the Identity database 7](#_Toc83633585)

[Configure Identity services 7](#_Toc83633586)

[Scaffold Register, Login, LogOut, and RegisterConfirmation 9](#_Toc83633587)

[Examine Register 9](#_Toc83633588)

[Disable default account verification 10](#_Toc83633589)

[Log in 11](#_Toc83633590)

[Log out 13](#_Toc83633591)

[Test Identity 14](#_Toc83633592)

[Explore Identity 15](#_Toc83633593)

[Identity Components 15](#_Toc83633594)

[Migrating to ASP.NET Core Identity 15](#_Toc83633595)

[Setting password strength 15](#_Toc83633596)

[AddDefaultIdentity and AddIdentity 15](#_Toc83633597)

[Prevent publish of static Identity assets 16](#_Toc83633598)

[ASP.NET Core MVC 16](#_Toc83633599)

[Get started with ASP.NET Core MVC 16](#_Toc83633600)

[Prerequisites 17](#_Toc83633601)

[Create a web app 17](#_Toc83633602)

[Run the app 18](#_Toc83633603)

[Part 2, add a controller to an ASP.NET Core MVC app 21](#_Toc83633604)

[Add a controller 22](#_Toc83633605)

[Part 4, add a model to an ASP.NET Core MVC app 26](#_Toc83633606)

[Add a data model class 26](#_Toc83633607)

[Add NuGet packages 27](#_Toc83633608)

[Scaffold movie pages 28](#_Toc83633609)

[Initial migration 29](#_Toc83633610)

[Test the app 30](#_Toc83633611)

[Examine the generated database context class and registration 30](#_Toc83633612)

[Examine the generated database connection string 31](#_Toc83633613)

[The InitialCreate class 32](#_Toc83633614)

[Dependency injection in the controller 33](#_Toc83633615)

[Strongly typed models and the @model directive 33](#_Toc83633616)

[SQL Logging of Entity Framework Core 37](#_Toc83633617)

[**Code First Approach With CRUD Operation In Entity Framework In MVC - Part Four** 37](#_Toc83633618)

# Summary of build

1. Set up ASP.NET MVC with identity databases. This produces the different databases but only the userlogins are active e.g. register a user and be able to login in and out.
   1. The source used:<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio>
   2. Still need to add the 3 permission levels/roles and activate corresponding tables with the correct roles and match to registered login accounts:
      1. https://www.c-sharpcorner.com/article/adding-role-authorization-to-a-asp-net-mvc-core-application/
   3. Still need to add Roles to the views.
      1. https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles?view=aspnetcore-5.0
2. Then add the quiz table with CRUD using:
   1. <https://www.c-sharpcorner.com/article/code-first-approach-with-crud-operation-in-entity-framework-in-mvc-part-four/>
3. Add one to many relationship (One=Quiz, many = Questions)
   1. Repeat this:https://www.c-sharpcorner.com/article/code-first-approach-with-crud-operation-in-entity-framework-in-mvc-part-four/
   2. Use this to one to many: <https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx>

# Getting database to show up

1. Set up model with no connections!!!!!

namespace WebApp1.Models

{

public class QuestionDataContext : DbContext

{

//public QuestionDataContext()

// : base("name=DefaultConnection")

public DbSet<Quiz2> Quiz2s { get; set; }

public DbSet<Question> Questions { get; set; }

}

}

1. Then a controller
2. using Microsoft.AspNetCore.Mvc;
3. using System;
4. using System.Collections.Generic;
5. using System.Linq;
6. using System.Threading.Tasks;
7. using WebApp1.Models;
8. namespace WebApp1.Controllers
9. {
10. public class QuestionController : Controller
11. {
12. QuestionDataContext objQuestionDataContext = new QuestionDataContext();
13. public IActionResult QuestionDetails()
14. {
15. return View(objQuestionDataContext.Questions.ToList());
16. }
17. }
18. }
19. Then the view, right click on questiondetails to generate a view use razor view not empty option
    1. Graphical user interface, text, application, email

       Description automatically generated

ApplicationDbContext

In Visual Studio, use the Package Manager Console to scaffold a new migration for these changes and apply them to the database:

PM> Add-Migration [migration name]

PM> Update-Database

This exception was originally thrown at this call stack:

[External Code]

WebApp1.Controllers.QuestionController.QuestionDetails() in [QuestionController.cs](navigate-to-context:15)

[External Code]

**System.InvalidOperationException:** 'The model backing the 'QuestionDataContext' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?LinkId=238269).'

# Update data table code first migration.

1. Enable-Migrations had to use EntityFrameworkCore\Enable-Migrations
2. Add-Migration RemoveCurrentQuizID <- name of migration
   1. EntityFrameworkCore\ Add-Migration AddQuiz2IdandMapping
3. Add-Migration RemoveCurrentQuizID
4. Update-Database –Verbose <- push update to table.

Add-Migration 12many

EntityFrameworkCore\Add-Migration 12many

Notes:

To build a link route to pages this will go to (back to front): https://localhost:44359/Question/QuestionQ/2 @Html.ActionLink(item.QuizTitle, "QuestionQ","Question", new {Id=item.Quiz2Id })

# **A database operation failed while processing the request.**

SqlException: Invalid column name 'Quiz2Id'. Invalid column name 'Quiz2Id'.

## **There are pending model changes**

Pending model changes are detected in the following:

* ApplicationDbContext

In Visual Studio, use the Package Manager Console to scaffold a new migration for these changes and apply them to the database:

PM> Add-Migration [migration name]  
PM> Update-Database

Alternatively, you can scaffold a new migration and apply it from a command prompt at your project directory:

> dotnet ef migrations add [migration name]  
> dotnet ef database update

# Introduction to Identity on ASP.NET Core

https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio

ASP.NET Core Identity:

* Is an API that supports user interface (UI) login functionality.
* Manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more.

Users can create an account with the login information stored in Identity or they can use an external login provider. Supported external login providers include [Facebook, Google, Microsoft Account, and Twitter](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/social/?view=aspnetcore-5.0).

For information on how to globally require all users to be authenticated, see [Require authenticated users](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/secure-data?view=aspnetcore-5.0#rau).

The [Identity source code](https://github.com/dotnet/AspNetCore/tree/main/src/Identity) is available on GitHub. [Scaffold Identity](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity?view=aspnetcore-5.0) and view the generated files to review the template interaction with Identity.

Identity is typically configured using a SQL Server database to store user names, passwords, and profile data. Alternatively, another persistent store can be used, for example, Azure Table Storage.

In this topic, you learn how to use Identity to register, log in, and log out a user. Note: the templates treat username and email as the same for users. For more detailed instructions about creating apps that use Identity, see [Next Steps](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#next).

[Microsoft identity platform](https://docs.microsoft.com/en-us/azure/active-directory/develop/) is:

* An evolution of the Azure Active Directory (Azure AD) developer platform.
* Unrelated to ASP.NET Core Identity.

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

* [Azure Active Directory](https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-protect-backend-with-aad)
* [Azure Active Directory B2C](https://docs.microsoft.com/en-us/azure/active-directory-b2c/active-directory-b2c-custom-rest-api-netfw) (Azure AD B2C)
* [IdentityServer4](https://identityserver.io/)

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. IdentityServer4 enables the following security features:

* Authentication as a Service (AaaS)
* Single sign-on/off (SSO) over multiple application types
* Access control for APIs
* Federation Gateway

For more information, see [Welcome to IdentityServer4](https://docs.identityserver.io/en/latest/index.html).

[View or download the sample code](https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/security/authentication/identity/sample) ([how to download](https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0#how-to-download-a-sample)).

## Create a Web app with authentication

Create an ASP.NET Core Web Application project with Individual User Accounts.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_4_visual-studio)
* [.NET Core CLI](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_4_netcore-cli)
* Select **File** > **New** > **Project**.
* Select **ASP.NET Core Web Application**. Name the project **WebApp1** to have the same namespace as the project download. Click **OK**.
* Select an ASP.NET Core **Web Application**, then select **Change Authentication**.
* Select **Individual User Accounts** and click **OK**.

The generated project provides [ASP.NET Core Identity](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0) as a [Razor Class Library](https://docs.microsoft.com/en-us/aspnet/core/razor-pages/ui-class?view=aspnetcore-5.0). The Identity Razor Class Library exposes endpoints with the Identity area. For example:

* /Identity/Account/Login
* /Identity/Account/Logout
* /Identity/Account/Manage

### Apply migrations

Apply the migrations to initialize the database.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_5_visual-studio)
* [.NET Core CLI](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_5_netcore-cli)

Run the following command in the Package Manager Console (PMC):

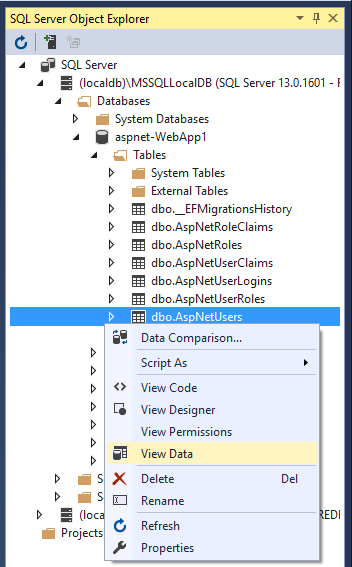
PM> Update-Database

### Test Register and Login

Run the app and register a user. Depending on your screen size, you might need to select the navigation toggle button to see the **Register** and **Login** links.

