**ASP.NET CORE - Learn CRUD Operations In Entity Framework Core From Zero To Hero [Updated .NET 7.0]**

In this ASP.NET Core tutorial, you will learn to perform CRUD operations in Entity Framework Core from the very beginning. This tutorial will help you to learn Entity Framework Core so that you can start using it in your web application.

Audience

This tutorial is made for everyone who wants to learn ASP.NET Core and Entity Framework Core quickly. All the things are explained in an easy & step-by-step manner. You can download the complete source code and run them in your PC.

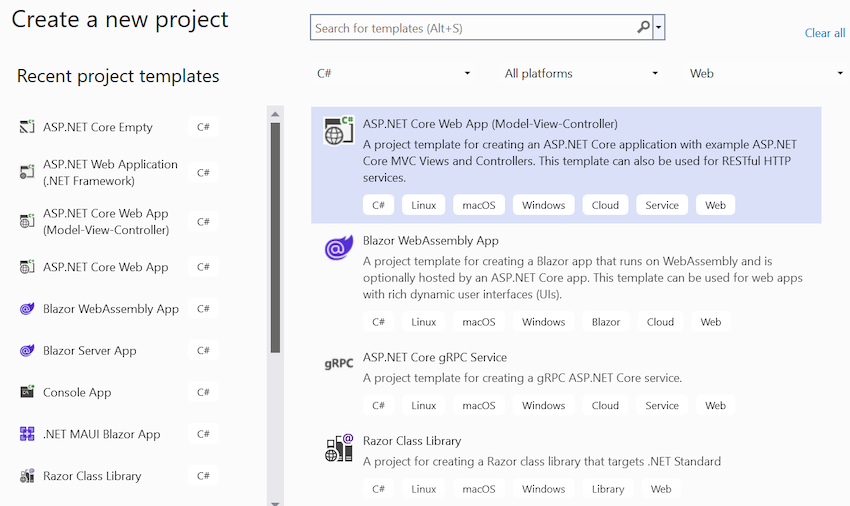
Here, I will perform the CRUD (CREATE, READ, UPDATE & DELETE) operations on a teachers table in the SQL Server database.

This tutorial is divided into the following sections.

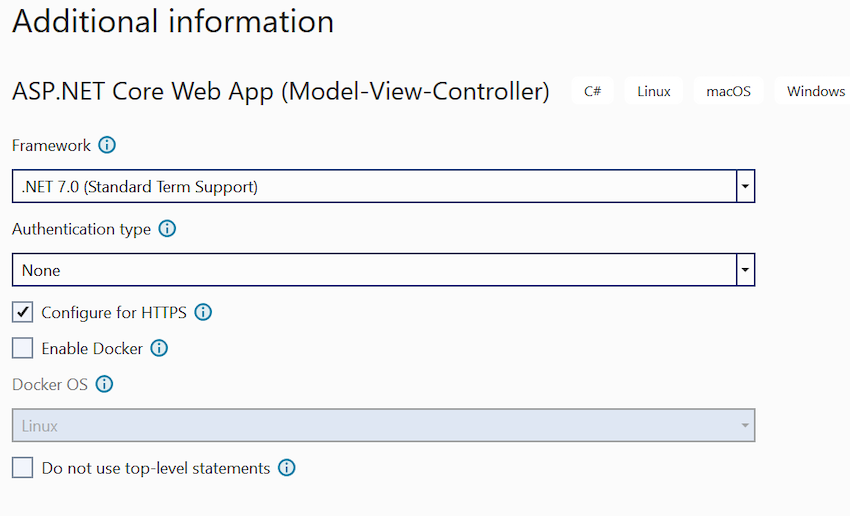
1. Creating a new ASP.NET Core Web Application
2. Install EF Core in the Application
3. Adding Connection String to the Database
4. Adding Model class for Teacher
5. Adding EF Core DbContext File
6. Creating a Database using EF Core Migration
7. Adding Controller code
8. Adding Client-Side Validation feature
9. Performing the CREATE Teacher Functionality
10. Performing the READ Teacher Functionality
11. Performing the UPDATE Teacher Functionality
12. Performing the DELETE Teacher Functionality

Creating a new ASP.NET Core Web Application

In Visual Studio create a new ASP.NET Core Web App (Model-View-Controller).



