**Asp.Net Core MVC CRUD with EF Core**

[Create ASP.NET Core MVC Project](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Create_ASPNET_Core_MVC_Project)

[Setup Database for EF Core](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Setup_Database_for_EF_Core)

[Inject DbContext Class with Dependency Injection.](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Inject_DbContext_Class_with_Dependency_Injection)

[Initiate DB Migration](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Initiate_DB_Migration)

[Create MVC Controller For CRUD Operations](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Create_MVC_Controller_For_CRUD_Operations)

[Retrieve List of Records in an MVC View](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Retrieve_List_of_Records_in_an_MVC_View)

[Add or Edit a Record](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Add_or_Edit_a_Record)

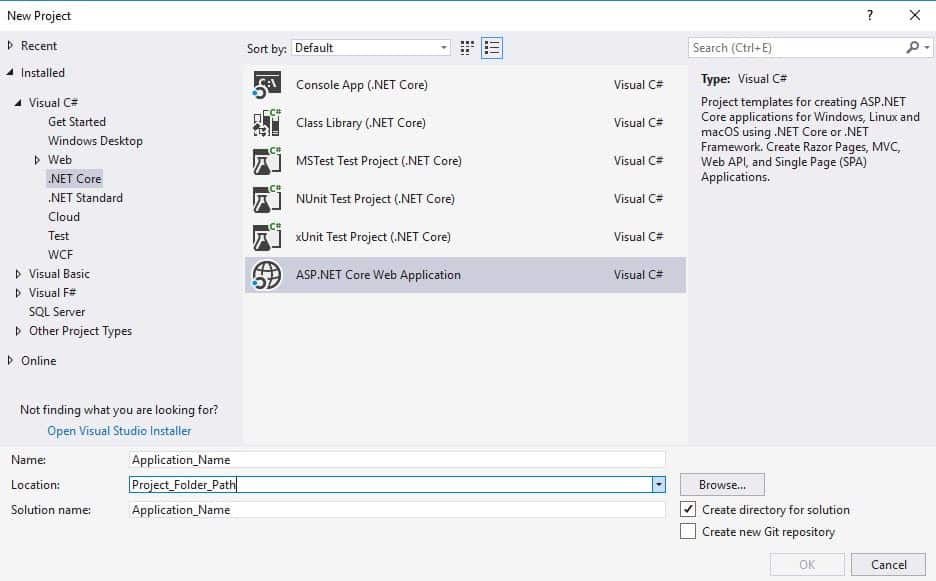
[Form Validation](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Form_Validation)

[Delete a Record](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Delete_a_Record)

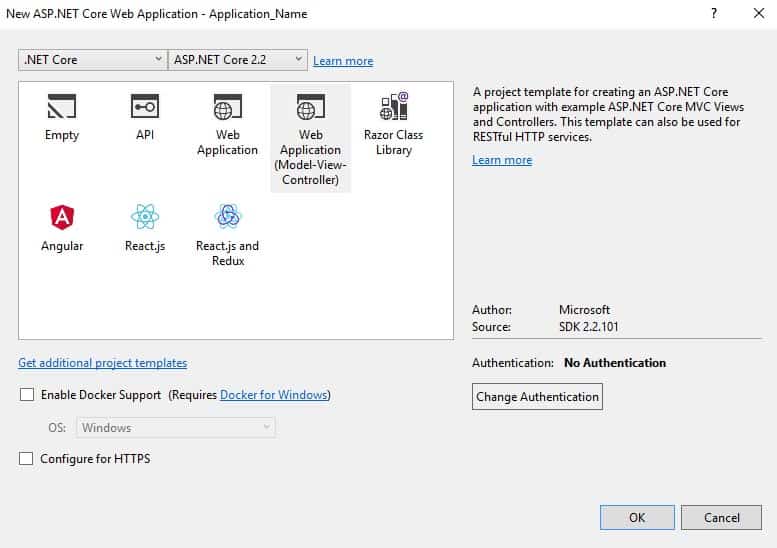
[Video Tutorial on this Topic](https://www.codaffection.com/asp-net-core-article/asp-net-core-mvc-crud-with-ef-core/#Video_Tutorial_on_this_Topic)

## Create ASP.NET Core MVC Project

From Visual Studio, Go to File > New > Project( Ctrl + Shift + N). Then select ASP.NET Core Web Application.