### View the Identity database

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_1_visual-studio)
* [.NET Core CLI](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_1_netcore-cli)
* From the **View** menu, select **SQL Server Object Explorer** (SSOX).
* Navigate to **(localdb)MSSQLLocalDB(SQL Server 13)**. Right-click on **dbo.AspNetUsers** > **View Data**:



### Configure Identity services

Services are added in ConfigureServices. The typical pattern is to call all the Add{Service} methods, and then call all the services.Configure{Service} methods.

C#Copy

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ApplicationDbContext>(options =>

// options.UseSqlite(

options.UseSqlServer(

Configuration.GetConnectionString("DefaultConnection")));

services.AddDatabaseDeveloperPageExceptionFilter();

services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)

.AddEntityFrameworkStores<ApplicationDbContext>();

services.AddRazorPages();

services.Configure<IdentityOptions>(options =>

{

// Password settings.

options.Password.RequireDigit = true;

options.Password.RequireLowercase = true;

options.Password.RequireNonAlphanumeric = true;

options.Password.RequireUppercase = true;

options.Password.RequiredLength = 6;

options.Password.RequiredUniqueChars = 1;

// Lockout settings.

options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);

options.Lockout.MaxFailedAccessAttempts = 5;

options.Lockout.AllowedForNewUsers = true;

// User settings.

options.User.AllowedUserNameCharacters =

"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-.\_@+";

options.User.RequireUniqueEmail = false;

});

services.ConfigureApplicationCookie(options =>

{

// Cookie settings

options.Cookie.HttpOnly = true;

options.ExpireTimeSpan = TimeSpan.FromMinutes(5);

options.LoginPath = "/Identity/Account/Login";

options.AccessDeniedPath = "/Identity/Account/AccessDenied";

options.SlidingExpiration = true;

});

}

The preceding code configures Identity with default option values. Services are made available to the app through [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0).

Identity is enabled by calling [UseAuthentication](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.authappbuilderextensions.useauthentication#Microsoft_AspNetCore_Builder_AuthAppBuilderExtensions_UseAuthentication_Microsoft_AspNetCore_Builder_IApplicationBuilder_). UseAuthentication adds authentication [middleware](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-5.0) to the request pipeline.

C#Copy

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

app.UseMigrationsEndPoint();

}

else

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapRazorPages();

});

}

The template-generated app doesn't use [authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/secure-data?view=aspnetcore-5.0). app.UseAuthorization is included to ensure it's added in the correct order should the app add authorization. UseRouting, UseAuthentication, UseAuthorization, and UseEndpoints must be called in the order shown in the preceding code.

For more information on IdentityOptions and Startup, see [IdentityOptions](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.identityoptions) and [Application Startup](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-5.0).

## Scaffold Register, Login, LogOut, and RegisterConfirmation

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_6_visual-studio)
* [.NET Core CLI](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_6_netcore-cli)

Add the Register, Login, LogOut, and RegisterConfirmation files. Follow the [Scaffold identity into a Razor project with authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity?view=aspnetcore-5.0#scaffold-identity-into-a-razor-project-with-authorization) instructions to generate the code shown in this section.

### Examine Register

When a user clicks the **Register** button on the Register page, the RegisterModel.OnPostAsync action is invoked. The user is created by [CreateAsync](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.usermanager-1.createasync#Microsoft_AspNetCore_Identity_UserManager_1_CreateAsync__0_System_String_) on the \_userManager object:

C#Copy

public async Task<IActionResult> OnPostAsync(string returnUrl = null)

{

returnUrl = returnUrl ?? Url.Content("~/");

ExternalLogins = (await \_signInManager.GetExternalAuthenticationSchemesAsync())

.ToList();

if (ModelState.IsValid)

{

var user = new IdentityUser { UserName = Input.Email, Email = Input.Email };

var result = await \_userManager.CreateAsync(user, Input.Password);

if (result.Succeeded)

{

\_logger.LogInformation("User created a new account with password.");

var code = await \_userManager.GenerateEmailConfirmationTokenAsync(user);

code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));

var callbackUrl = Url.Page(

"/Account/ConfirmEmail",

pageHandler: null,

values: new { area = "Identity", userId = user.Id, code = code },

protocol: Request.Scheme);

await \_emailSender.SendEmailAsync(Input.Email, "Confirm your email",

$"Please confirm your account by <a href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

if (\_userManager.Options.SignIn.RequireConfirmedAccount)

{

return RedirectToPage("RegisterConfirmation",

new { email = Input.Email });

}

else

{

await \_signInManager.SignInAsync(user, isPersistent: false);

return LocalRedirect(returnUrl);

}

}

foreach (var error in result.Errors)

{

ModelState.AddModelError(string.Empty, error.Description);

}

}

// If we got this far, something failed, redisplay form

return Page();

}

### Disable default account verification

With the default templates, the user is redirected to the Account.RegisterConfirmation where they can select a link to have the account confirmed. The default Account.RegisterConfirmation is used ***only*** for testing, automatic account verification should be disabled in a production app.

To require a confirmed account and prevent immediate login at registration, set DisplayConfirmAccountLink = false in /Areas/Identity/Pages/Account/RegisterConfirmation.cshtml.cs:

C#Copy

[AllowAnonymous]

public class RegisterConfirmationModel : PageModel

{

private readonly UserManager<IdentityUser> \_userManager;

private readonly IEmailSender \_sender;

public RegisterConfirmationModel(UserManager<IdentityUser> userManager, IEmailSender sender)

{

\_userManager = userManager;

\_sender = sender;

}

public string Email { get; set; }

public bool DisplayConfirmAccountLink { get; set; }

public string EmailConfirmationUrl { get; set; }

public async Task<IActionResult> OnGetAsync(string email, string returnUrl = null)

{

if (email == null)

{

return RedirectToPage("/Index");

}

var user = await \_userManager.FindByEmailAsync(email);

if (user == null)

{

return NotFound($"Unable to load user with email '{email}'.");

}

Email = email;

// Once you add a real email sender, you should remove this code that lets you confirm the account

DisplayConfirmAccountLink = false;

if (DisplayConfirmAccountLink)

{

var userId = await \_userManager.GetUserIdAsync(user);

var code = await \_userManager.GenerateEmailConfirmationTokenAsync(user);

code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));

EmailConfirmationUrl = Url.Page(

"/Account/ConfirmEmail",

pageHandler: null,

values: new { area = "Identity", userId = userId, code = code, returnUrl = returnUrl },

protocol: Request.Scheme);

}

return Page();

}

}

### Log in

The Login form is displayed when:

* The **Log in** link is selected.
* A user attempts to access a restricted page that they aren't authorized to access **or** when they haven't been authenticated by the system.

When the form on the Login page is submitted, the OnPostAsync action is called. PasswordSignInAsync is called on the \_signInManager object.

C#Copy

public async Task<IActionResult> OnPostAsync(string returnUrl = null)

{

returnUrl = returnUrl ?? Url.Content("~/");

if (ModelState.IsValid)

{

// This doesn't count login failures towards account lockout

// To enable password failures to trigger account lockout,

// set lockoutOnFailure: true

var result = await \_signInManager.PasswordSignInAsync(Input.Email,

Input.Password, Input.RememberMe, lockoutOnFailure: true);

if (result.Succeeded)

{

\_logger.LogInformation("User logged in.");

return LocalRedirect(returnUrl);

}

if (result.RequiresTwoFactor)

{

return RedirectToPage("./LoginWith2fa", new

{

ReturnUrl = returnUrl,

RememberMe = Input.RememberMe

});

}

if (result.IsLockedOut)

{

\_logger.LogWarning("User account locked out.");

return RedirectToPage("./Lockout");

}

else

{

ModelState.AddModelError(string.Empty, "Invalid login attempt.");

return Page();

}

}

// If we got this far, something failed, redisplay form

return Page();

}

For information on how to make authorization decisions, see [Introduction to authorization in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction?view=aspnetcore-5.0).

### Log out

The **Log out** link invokes the LogoutModel.OnPost action.