In the next window, give the app name as **CRUDEF** and provide it some location on your computer. Finally after it you select the latest version of .NET which is **.NET 7.0** right now.



The app will be based on MVC architecture and comes preconfigured with routing, Static files and necessary middlewares. All this is provided in the **Program.cs** class which is located in the root of the application.

Open the Program.cs class and you can see the default settings it contains. See it's code below:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see <https://aka.ms/aspnetcore-hsts.>

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

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

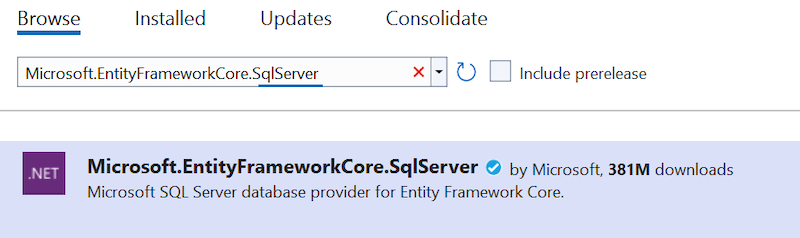
app.Run();

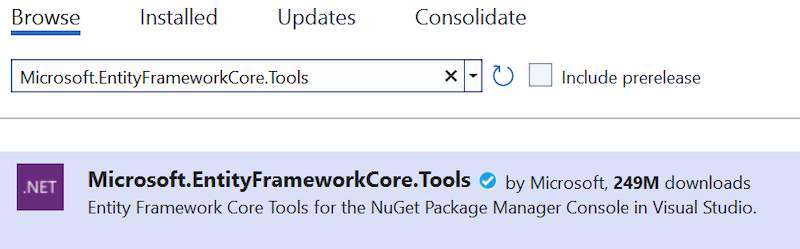
C#

You can register new services and add other new settings in this file. All this you can read at [ASP.NET Core Configurations – Program.cs Middleware AppSettings](https://www.yogihosting.com/aspnet-core-configurations/). Later in this tutorial we will be registering **Entity Framework Core** as a service in this file, so stay tuned.

Install EF Core in the Application

EF Core is an Object/Relational Mapping (O/RM) framework, an enhanced version of ADO.NET, which automates data storage and retrieval from the database. To Install EF Core in our app install the following 2 packages,

1. *Install EF Core DB Providers*  
   Go to *Tools > NuGet Package Manager > Manage NuGet Packages for Solution* and search for ***Microsoft.EntityFrameworkCore.SqlServer***. When it shows up, select it and install it to your app  
     
   
2. *Install EF Core Tools*  
   Next, search for **Microsoft.EntityFrameworkCore.Tools**. Once you get the result, install it.



Adding Connection String to the Database

A Connection String stores the parameters to the database and is used to connect the application to it. These parameters are the name of the driver, Server name and Database name. It also contains security information such as user name and password needed for the database connection.

The connection string is stored inside the appsettings.json file which resides in the root of the application. So add the following Connection string to this file.

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=School;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

JavaScript

I defined trusted Connection as true in the connection string so the Windows credentials of the current user are used to authenticate against the SQL Server.

Adding Model class for Teacher

Right click this ‘*Models*’ folder and select Add > New Item.

In the window that opens, select a new Class and name it **Teacher.cs**. Add the following properties to this class:

using CRUDEF.CustomValidation;

using System.ComponentModel.DataAnnotations;

namespace CRUDEF.Models

{

public class Teacher

{

[Required]

public int Id { get; set; }

[Required]

public string Name { get; set; }

[Required]

[SkillsValidate(Allowed = new string[] { "ASP.NET Core", "ASP.NET MVC", "ASP.NET Web Forms" }, ErrorMessage = "You skills are invalid")]

public string Skills { get; set; }

[Range(5, 50)]

public int TotalStudents { get; set; }

[Required]

public decimal Salary { get; set; }

[Required]

public DateTime AddedOn { get; set; }

}

}

C#

This class has 6 properties for the 6 columns in the Teacher table in the database. I have applied the [Required] attributes on Id, Name, Skills, Salary and AddedOn properties, the TotalStudent property is applied with the [Range] attribute so it will only allow values from 5 to 50.