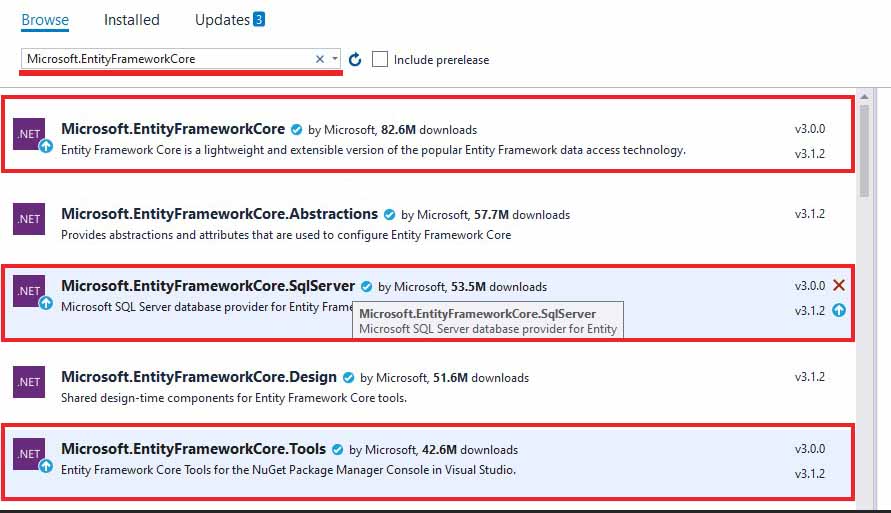


In template wizard, Select Web Application(MVC) template. Make sure to select the latest ASP.NET Core Version from top dropdown. Uncheck HTTPS option, which is not required in the development environment.



## Setup Database for EF Core

The database for the project will be created and managed through EF Core – Code First Approach. So first of all, we have to install corresponding NuGet Packages. Right-click on the project from solution explorer, select Manage NuGet Packages. From browse tab, search and install Microsoft.EntityFrameworkCore and its dependent packages.



Now let’s define a model class for employee entity in Models folder as Employee.

**//File : /Models/Employee.cs**

**public** **class** **Employee**

**{**

**[Key]**

**public** **int** EmployeeId **{** **get;** **set;** **}**

**[Column(TypeName ="nvarchar(250)")]**

**[Required(ErrorMessage ="This field is required.")]**

**[DisplayName("Full Name")]**

**public** **string** FullName **{** **get;** **set;** **}**

**[Column(TypeName = "varchar(10)")]**

**[DisplayName("Emp. Code")]**

**public** **string** EmpCode **{** **get;** **set;** **}**

**[Column(TypeName = "varchar(100)")]**

**public** **string** Position **{** **get;** **set;** **}**

**[Column(TypeName = "varchar(100)")]**

**[DisplayName("Office Location")]**

**public** **string** OfficeLocation **{** **get;** **set;** **}**

**}**

In Entity Framework, the actual physical entities in a database are created and managed from DBContext class. In our application, EmployeeContext class does the job. To create a table corresponding to the employee model class, We’ve added Employees property of the type DbSet.

**//File : /Models/EmployeeContext.cs**

**public** **class** **EmployeeContext:DbContext**

**{**

**public** **EmployeeContext(DbContextOptions<EmployeeContext>** options**):base(**options**)**

**{** **}**

**public** **DbSet<Employee>** Employees **{** **get;** **set;** **}**

**}**

## Inject DbContext Class with Dependency Injection.

In order to interact with database, we need an instance of the DbContext class. We can do that with dependency injection. For the class constructor parameter options, following informations are required.

* Database Provider – whether it is SQL Server or MySQL or PostgreSQL etc.
* DB Connection String

First of all, we will save the connection string in appsettings.json file as follows.

{

...,

"ConnectionStrings": {

"DevConnection": "Server=(local)\\sqlexpress;Database=EmployeeDB;Trusted\_Connection=True;MultipleActiveResultSets=True;"

}

}

JSON

Copy

Now let’s create the DbContext instance by passing both DB connection string and DB provider using dependency injection from Asp.Net Core. For that, we just need to update ConfigureServices method from Startup class. Invoke AddDbConext method from services collection as follows.

**public** **void** **ConfigureServices(IServiceCollection** services**)**

**{**

**...**

services**.AddDbContext<EmployeeContext>(**options **=>**

options**.UseSqlServer(**Configuration**.GetConnectionString("DevConnection")));**

**}**

How and when this dependency injection works? we’ll discuss that later, once we add an employee controller in this project.