C#Copy

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.RazorPages;

using Microsoft.Extensions.Logging;

using System.Threading.Tasks;

namespace WebApp1.Areas.Identity.Pages.Account

{

[AllowAnonymous]

public class LogoutModel : PageModel

{

private readonly SignInManager<IdentityUser> \_signInManager;

private readonly ILogger<LogoutModel> \_logger;

public LogoutModel(SignInManager<IdentityUser> signInManager, ILogger<LogoutModel> logger)

{

\_signInManager = signInManager;

\_logger = logger;

}

public void OnGet()

{

}

public async Task<IActionResult> OnPost(string returnUrl = null)

{

await \_signInManager.SignOutAsync();

\_logger.LogInformation("User logged out.");

if (returnUrl != null)

{

return LocalRedirect(returnUrl);

}

else

{

return RedirectToPage();

}

}

}

}

In the preceding code, the code return RedirectToPage(); needs to be a redirect so that the browser performs a new request and the identity for the user gets updated.

[SignOutAsync](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.signinmanager-1.signoutasync#Microsoft_AspNetCore_Identity_SignInManager_1_SignOutAsync) clears the user's claims stored in a cookie.

Post is specified in the Pages/Shared/\_LoginPartial.cshtml:

CSHTMLCopy

@using Microsoft.AspNetCore.Identity

@inject SignInManager<IdentityUser> SignInManager

@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">

@if (SignInManager.IsSignedIn(User))

{

<li class="nav-item">

<a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Manage/Index"

title="Manage">Hello @User.Identity.Name!</a>

</li>

<li class="nav-item">

<form class="form-inline" asp-area="Identity" asp-page="/Account/Logout"

asp-route-returnUrl="@Url.Page("/", new { area = "" })"

method="post" >

<button type="submit" class="nav-link btn btn-link text-dark">Logout</button>

</form>

</li>

}

else

{

<li class="nav-item">

<a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Register">Register</a>

</li>

<li class="nav-item">

<a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Login">Login</a>

</li>

}

</ul>

## Test Identity

The default web project templates allow anonymous access to the home pages. To test Identity, add [[Authorize]](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authorization.authorizeattribute):

C#Copy

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc.RazorPages;

using Microsoft.Extensions.Logging;

namespace WebApp1.Pages

{

[Authorize]

public class PrivacyModel : PageModel

{

private readonly ILogger<PrivacyModel> \_logger;

public PrivacyModel(ILogger<PrivacyModel> logger)

{

\_logger = logger;

}

public void OnGet()

{

}

}

}

If you are signed in, sign out. Run the app and select the **Privacy** link. You are redirected to the login page.

### Explore Identity

To explore Identity in more detail:

* [Create full identity UI source](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity?view=aspnetcore-5.0#create-full-identity-ui-source)
* Examine the source of each page and step through the debugger.

## Identity Components

All the Identity-dependent NuGet packages are included in the [ASP.NET Core shared framework](https://docs.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-3.0?view=aspnetcore-5.0#use-the-aspnet-core-shared-framework).

The primary package for Identity is [Microsoft.AspNetCore.Identity](https://www.nuget.org/packages/Microsoft.AspNetCore.Identity/). This package contains the core set of interfaces for ASP.NET Core Identity, and is included by Microsoft.AspNetCore.Identity.EntityFrameworkCore.

## Migrating to ASP.NET Core Identity

For more information and guidance on migrating your existing Identity store, see [Migrate Authentication and Identity](https://docs.microsoft.com/en-us/aspnet/core/migration/identity?view=aspnetcore-5.0).

## Setting password strength

See [Configuration](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio#pw) for a sample that sets the minimum password requirements.

## AddDefaultIdentity and AddIdentity

[AddDefaultIdentity](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.identityservicecollectionuiextensions.adddefaultidentity) was introduced in ASP.NET Core 2.1. Calling AddDefaultIdentity is similar to calling the following:

* [AddIdentity](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.identityservicecollectionextensions.addidentity)
* [AddDefaultUI](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.identitybuilderuiextensions.adddefaultui)
* [AddDefaultTokenProviders](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.identitybuilderextensions.adddefaulttokenproviders)

See [AddDefaultIdentity source](https://github.com/dotnet/AspNetCore/blob/release/3.1/src/Identity/UI/src/IdentityServiceCollectionUIExtensions.cs#L47-L63) for more information.

## Prevent publish of static Identity assets

To prevent publishing static Identity assets (stylesheets and JavaScript files for Identity UI) to the web root, add the following ResolveStaticWebAssetsInputsDependsOn property and RemoveIdentityAssets target to the app's project file:

XMLCopy

<PropertyGroup>

<ResolveStaticWebAssetsInputsDependsOn>RemoveIdentityAssets</ResolveStaticWebAssetsInputsDependsOn>

</PropertyGroup>

<Target Name="RemoveIdentityAssets">

<ItemGroup>

<StaticWebAsset Remove="@(StaticWebAsset)" Condition="%(SourceId) == 'Microsoft.AspNetCore.Identity.UI'" />

</ItemGroup>

</Target>

# ASP.NET Core MVC

## Get started with ASP.NET Core MVC

https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the [Razor Pages](https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start?view=aspnetcore-5.0) version of this tutorial, which provides an easier starting point. See [Choose an ASP.NET Core UI](https://docs.microsoft.com/en-us/aspnet/core/tutorials/choose-web-ui?view=aspnetcore-5.0), which compares Razor Pages, MVC, and Blazor for UI development.

This is the first tutorial of a series that teaches ASP.NET Core MVC web development with controllers and views.

At the end of the series, you'll have an app that manages and displays movie data. You learn how to:

* Create a web app.
* Add and scaffold a model.
* Work with a database.
* Add search and validation.

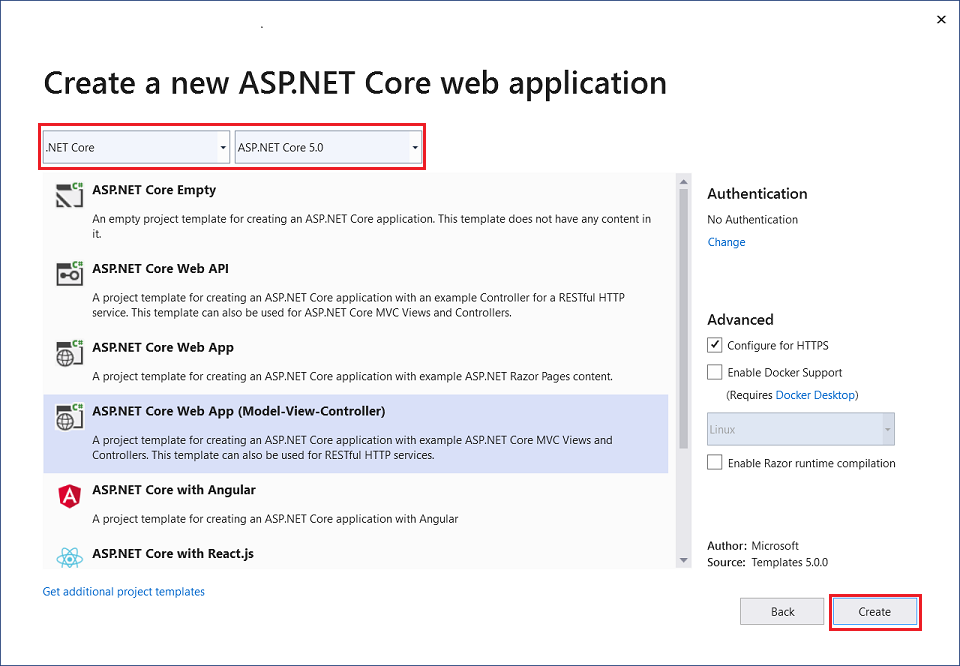
[View or download sample code](https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/tutorials/first-mvc-app/start-mvc/sample) ([how to download](https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0#how-to-download-a-sample)).

## Prerequisites

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_4_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_4_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_4_visual-studio-mac)
* [Visual Studio 2019 16.8 or later](https://visualstudio.microsoft.com/downloads/?utm_medium=microsoft&utm_source=docs.microsoft.com&utm_campaign=inline+link&utm_content=download+vs2019) with the **ASP.NET and web development** workload
* [.NET 5.0 SDK](https://dotnet.microsoft.com/download/dotnet/5.0)

## Create a web app

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_5_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_5_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_5_visual-studio-mac)
* Start Visual Studio and select **Create a new project**.
* In the **Create a new project** dialog, select **ASP.NET Core Web Application** > **Next**.
* In the **Configure your new project** dialog, enter MvcMovie for **Project name**. It's important to name the project MvcMovie. Capitalization needs to match each namespace matches when code is copied.
* Select **Create**.
* In the **Create a new ASP.NET Core web application** dialog, select:
  + **.NET Core** and **ASP.NET Core 5.0** in the dropdowns.
  + **ASP.NET Core Web App (Model-View-Controller)**.
  + **Create**.



For alternative approaches to create the project, see [Create a new project in Visual Studio](https://docs.microsoft.com/en-us/visualstudio/ide/create-new-project).