Also, note the [SkillsValidate] attribute applied to the Skills property. The [SkillsValidate] attribute Is a custom validation attribute through which I will force Skills property to only accept 3 values which are:

1. *ASP.NET Core*
2. *ASP.NET MVC*
3. *ASP.NET Web Forms*

Next, I will create the **[SkillsValidate]** custom validation attribute. So, create a new folder on the root of the application and name it as *CustomValidation*. Inside this folder, add a new class called ***SkillsValidate.cs*** and add the following code to it.

using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;

namespace CRUDEF.CustomValidation

{

public class SkillsValidate : Attribute, IModelValidator

{

public string[] Allowed { get; set; }

public string ErrorMessage { get; set; }

public IEnumerable<ModelValidationResult> Validate(ModelValidationContext context)

{

if (Allowed.Contains(context.Model as string))

return Enumerable.Empty<ModelValidationResult>();

else

return new List<ModelValidationResult> {

new ModelValidationResult("", ErrorMessage)

};

}

}

}

C#

Adding EF Core DbContext File

We now have to add **Entity Framework Core Database Context** file for defining your database and tables. So, create a new class called **SchoolContext.cs** (on the root of the app) and add the following code to it.

using CRUDEF.Models;

using Microsoft.EntityFrameworkCore;

namespace CRUDEF

{

public class SchoolContext: DbContext

{

public SchoolContext(DbContextOptions<SchoolContext> options) : base(options) { }

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Teacher>(entity =>

{

entity.Property(e => e.Name)

.IsRequired()

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.Skills)

.IsRequired()

.HasMaxLength(250)

.IsUnicode(false);

entity.Property(e => e.Salary)

.IsRequired()

.HasColumnType("money");

entity.Property(e => e.AddedOn)

.HasColumnType("date")

.HasDefaultValueSql("(getdate())");

});

}

}

}

C#

The class inherits from ‘*DbContext*’ class and defines a ‘*DbSet*’ of type Teacher. So, when migrations are run, the School database will be created along with the ‘*Teacher*’ table in it.

Inside the **OnModelCreating()** method I have specified the **column types, max length and SQL types** for these properties.

Now, I have to add SchoolContext class as a service and also specify how it will get the connection string which is stored in the ‘*appsettings.json*’ file. I do this inside the Program.cs class, so update its code to include the following code line after builder.Services.AddControllersWithViews().

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

C#

I have added **SchoolContext** as a service and provided with the connection string from the configuration file,

**Creating a Database using EF Core Migration**

Now run the Migrations in Entity Framework Core so that the ‘School’ database and ‘Teacher’ table are created. So open the Package Manager Console window in Visual Studio then run this below command,

PM> **add-migration Migration1**

Or

**dotnet ef migrations add Migration1 (in vs code or package manager console)**

C#

This command will create Migration classes inside **Migrations** folder in the root of the app. Next, execute the migrations by the below command,

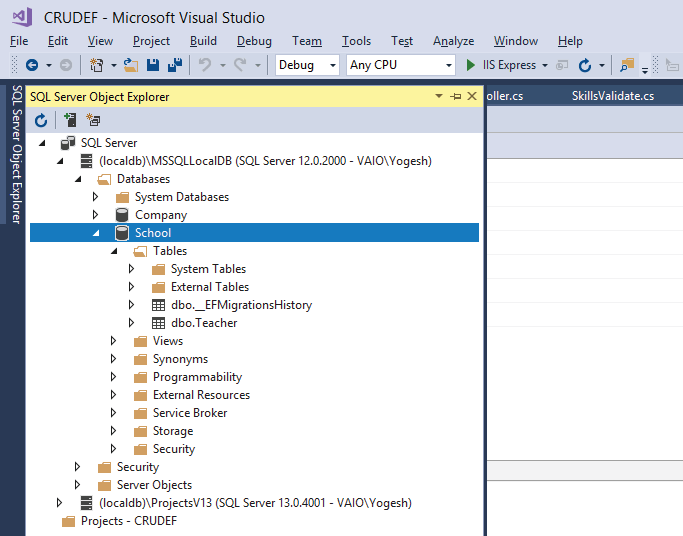
PM> Update-Database

**dotnet ef database update** **(in vs code or package manager console)**

C#

This will create the database in your SQL Server LocalDB database.