## Initiate DB Migration

So far we have been modeling our Database, Let’s create the actual DB. First of all open Package Manager Console.for that you can right-click the project from Solution Explorer then go to Tools > NuGet Package Manager > Package Manager Console. Inside the console execute the following commands one by one.

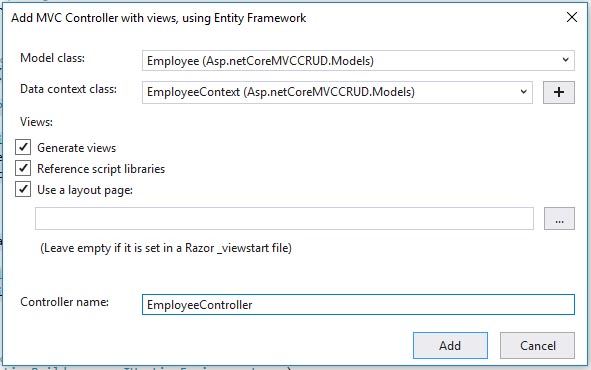
Add-Migration **"InitialCreate"**

Update-Database

Now you should have the EmployeeDB(mentioned in connection string) created as per our employee model.

## Create MVC Controller For CRUD Operations

Let’s create a new MVC controller EmployeeController. Go to Controller > Add > Controller, then select ‘MVC Controller with views, using Entity Framework’ template. Select Employee and EmployeeContext as Model and DbContext class respectively.



The created MVC controller will have all the action methods for all CRUD operations, even though I would like to simplify it as follows. Each of the action methods will be discussed later in this article.

**public** **class** **EmployeeController** **:** **Controller**

**{**

**private** **readonly** **EmployeeContext** \_context**;**

**//constructor**

**public** **EmployeeController(EmployeeContext** context**)**

**{**

\_context **=** context**;**

**}**

**// GET: Employee**

**public** **async** **Task<IActionResult>** **Index()**

**{** **...** **}**

**// GET: Employee/AddOrEdit**

**public** **IActionResult** **AddOrEdit(int** id **=** **0)**

**{** **...** **}**

**// POST: Employee/AddOrEdit**

**[**HttpPost**]**

**[ValidateAntiForgeryToken]**

**public** **async** **Task<IActionResult>** **AddOrEdit([Bind("EmployeeId,FullName,EmpCode,Position,OfficeLocation")]** **Employee** employee**)**

**{** **...** **}**

**// GET: Employee/Delete/5**

**public** **async** **Task<IActionResult>** **Delete(int?** id**)**

**{** **...** **}**

**}**

Now a few words on ASP.NET Core dependency injection: As you know, an instance of MVC controller will be created once a request is made into it. But there is a constructor parameter context of the type EmployeeContext. How can we pass value for the parameter when the controller instance is created by the ASP.NET Core framework itself? There comes the importance of dependency injection, that we configured above in Startup class, whenever a controller constructor needs an instance of DbContext, the dependency injection passes an instance. So the rest of the action methods inside the controller can interact with the Database through the injected DbContext instance.

## Retrieve List of Records in an MVC View

Purpose of each action methods in EmployeeController.

* Index: Retrieve list of records from the employees table.
* AddOrEdit: Handle both insert and update operation.
* Delete: Remove an employee record with a given employee id.

All of the above action methods either manipulate the data or retrieve existing data from the database. Now we’ve to design the user interface with razor views for showing retrieved data or a form/ button to submit data to action methods. You could see such razor views in /Views/Employee folder.

Now let’s retrieve the list of employees using Index action method.

**// GET: Employees**

**public** **async** **Task<IActionResult>** **Index()**

**{**

**return** **View(await** \_context**.**Employees**.ToListAsync());**

**}**

List of records from employees table can easily be retrieved with DbSet property Employees as shown above. The collection of records is returned with the function View. So there should be a razor view file index.cshtml (same name as that of index action method) to render the collection of records. We’ve done that with help of an HTML table using a foreach loop.