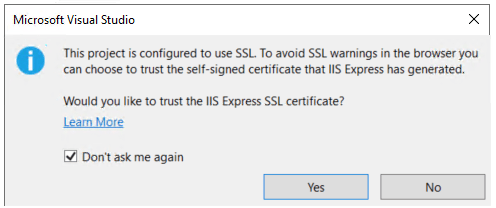
Visual Studio used the default project template for the created MVC project. The created project:

* Is a working app.
* Is a basic starter project.

### Run the app

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_6_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_6_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_6_visual-studio-mac)
* Select Ctrl+F5 to run the app without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see [Firefox SEC\_ERROR\_INADEQUATE\_KEY\_USAGE certificate error](https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-5.0#trust-ff).

Visual Studio:

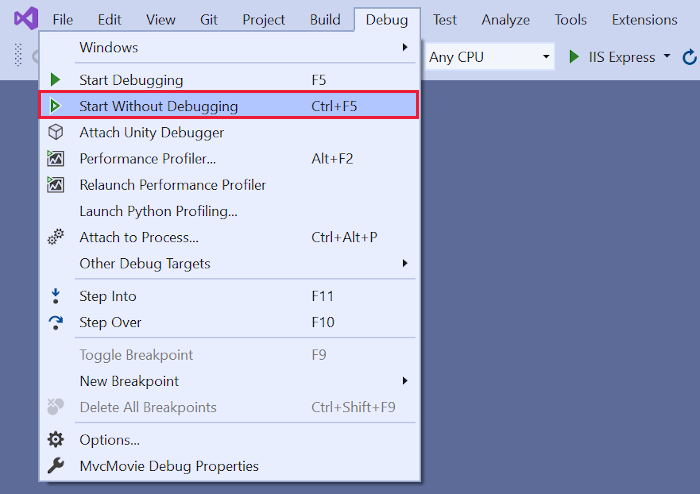
* + Starts [IIS Express](https://docs.microsoft.com/en-us/iis/extensions/introduction-to-iis-express/iis-express-overview).
  + Runs the app.

The address bar shows localhost:port# and not something like example.com. The standard hostname for your local computer is localhost. When Visual Studio creates a web project, a random port is used for the web server.

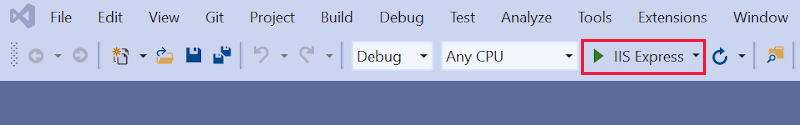
Launching the app without debugging by selecting Ctrl+F5 allows you to:

* Make code changes.
* Save the file.
* Quickly refresh the browser and see the code changes.

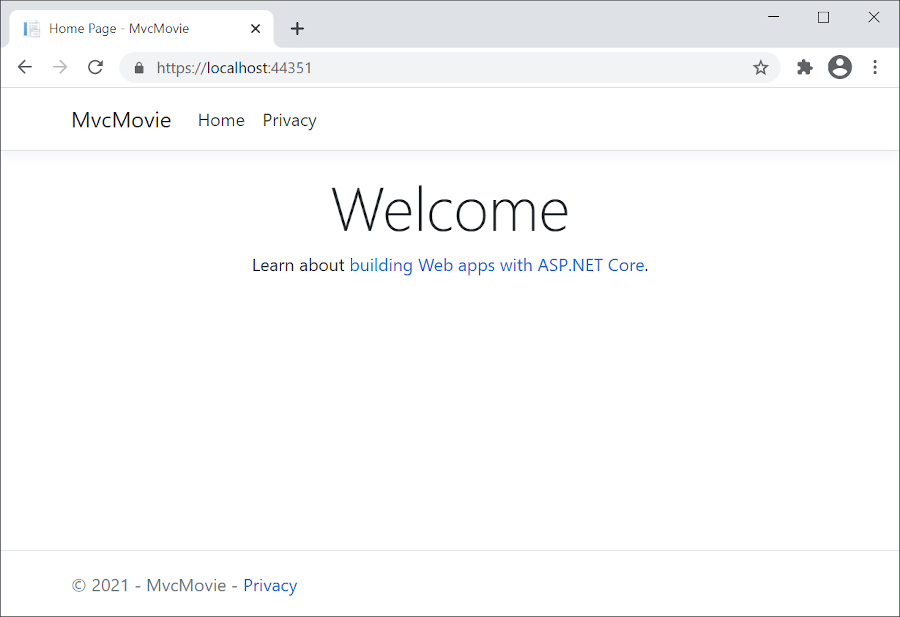
You can launch the app in debug or non-debug mode from the **Debug** menu item:



You can debug the app by selecting the **IIS Express** button



The following image shows the app:



# Part 2, add a controller to an ASP.NET Core MVC app

https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-view?view=aspnetcore-5.0&tabs=visual-studio

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps.

MVC-based apps contain:

* **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a Movie model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
* **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
* **C**ontrollers: Classes that:
  + Handle browser requests.
  + Retrieve model data.
  + Call view templates that return a response.

In an MVC app, the view only displays information. The controller handles and responds to user input and interaction. For example, the controller handles URL segments and query-string values, and passes these values to the model. The model might use these values to query the database. For example:

* https://localhost:5001/Home/Privacy: specifies the Home controller and the Privacy action.
* https://localhost:5001/Movies/Edit/5: is a request to edit the movie with ID=5 using the Movies controller and the Edit action, which are detailed later in the tutorial.

Route data is explained later in the tutorial.

The MVC architectural pattern separates an app into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns: The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps manage complexity when building an app, because it enables work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

These concepts are introduced and demonstrated in this tutorial series while building a movie app. The MVC project contains folders for the Controllers and Views.

## Add a controller

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-controller?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_2_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-controller?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_2_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-controller?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_2_visual-studio-mac)

In the **Solution Explorer**, right-click **Controllers > Add > Controller**.

In the **Add Scaffold** dialog box, select **MVC Controller - Empty**.

In the **Add New Item - MvcMovie dialog**, enter **HelloWorldController.cs** and select **Add**.

Replace the contents of Controllers/HelloWorldController.cs with the following:

C#Copy

using Microsoft.AspNetCore.Mvc;

using System.Text.Encodings.Web;

namespace MvcMovie.Controllers

{

public class HelloWorldController : Controller

{

//

// GET: /HelloWorld/

public string Index()

{

return "This is my default action...";

}

//

// GET: /HelloWorld/Welcome/

public string Welcome()

{

return "This is the Welcome action method...";

}

}

}

Every public method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint:

* Is a targetable URL in the web application, such as https://localhost:5001/HelloWorld.
* Combines:
  + The protocol used: HTTPS.
  + The network location of the web server, including the TCP port: localhost:5001.
  + The target URI: HelloWorld.

The first comment states this is an [HTTP GET](https://developer.mozilla.org/docs/Web/HTTP/Methods/GET) method that's invoked by appending /HelloWorld/ to the base URL.

The second comment specifies an [HTTP GET](https://developer.mozilla.org/docs/Web/HTTP/Methods) method that's invoked by appending /HelloWorld/Welcome/ to the URL. Later on in the tutorial, the scaffolding engine is used to generate HTTP POST methods, which update data.

Run the app without the debugger.

Append "HelloWorld" to the path in the address bar. The Index method returns a string.

MVC invokes controller classes, and the action methods within them, depending on the incoming URL. The default [URL routing logic](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-5.0) used by MVC, uses a format like this to determine what code to invoke:

/[Controller]/[ActionName]/[Parameters]

The routing format is set in the Configure method in Startup.cs file.

C#Copy

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

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

});

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above. In the preceding URL segments:

* The first URL segment determines the controller class to run. So localhost:5001/HelloWorld maps to the **HelloWorld**Controller class.
* The second part of the URL segment determines the action method on the class. So localhost:5001/HelloWorld/Index causes the Index method of the HelloWorldController class to run. Notice that you only had to browse to localhost:5001/HelloWorld and the Index method was called by default. Index is the default method that will be called on a controller if a method name isn't explicitly specified.
* The third part of the URL segment ( id) is for route data. Route data is explained later in the tutorial.

Browse to: https://localhost:{PORT}/HelloWorld/Welcome. Replace {PORT} with your port number.

The Welcome method runs and returns the string This is the Welcome action method.... For this URL, the controller is HelloWorld and Welcome is the action method. You haven't used the [Parameters] part of the URL yet.

Modify the code to pass some parameter information from the URL to the controller. For example, /HelloWorld/Welcome?name=Rick&numtimes=4.

Change the Welcome method to include two parameters as shown in the following code.

C#Copy

// GET: /HelloWorld/Welcome/

// Requires using System.Text.Encodings.Web;

public string Welcome(string name, int numTimes = 1)

{

return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");

}

The preceding code:

* Uses the C# optional-parameter feature to indicate that the numTimes parameter defaults to 1 if no value is passed for that parameter.
* Uses HtmlEncoder.Default.Encode to protect the app from malicious input, such as through JavaScript.
* Uses [Interpolated Strings](https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/interpolated-strings) in $"Hello {name}, NumTimes is: {numTimes}".

Run the app and browse to: https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4. Replace {PORT} with your port number.

Try different values for name and numtimes in the URL. The MVC [model binding](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-5.0) system automatically maps the named parameters from the query string to parameters in the method. See [Model Binding](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-5.0) for more information.

In the previous image:

* The URL segment Parameters isn't used.
* The name and numTimes parameters are passed in the [query string](https://wikipedia.org/wiki/Query_string).
* The ? (question mark) in the above URL is a separator, and the query string follows.
* The & character separates field-value pairs.

Replace the Welcome method with the following code:

C#Copy

public string Welcome(string name, int ID = 1)

{

return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");

}

Run the app and enter the following URL: https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick

In the preceding URL:

* The third URL segment matched the route parameter id.
* The Welcome method contains a parameter id that matched the URL template in the MapControllerRoute method.
* The trailing ? starts the [query string](https://wikipedia.org/wiki/Query_string).

C#Copy

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

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

});

In the preceding example:

* The third URL segment matched the route parameter id.
* The Welcome method contains a parameter id that matched the URL template in the MapControllerRoute method.
* The trailing ? (in id?) indicates the id parameter is optional.

# Part 4, add a model to an ASP.NET Core MVC app

https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio

In this section, classes are added for managing movies in a database. These classes are the "**M**odel" part of the **M**VC app.

These model classes are used with [Entity Framework Core](https://docs.microsoft.com/en-us/ef/core) (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes created are known as ***POCO*** classes, from **P**lain **O**ld **C**LR **O**bjects. POCO classes don't have any dependency on EF Core. They only define the properties of the data to be stored in the database.

In this tutorial, model classes are created first, and EF Core creates the database.

## Add a data model class

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_8_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_8_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_8_visual-studio-mac)

Right-click the Models folder > **Add** > **Class**. Name the file Movie.cs.

Update the Models/Movie.cs file with the following code:

C#Copy

using System;

using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models

{

public class Movie

{

public int Id { get; set; }

public string Title { get; set; }

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

public string Genre { get; set; }

public decimal Price { get; set; }

}

}

The Movie class contains an Id field, which is required by the database for the primary key.

The [DataType](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations.datatype) attribute on ReleaseDate specifies the type of the data (Date). With this attribute:

* The user isn't required to enter time information in the date field.
* Only the date is displayed, not time information.

[DataAnnotations](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations) are covered in a later tutorial.

## Add NuGet packages

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_9_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_9_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_9_visual-studio-mac)

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).

In the PMC, run the following command:

PowerShellCopy

Install-Package Microsoft.EntityFrameworkCore.Design

The preceding commands add:

* The EF Core SQL Server provider. The provider package installs the EF Core package as a dependency.
* The utilities used by the packages installed automatically in the scaffolding step, later in the tutorial.

Build the project as a check for compiler errors.

## Scaffold movie pages

Use the scaffolding tool to produce Create, Read, Update, and Delete (CRUD) pages for the movie model.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_10_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_10_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_10_visual-studio-mac)

In **Solution Explorer**, right-click the Controllers folder and select **Add > New Scaffolded Item**.

In the **Add Scaffold** dialog, select **MVC Controller with views, using Entity Framework > Add**.

Complete the **Add MVC Controller with views, using Entity Framework** dialog:

* In the **Model class** drop down, select **Movie (MvcMovie.Models)**.
* In the **Data context class** row, select the **+** (plus) sign.
  + In the **Add Data Context** dialog, the class name MvcMovie.Data.MvcMovieContext is generated.
  + Select **Add**.
* **Views** and **Controller name**: Keep the default.
* Select **Add**.

Scaffolding updates the following:

* Inserts required package references in the MvcMovie.csproj project file.
* Registers the database context in Startup.ConfigureServices of the Startup.cs file.
* Adds a database connection string to the appsettings.json file.

Scaffolding creates the following:

* A movies controller: Controllers/MoviesController.cs
* Razor view files for **Create**, **Delete**, **Details**, **Edit**, and **Index** pages: Views/Movies/\*.cshtml
* A database context class: Data/MvcMovieContext.cs

The automatic creation of these files and file updates are known as scaffolding.

The scaffolded pages can't be used yet because the database doesn't exist. Running the app and selecting the **Movie App** link results in a Cannot open database or no such table: Movie error message.

## Initial migration

Use the EF Core [Migrations](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/migrations?view=aspnetcore-5.0) feature to create the database. Migrations are a set of tools that create and update a database to match the data model.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_11_visual-studio)
* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_11_visual-studio-code+visual-studio-mac)

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** .

In the Package Manager Console (PMC), enter the following commands:

PowerShellCopy

Add-Migration InitialCreate

Update-Database

* Add-Migration InitialCreate: Generates a Migrations/{timestamp}\_InitialCreate.cs migration file. The InitialCreate argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the MvcMovieContext class.
* Update-Database: Updates the database to the latest migration, which the previous command created. This command runs the Up method in the Migrations/{time-stamp}\_InitialCreate.cs file, which creates the database.

The Update-Database command generates the following warning:

No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'.

Ignore the preceding warning, it's fixed in a later tutorial.

For more information on the PMC tools for EF Core, see [EF Core tools reference - PMC in Visual Studio](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/powershell).

## Test the app

Run the app and select the **Movie App** link.

If you get an exception similar to the following, you may have missed the [migrations step](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#migration):

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_12_visual-studio)
* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_12_visual-studio-code+visual-studio-mac)

ConsoleCopy

SqlException: Cannot open database "MvcMovieContext-1" requested by the login. The login failed.

**Note**

You may not be able to enter decimal commas in the Price field. To support [**jQuery validation**](https://jqueryvalidation.org/) for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see [**this GitHub issue**](https://github.com/dotnet/AspNetCore.Docs/issues/4076#issuecomment-326590420).

### Examine the generated database context class and registration

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. The database context is derived from [Microsoft.EntityFrameworkCore.DbContext](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontext) and specifies the entities to include in the data model.

Scaffolding creates the Data/MvcMovieContext.cs database context class:

C#Copy

using Microsoft.EntityFrameworkCore;

using MvcMovie.Models;

namespace MvcMovie.Data

{

public class MvcMovieContext : DbContext

{

public MvcMovieContext (DbContextOptions<MvcMovieContext> options)

: base(options)

{

}

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

}

}

The preceding code creates a [DbSet<Movie>](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbset-1) property that represents the movies in the database.

ASP.NET Core is built with [dependency injection (DI)](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0). Services, such as the database context, must be registered with DI in Startup. Components that require these services are provided via constructor parameters.

In the Controllers/MoviesController.cs file, the constructor uses [Dependency Injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0) to inject the MvcMovieContext database context into the controller. The database context is used in each of the [CRUD](https://wikipedia.org/wiki/Create,_read,_update_and_delete) methods in the controller.

Scaffolding generated the following highlighted code in Startup.ConfigureServices:

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_13_visual-studio)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_13_visual-studio-code)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_13_visual-studio-mac)

C#Copy

public void ConfigureServices(IServiceCollection services)

{

services.AddControllersWithViews();

services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));

}

The [ASP.NET Core configuration system](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-5.0) reads the "MvcMovieContext" database connection string.

### Examine the generated database connection string

Scaffolding added a connection string to the appsettings.json file:

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_14_visual-studio)
* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#tabpanel_14_visual-studio-code+visual-studio-mac)

JSONCopy

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-1;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

For local development, the [ASP.NET Core configuration system](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-5.0) reads the ConnectionString key from the appsettings.json file.

### The InitialCreate class

Examine the Migrations/{timestamp}\_InitialCreate.cs migration file:

C#Copy

public partial class InitialCreate : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "Movie",

columns: table => new

{

Id = table.Column<int>(type: "int", nullable: false)

.Annotation("SqlServer:Identity", "1, 1"),

Title = table.Column<string>(type: "nvarchar(max)", nullable: true),

ReleaseDate = table.Column<DateTime>(type: "datetime2", nullable: false),

Genre = table.Column<string>(type: "nvarchar(max)", nullable: true),

Price = table.Column<decimal>(type: "decimal(18,2)", nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_Movie", x => x.Id);

});

}

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropTable(

name: "Movie");

}

}

In the preceding code:

* InitialCreate.Up creates the Movie table and configures Id as the primary key.
* InitialCreate.Down reverts the schema changes made by the Up migration.