@model IEnumerable**<Asp.netCoreMVCCRUD.Models.Employee>**

@{

ViewData["Title"] = "Index";

}

**<h4>**Employee Register**</h4>**

**<hr />**

**<table class="table table-hover">**

**<thead>**

**<tr>**

**<th>**

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

**</th>**

**<th>**

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

**</th>**

**<th>**

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

**</th>**

**<th>**

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

**</th>**

**<th>**

**<a asp-action="AddOrEdit" class="btn btn-outline-success"><i class="far fa-plus-square"></i>** Employee**</a>**

**</th>**

**</tr>**

**</thead>**

**<tbody>**

@foreach (var item in Model) {

**<tr>**

**<td>**

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

**</td>**

**<td>**

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

**</td>**

**<td>**

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

**</td>**

**<td>**

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

**</td>**

**<td>**

**<a asp-action="AddOrEdit" asp-route-id="@item.EmployeeId"><i class="fa fa-marker fa-lg"></i></a>**

**<a asp-action="Delete" asp-route-id="@item.EmployeeId" class="text-danger ml-1" onclick="return confirm('Are you sure to delete this record?')"><i class="fa fa-trash-alt fa-lg"></i></a>**

**</td>**

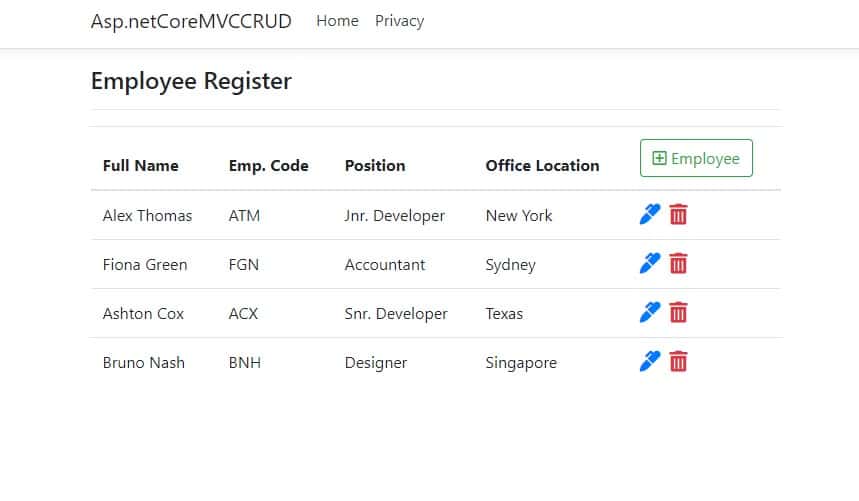
**</tr>**

}

**</tbody>**

**</table>**

New employee can be inserted with + button from last column header. The last column contains button for edit/ update and delete operation. Now the index view should look like this.



Did you see the navbar here? inside our razor view, we only said to show the HTML table, now where is this navbar coming from? It’s from our global layout view Shared/\_Layout.cshtml. Not only the navbar, the entire HTML declaration and surrounding body and head tag are defined inside this layout view. It will thereby default in any MVC project to enclose all of the views file. Stylesheet reference for Bootstrap and Font Awesome should be added to it. Most of the case Bootstrap will be there already, just add the Font Awesome stylesheet reference.

**<!-- Add following stylesheets to <head><meta charset="UTF-8"> tag -->**

**<link rel="preload" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.12.1/css/all.min.css" data-rocket-async="style" as="style" onload="this.onload=null;this.rel='stylesheet'" />**

## Add or Edit a Record

Normally by default there will be separate methods for insert and update operation. here in this project we’ve combined them into one. First of all, let’s define GET and POST methods for both insert and update operation with AddOrEdit action method.

**// GET: Employee/AddOrEdit**

**public** **IActionResult** **AddOrEdit(int** id **=** **0)**

**{**

**if** **(**id **==** **0)**

**return** **View(new** **Employee());**

**else**

**return** **View(**\_context**.**Employees**.Find(**id**));**

**}**

**// POST: Employee/AddOrEdit**

**[**HttpPost**]**

**[ValidateAntiForgeryToken]**

**public** **async** **Task<IActionResult>** **AddOrEdit([Bind("EmployeeId,FullName,EmpCode,Position,OfficeLocation")]** **Employee** employee**)**

**{**

**if** **(**ModelState**.**IsValid**)**

**{**

**if** **(**employee**.**EmployeeId **==** **0)**

\_context**.Add(**employee**);**

**else**

\_context**.Update(**employee**);**

**await** \_context**.SaveChangesAsync();**

**return** **RedirectToAction(nameof(**Index**));**

**}**

**return** **View(**employee**);**

**}**

And here is our AddOrEdit.cshtml view for the above GET action method.