## Dependency injection in the controller

Open the Controllers/MoviesController.cs file and examine the constructor:

C#Copy

public class MoviesController : Controller

{

private readonly MvcMovieContext \_context;

public MoviesController(MvcMovieContext context)

{

\_context = context;

}

The constructor uses [Dependency Injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0) to inject the database context (MvcMovieContext) into the controller. The database context is used in each of the [CRUD](https://wikipedia.org/wiki/Create,_read,_update_and_delete) methods in the controller.

Test the **Create** page. Enter and submit data.

Test the **Edit**, **Details**, and **Delete** pages.

## Strongly typed models and the @model directive

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the ViewData dictionary. The ViewData dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables compile time code checking. The scaffolding mechanism passed a strongly typed model in the MoviesController class and views.

Examine the generated Details method in the Controllers/MoviesController.cs file:

C#Copy

// GET: Movies/Details/5

public async Task<IActionResult> Details(int? id)

{

if (id == null)

{

return NotFound();

}

var movie = await \_context.Movie

.FirstOrDefaultAsync(m => m.Id == id);

if (movie == null)

{

return NotFound();

}

return View(movie);

}

The id parameter is generally passed as route data. For example, https://localhost:5001/movies/details/1 sets:

* The controller to the movies controller, the first URL segment.
* The action to details, the second URL segment.
* The id to 1, the last URL segment.

The id can be passed in with a query string, as in the following example:

https://localhost:5001/movies/details?id=1

The id parameter is defined as a [nullable type](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/nullable-types/index) (int?) in cases when the id value isn't provided.

A [lambda expression](https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/statements-expressions-operators/lambda-expressions) is passed in to the [FirstOrDefaultAsync](https://docs.microsoft.com/en-us/dotnet/api/system.data.entity.queryableextensions.firstordefaultasync) method to select movie entities that match the route data or query string value.

C#Copy

var movie = await \_context.Movie

.FirstOrDefaultAsync(m => m.Id == id);

If a movie is found, an instance of the Movie model is passed to the Details view:

C#Copy

return View(movie);

Examine the contents of the Views/Movies/Details.cshtml file:

CSHTMLCopy

@model MvcMovie.Models.Movie

@{

ViewData["Title"] = "Details";

}

<h1>Details</h1>

<div>

<h4>Movie</h4>

<hr />

<dl class="row">

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.Title)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Title)

</dd>

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.ReleaseDate)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.ReleaseDate)

</dd>

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.Genre)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Genre)

</dd>

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.Price)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Price)

</dd>

</dl>

</div>

<div>

<a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |

<a asp-action="Index">Back to List</a>

</div>

The @model statement at the top of the view file specifies the type of object that the view expects. When the movie controller was created, the following @model statement was included:

CSHTMLCopy

@model MvcMovie.Models.Movie

This @model directive allows access to the movie that the controller passed to the view. The Model object is strongly typed. For example, in the Details.cshtml view, the code passes each movie field to the DisplayNameFor and DisplayFor HTML Helpers with the strongly typed Model object. The Create and Edit methods and views also pass a Movie model object.

Examine the Index.cshtml view and the Index method in the Movies controller. Notice how the code creates a List object when it calls the View method. The code passes this Movies list from the Index action method to the view:

C#Copy

// GET: Movies

public async Task<IActionResult> Index()

{

return View(await \_context.Movie.ToListAsync());

}

When the movies controller was created, scaffolding included the following @model statement at the top of the Index.cshtml file:

CSHTMLCopy

@model IEnumerable<MvcMovie.Models.Movie>

The @model directive allows access to the list of movies that the controller passed to the view by using a Model object that's strongly typed. For example, in the Index.cshtml view, the code loops through the movies with a foreach statement over the strongly typed Model object:

CSHTMLCopy

@model IEnumerable<MvcMovie.Models.Movie>

@{

ViewData["Title"] = "Index";

}

<h1>Index</h1>

<p>

<a asp-action="Create">Create New</a>

</p>

<table class="table">

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.Title)

</th>

<th>

@Html.DisplayNameFor(model => model.ReleaseDate)

</th>

<th>

@Html.DisplayNameFor(model => model.Genre)

</th>

<th>

@Html.DisplayNameFor(model => model.Price)

</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.Title)

</td>

<td>

@Html.DisplayFor(modelItem => item.ReleaseDate)

</td>

<td>

@Html.DisplayFor(modelItem => item.Genre)

</td>

<td>

@Html.DisplayFor(modelItem => item.Price)

</td>

<td>

<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |

<a asp-action="Details" asp-route-id="@item.Id">Details</a> |

<a asp-action="Delete" asp-route-id="@item.Id">Delete</a>

</td>

</tr>

}

</tbody>

</table>

Because the Model object is strongly typed as an IEnumerable<Movie> object, each item in the loop is typed as Movie. Among other benefits, the compiler validates the types used in the code.

## SQL Logging of Entity Framework Core

Logging configuration is commonly provided by the Logging section of appsettings.{Environment}.json files. To log SQL statements, add "Microsoft.EntityFrameworkCore.Database.Command": "Information" to the appsettings.Development.json file:

JSONCopy

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyDB-2;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

,"Microsoft.EntityFrameworkCore.Database.Command": "Information"

}

},

"AllowedHosts": "\*"

}

With the preceding JSON, SQL statements are displayed on the command line and in the Visual Studio output window.

# **Code First Approach With CRUD Operation In Entity Framework In MVC - Part Four**

**https://www.c-sharpcorner.com/article/code-first-approach-with-crud-operation-in-entity-framework-in-mvc-part-four/**

In the previous article, I have explained how to implement a code first approach when we have an existing database. Now, I will explain how to implement a code first approach when we don’t have an existing database.  Let‘s see step by step how to implement it.

See the previous articles for basic details.

* [Entity Framework in MVC - Part One](https://www.c-sharpcorner.com/article/entity-framework-in-mvc-part-one/)
* [Entity Framework in MVC - Part Two](https://www.c-sharpcorner.com/article/entity-framework-in-mvc-part-two/)
* [Entity Framework in MVC - Part Three](https://www.c-sharpcorner.com/article/entity-framework-in-mvc-part-three/)

**Step 1**

Open Visual Studio >> File >> Project >> Web application >> choose MVC >> OK.

After opening the project, first, we have to create a domain class.

Right-click on Models folder and add a class like below. I am adding an Employee class.

1. **public** **class** Employee
2. {
3. **public** **int** EmpId { get; set; }
4. **public** string Name { get; set; }
5. **public** string Address { get; set; }
6. **public** string Email { get; set; }
7. **public** string MobileNo { get; set; }
8. }

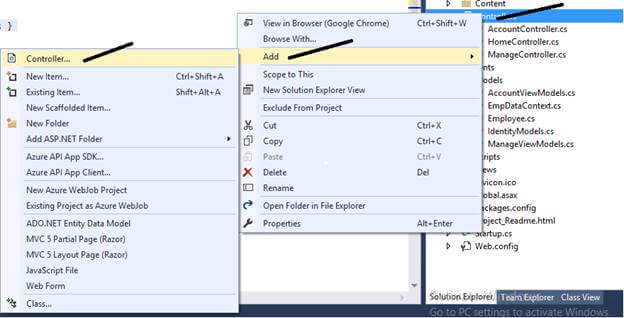
**Step 2**

Now, I am going to add a DbContext class. For that, right-click the Models folder and add a class and give the name EmpDataContext:DbContext.

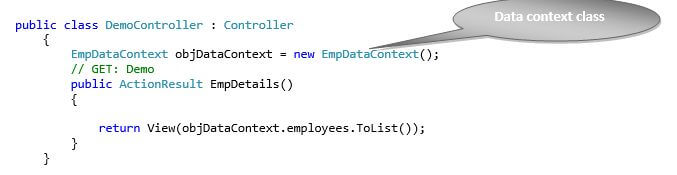
1. **public** **class** EmpDataContext : DbContext
2. {
3. **public** DbSet<Employee> employees { get; set; }
4. }

**Step 3**

Now, we have to add a Controller. Go to Controllers folder and add a controller.

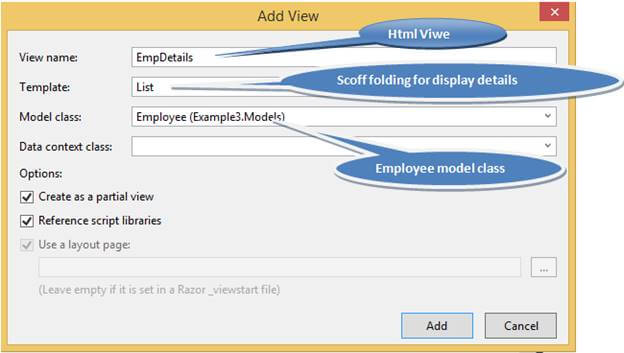


When we add a controller, in this controller an Index Action method is automatically created. We can change it to a user-friendly method name. Now, create the object of your Dbcontext class and write the logic for retrieval of the data.



1. **public** **class** DemoController : Controller
2. {
3. EmpDataContext objDataContext = **new** EmpDataContext();
4. // GET: Demo
5. **public** ActionResult EmpDetails()
6. {
8. **return** View(objDataContext.employees.ToList());
9. }
10. }

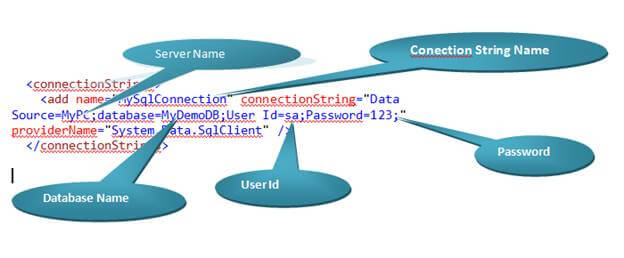
After that, add a View for displaying the employee records. Right-click on action method and add a View.



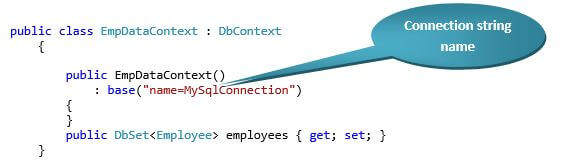
**Step 4**

Now, before the retrieval of data, we have to set our connection string in web.confifile.

1. <connectionStrings>
2. <add name="MySqlConnection" connectionString="Data Source=MANNU;database=MyDemoDB;User Id=sa;Password=123;" providerName="System.Data.SqlClient" />
3. </connectionStrings>



Now, set the connection string in DbContext class. If we don’t want to give manual connection string, then when running the project, it will automatically create the connection string with the same name as the class name of DbContext class. If we create manually, then we have to pass our Connection String name in base class parameter.



1. **public** **class** EmpDataContext : DbContext
2. {
4. **public** EmpDataContext()
5. : base("name=MySqlConnection")
6. {
7. }
8. **public** DbSet<Employee> employees { get; set; }
9. }

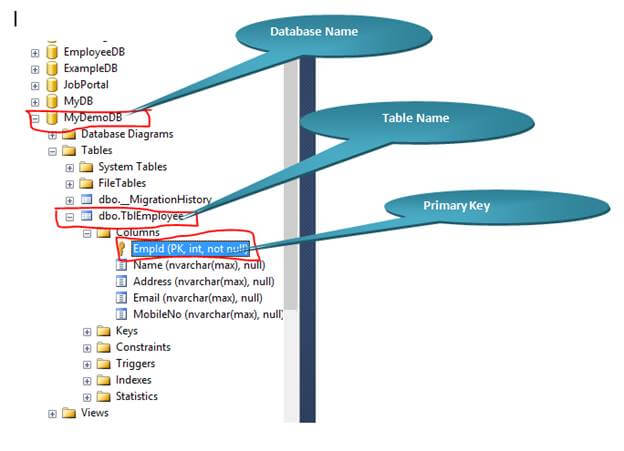
As we know, in this Code First approach when we will run the project, at that moment automatically, it will create the DataBase and the Table; table name will be same as our domain Class name; as below my class name is Employee. If we want to change the table name in a user-friendly name as “ TblEmployee”,  then we can use Table class attribute. Also, we can set the Primary key in SQL table from our program by using the Key class attribute in “[ ]” bracteates.

For using this functionality, we have to import the namespace like below.

1. using System.ComponentModel.DataAnnotations;           (Import namespace)
2. using System.ComponentModel.DataAnnotations.Schema;
4. [Table("TblEmployee")]
5. **public** **class** Employee
6. {
7. [Key]
8. **public** **int** EmpId { get; set; }
9. **public** string Name { get; set; }
10. **public** string Address { get; set; }
11. **public** string Email { get; set; }
12. **public** string MobileNo { get; set; }
13. }

**Step 5**

Now, we will**r**un the project and then check in SQL.



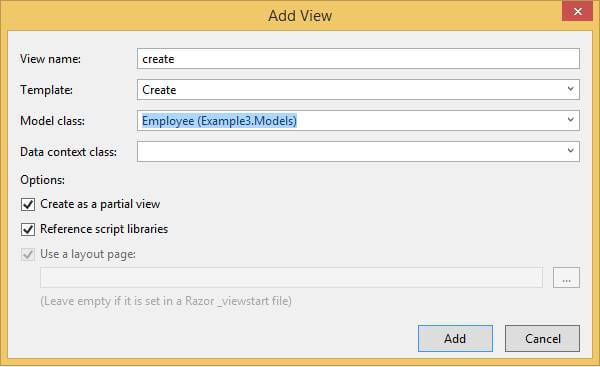
**Step 6**

Now, we will complete our CRUD operation using the Code First approach. So for that, we will add one Controller.

First, we will write the code for Create operation. For this, we have to create the action method for getting the request named as Create method.

1. **public** ActionResult create()
2. {
4. **return** View();
5. }

Now, right-click on Create Action Method and add View.



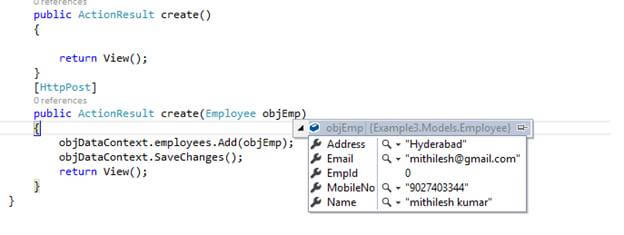
And, after that, we will write HTML code like below.

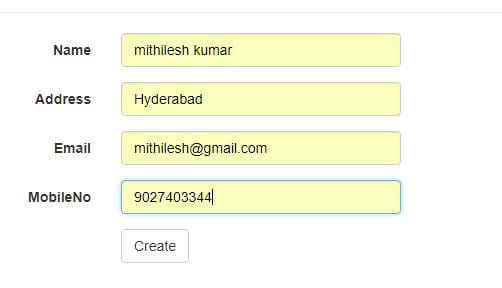
1. @model Example3.Models.Employee
2. @using (Html.BeginForm())
3. {
4. <div **class**="form-horizontal">
5. <h4>Employee</h4>
6. <hr />
8. <div **class**="form-group">
9. @Html.LabelFor(model => model.Name, htmlAttributes: **new** { @**class** = "control-label col-md-2" })
10. <div **class**="col-md-10">
11. @Html.EditorFor(model => model.Name, **new** { htmlAttributes = **new** { @**class** = "form-control" } })
13. </div>
14. </div>
16. <div **class**="form-group">
17. @Html.LabelFor(model => model.Address, htmlAttributes: **new** { @**class** = "control-label col-md-2" })
18. <div **class**="col-md-10">
19. @Html.EditorFor(model => model.Address, **new** { htmlAttributes = **new** { @**class** = "form-control" } })
21. </div>
22. </div>
24. <div **class**="form-group">
25. @Html.LabelFor(model => model.Email, htmlAttributes: **new** { @**class** = "control-label col-md-2" })
26. <div **class**="col-md-10">
27. @Html.EditorFor(model => model.Email, **new** { htmlAttributes = **new** { @**class** = "form-control" } })
29. </div>
30. </div>
32. <div **class**="form-group">
33. @Html.LabelFor(model => model.MobileNo, htmlAttributes: **new** { @**class** = "control-label col-md-2" })
34. <div **class**="col-md-10">
35. @Html.EditorFor(model => model.MobileNo, **new** { htmlAttributes = **new** { @**class** = "form-control" } })
37. </div>
38. </div>
40. <div **class**="form-group">
41. <div **class**="col-md-offset-2 col-md-10">
42. <input type="submit" value="Create" **class**="btn btn-default" />
43. </div>
44. </div>
45. </div>
46. }

Now, we will write the code for posting the details.

1. [HttpPost]
2. **public** ActionResult create(Employee objEmp)
3. {
4. objDataContext.employees.Add(objEmp);
5. objDataContext.SaveChanges();
6. **return** View();
7. }

Now, run the project and insert the data.





After that, click on the Create button. Then we will write the code for retrieving the data.