@model Asp.netCoreMVCCRUD.Models.Employee

@{

ViewData["Title"] = "Create";

}

**<h4>**Employee Form**</h4>**

**<hr />**

**<div class="row">**

**<div class="col-md-6">**

**<form asp-action="AddOrEdit">**

**<div asp-validation-summary="ModelOnly" class="text-danger"></div>**

**<input type="hidden" asp-for="EmployeeId" />**

**<div class="form-group">**

**<label asp-for="FullName" class="control-label"></label>**

**<input asp-for="FullName" class="form-control" />**

**<span asp-validation-for="FullName" class="text-danger"></span>**

**</div>**

**<div class="form-row">**

**<div class="form-group col-md-6">**

**<label asp-for="EmpCode" class="control-label"></label>**

**<input asp-for="EmpCode" class="form-control" />**

**<span asp-validation-for="EmpCode" class="text-danger"></span>**

**</div>**

**<div class="form-group col-md-6">**

**<label asp-for="Position" class="control-label"></label>**

**<input asp-for="Position" class="form-control" />**

**<span asp-validation-for="Position" class="text-danger"></span>**

**</div>**

**</div>**

**<div class="form-group">**

**<label asp-for="OfficeLocation" class="control-label"></label>**

**<input asp-for="OfficeLocation" class="form-control" />**

**<span asp-validation-for="OfficeLocation" class="text-danger"></span>**

**</div>**

**<div class="form-row">**

**<div class="form-group col-md-6">**

**<input type="submit" value="Submit" class="btn btn-primary btn-block" />**

**</div>**

**<div class="form-group col-md-6">**

**<a asp-action="Index" class="btn btn-secondary btn-block"><i class="fa fa-table"></i>** Back to List**</a>**

**</div>**

**</div>**

**</form>**

**</div>**

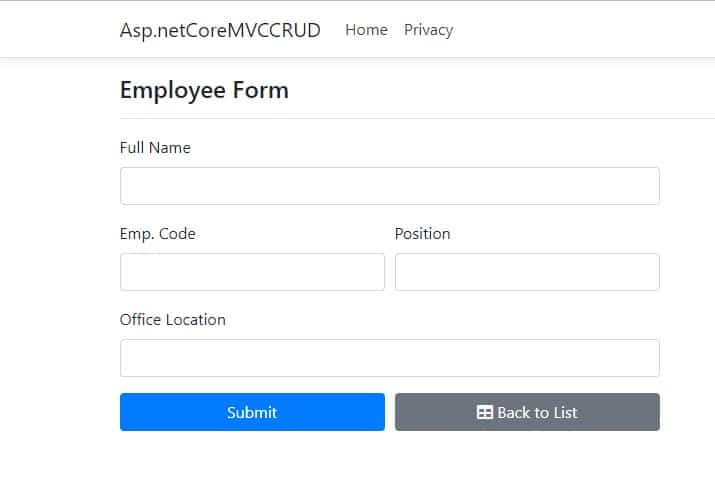
**</div>**

@section Scripts {

@{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}

}

This view is returned from AddOrEdit action method of the type GET. The method returns a fresh instance of the employee model, if the id parameter is 0. Else corresponding employee record with given id is returned. Finally, the returned model is populated inside AdddOrEdit.cshtml view. For an insert operation the form looks like this.



This formwill submited to AddOrEdit Post action method. We’ve handled both operations within if-else block based on the value of id parameter. We just need to call Add or Update method from DbContext object, for insert and update operation respectively. After all, invoke the Entity Framework method SaveChangesAsync. If the operation is successful we’ll be redirected to Index action method to return the list of records with the latest modification.

## Form Validation

Inside our Employee model, we defined FullName property with Required attribute, Which is means the property should contain a value before submitting the above form. [Entity Framework Core Validation](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/database-first-development/enhancing-data-validation) support many attributes like this. We’ve checked the validation status of the model within GET action method AddOrEdit with ModelState.IsValid property.

If there is any validation error, the same form is returned with the same model updated with validation error message.

## Delete a Record

You could delete a record with delete button from index view, it will handle in Delete action method.

**// GET: Employee/Delete/5**

**public** **async** **Task<IActionResult>** **Delete(int?** id**)**

**{**

**var** employee **=await** \_context**.**Employees**.FindAsync(**id**);**

\_context**.**Employees**.Remove(**employee**);**

**await** \_context**.SaveChangesAsync();**

**return** **RedirectToAction(nameof(**Index**));**

**}**

Inside the method, FindAsync retrieves the corresponding record with the given id into employee variable, Finally invoke Remove method from Employees DbSet property. and don’t forget to call SaveChangesAsync method which executes respective SQL Commands in back-end SQL Server Engine.