1. EmpDataContext objDataContext = **new** EmpDataContext();
2. // GET: Demo
3. **public** ActionResult EmpDetails()
4. {
6. **return** View(objDataContext.employees.ToList());
7. }

Here, right click on the Action method “EmpDetails” and press Add View option. And write below html code,

1. @model IEnumerable<Example3.Models.Employee>
3. <p>
4. @Html.ActionLink("Create New", "Create")
5. </p>
6. <table **class**="table">
7. <tr **class**="btn-primary">
8. <th>
9. @Html.DisplayNameFor(model => model.EmpId)
10. </th>
11. <th>
12. @Html.DisplayNameFor(model => model.Name)
13. </th>
14. <th>
15. @Html.DisplayNameFor(model => model.Address)
16. </th>
17. <th>
18. @Html.DisplayNameFor(model => model.Email)
19. </th>
20. <th>
21. @Html.DisplayNameFor(model => model.MobileNo)
22. </th>
23. <th>Action</th>
24. </tr>
26. @foreach (**var** item **in** Model) {
27. <tr **class**="btn-info">
28. <td>
29. @Html.DisplayFor(modelItem => item.EmpId)
30. </td>
31. <td>
32. @Html.DisplayFor(modelItem => item.Name)
33. </td>
34. <td>
35. @Html.DisplayFor(modelItem => item.Address)
36. </td>
37. <td>
38. @Html.DisplayFor(modelItem => item.Email)
39. </td>
40. <td>
41. @Html.DisplayFor(modelItem => item.MobileNo)
42. </td>
43. <td>
44. @Html.ActionLink("Edit", "Edit", **new** {  id=item.EmpId }) |
45. @Html.ActionLink("Details", "Details", **new** { id = item.EmpId }) |
46. @Html.ActionLink("Delete", "Delete", **new** { id = item.EmpId })
47. </td>
48. </tr>
49. }
51. </table>

Now, we will see the output.

MVC

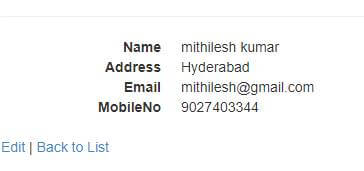
Now we will write code for seeing the details of a particular employee,

1. **public** ActionResult Details(string id)
2. {
3. **int** empId = Convert.ToInt32(id);
4. **var** emp = objDataContext.employees.Find(empId);
5. **return** View(emp);
6. }

Here, right click on the Action method “Details” and press Add View option. And write the below html code,

1. @model Example3.Models.Employee
3. <div>
4. <h4>Employee</h4>
5. <hr />
6. <dl **class**="dl-horizontal">
7. <dt>
8. @Html.DisplayNameFor(model => model.Name)
9. </dt>
11. <dd>
12. @Html.DisplayFor(model => model.Name)
13. </dd>
15. <dt>
16. @Html.DisplayNameFor(model => model.Address)
17. </dt>
19. <dd>
20. @Html.DisplayFor(model => model.Address)
21. </dd>
23. <dt>
24. @Html.DisplayNameFor(model => model.Email)
25. </dt>
27. <dd>
28. @Html.DisplayFor(model => model.Email)
29. </dd>
31. <dt>
32. @Html.DisplayNameFor(model => model.MobileNo)
33. </dt>
35. <dd>
36. @Html.DisplayFor(model => model.MobileNo)
37. </dd>
39. </dl>
40. </div>
41. <p>
42. @Html.ActionLink("Edit", "Edit", **new** { id = Model.EmpId }) |
43. @Html.ActionLink("Back to List", "EmpDetails")
44. </p>

**Output**

****

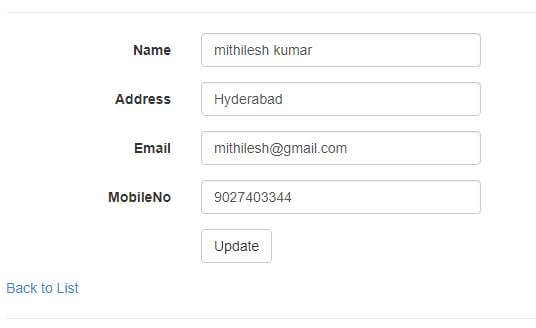
Now, we will perform edit functionality so for this we will create edit action method.

1. **public** ActionResult Edit(string id)
2. {
3. **int** empId = Convert.ToInt32(id);
4. **var** emp = objDataContext.employees.Find(empId);
5. **return** View(emp);
6. }

Now, right click on the Action method “Edit” and press Add View option. And write the below html code,

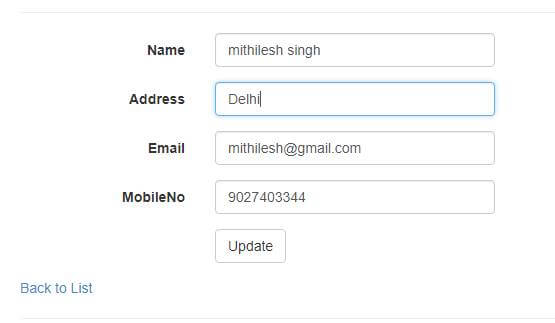
1. @model Example3.Models.Employee
3. @using (Html.BeginForm())
4. {
6. <div **class**="form-horizontal">
7. <h4>Employee</h4>
8. <hr />
10. @Html.HiddenFor(model => model.EmpId)
12. <div **class**="form-group">
13. @Html.LabelFor(model => model.Name, htmlAttributes: **new** { @**class** = "control-label col-md-2" })
14. <div **class**="col-md-10">
15. @Html.EditorFor(model => model.Name, **new** { htmlAttributes = **new** { @**class** = "form-control" } })
17. </div>
18. </div>
20. <div **class**="form-group">
21. @Html.LabelFor(model => model.Address, htmlAttributes: **new** { @**class** = "control-label col-md-2" })
22. <div **class**="col-md-10">
23. @Html.EditorFor(model => model.Address, **new** { htmlAttributes = **new** { @**class** = "form-control" } })
25. </div>
26. </div>
28. <div **class**="form-group">
29. @Html.LabelFor(model => model.Email, htmlAttributes: **new** { @**class** = "control-label col-md-2" })
30. <div **class**="col-md-10">
31. @Html.EditorFor(model => model.Email, **new** { htmlAttributes = **new** { @**class** = "form-control" } })
33. </div>
34. </div>
36. <div **class**="form-group">
37. @Html.LabelFor(model => model.MobileNo, htmlAttributes: **new** { @**class** = "control-label col-md-2" })
38. <div **class**="col-md-10">
39. @Html.EditorFor(model => model.MobileNo, **new** { htmlAttributes = **new** { @**class** = "form-control" } })
41. </div>
42. </div>
44. <div **class**="form-group">
45. <div **class**="col-md-offset-2 col-md-10">
46. <input type="submit" value="Save" **class**="btn btn-default" />
47. </div>
48. </div>
49. </div>
50. }
52. <div>
53. @Html.ActionLink("Back to List", "EmpDetails")
54. </div>

Now , we will check the output .



Now, we will write the code for updating the page.

1. [HttpPost]
2. **public** ActionResult Edit(Employee objEmp)
3. {
4. **var** data =  objDataContext.employees.Find(objEmp.EmpId);
5. **if**(data != **null**)
6. {
7. data.Name = objEmp.Name;
8. data.Address = objEmp.Address;
9. data.Email = objEmp.Email;
10. data.MobileNo = objEmp.MobileNo;
11. }
12. objDataContext.SaveChanges();
13. **return** View();
14. }



After clicking on the Update button the data will be updated.

MVC

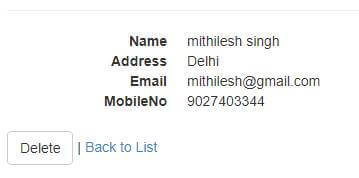
So, finally we write code for Delete operation.

1. **public** ActionResult Delete(string id)
2. {
3. **int** empId = Convert.ToInt32(id);
4. **var** emp = objDataContext.employees.Find(empId);
5. **return** View(emp);
6. }

Right click on the Delete Action method and add the view. And we will write the HTML code.

1. @model Example3.Models.Employee
3. <h3>Are you sure you want to **delete** **this**?</h3>
4. <div>
5. <h4>Employee</h4>
6. <hr />
7. @using (Html.BeginForm())
8. {
9. @Html.AntiForgeryToken()
10. <dl **class**="dl-horizontal">
11. @Html.HiddenFor(model => model.EmpId)
12. <dt>
13. @Html.DisplayNameFor(model => model.Name)
14. </dt>
16. <dd>
17. @Html.DisplayFor(model => model.Name)
18. </dd>
20. <dt>
21. @Html.DisplayNameFor(model => model.Address)
22. </dt>
24. <dd>
25. @Html.DisplayFor(model => model.Address)
26. </dd>
28. <dt>
29. @Html.DisplayNameFor(model => model.Email)
30. </dt>
32. <dd>
33. @Html.DisplayFor(model => model.Email)
34. </dd>
36. <dt>
37. @Html.DisplayNameFor(model => model.MobileNo)
38. </dt>
40. <dd>
41. @Html.DisplayFor(model => model.MobileNo)
42. </dd>
44. </dl>
46. <div **class**="form-actions no-color">
47. <input type="submit" value="Delete" **class**="btn btn-default" /> |
48. @Html.ActionLink("Back to List", "EmpDetails")
49. </div>
50. }
51. </div>

We will see the Output.



Here, when click the Delete button then the Data will be deleted.

MVC

So, here no data is available because we had taken one value.

**Summary**

Finally, we knew how to perform Code First approach, if the DataBase does not exist with CRUD operation example.

In our next article we will see “How to give relationships among multiple tables using Code First approach”.