Now right click the ‘*(localdb)\MSSQLLocalDB*’ node, in the SQL Server Object Explorer window, and select ‘*Refresh*’. You will see the School database is created.



You can find this newly created ‘*Teachers*’ table in the database. Open it's node to find 6 columns which are:

1. Id – a primary key column that will be auto-generated from 1 since it is provided with *Identity(1,1)* attribute.
2. Name – a varchar(50) column for storing the name of the teacher.
3. Skills – for storing the skills of the teacher.
4. TotalStudents – an Int column for storing the total number of students which a given teacher teaches.
5. Salary – for storing the salary of the teacher.
6. AddedOn – for storing the time of the creation of the record. The column has a Date type and is it automatically gets the current time using the GETDATE() SQL function.

Note that all columns have NOT NULL attribute unchecked so they cannot contain NULL value in them.

Adding Controller code

To the ***HomeController.cs*** file we will write the the **Entity Framework Core code to perform CRUD operations** for the teachers. We will need the ‘*StudentContext*’ service injected to it using Dependency Injection feature. So update this controller as shown below:

using Microsoft.AspNetCore.Mvc;

namespace CRUDEF.Controllers

{

public class HomeController : Controller

{

private SchoolContext schoolContext;

public HomeController(SchoolContext sc)

{

schoolContext = sc;

}

public IActionResult Index()

{

return View();

}

}

}

C#

Adding Client Side Validation feature

The Client Side validation feature is performed by ‘jQuery’ and 2 validation plugins,

1. jQuery Validation
2. jQuery Validation Unobtrusive

All these comes pre-installed in the app. It is still good to know their installation commands which are given below.

PM> Install-Package jQuery

PM> Install-Package jQuery.Validation

PM> Install-Package jQuery.Validation.Unobtrusive

C#

Performing the CREATE Teacher Functionality

The CREATE Teacher Functionality is done through a new Action method called ‘CREATE’. Add this action method to the ‘Home Controller’. It is shown in the below code,

public IActionResult Create()

{

return View();

}

[HttpPost]

public IActionResult Create(Teacher teacher)

{

if (ModelState.IsValid)

{

schoolContext.Teacher.Add(teacher);

schoolContext.SaveChanges();

return RedirectToAction("Index");

}

else

return View();

}

C#

The [HttpPost] version of the Create action creates a new teacher in the database. It uses Entity Framework Core to create the record. Notice the below 2 lines of code with actually does the record creation in the database,

schoolContext.Teacher.Add(teacher);

schoolContext.SaveChanges();

C#

Next, we create the view for this create action method. So to the ***Views > Home***folder in the app, create a new **Razor View - Empty** file and name it *Create.cshtml.* Add the following code to this View file,

@model Teacher

@{

Layout = "\_Layout";

var title = "CREATE Teacher";

ViewData["Title"] = title;

}

<style>

.input-validation-error {

border-color: red;

}

</style>

<h2>@title</h2>

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

<form class="m-1 p-1" method="post">

<div class="form-group">

<label asp-for="Name"></label>

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

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

</div>

<div class="form-group">

<label asp-for="Skills"></label>

<input asp-for="Skills" type="text" class="form-control" />

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

</div>

<div class="form-group">

<label asp-for="TotalStudents"></label>

<input asp-for="TotalStudents" type="text" class="form-control" />

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

</div>

<div class="form-group">

<label asp-for="Salary"></label>

<input asp-for="Salary" type="text" class="form-control" />

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

</div>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

@section scripts {

<script src="~/lib/jquery/dist/jquery.min.js"></script>

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

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

}

C#

The View has a Model of type ‘*Teacher*’ and it creates Input fields for ‘*Name, Skills, TotalStudents and Salary*’, so that user can fill and submit it.

When the Submit button is clicked then the Create Action of type HttpPost, is called and the new teacher records is created.

Notice the 3 script files which do the Client Side Validation of the input fields in the View, We need to add them to the scripts section given on the **\_Layout.cshtml** page. That is why we have used @section scripts {} and then added the 3 scripts to them.

@section scripts {

<script src="~/lib/jquery/dist/jquery.min.js"></script>

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

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

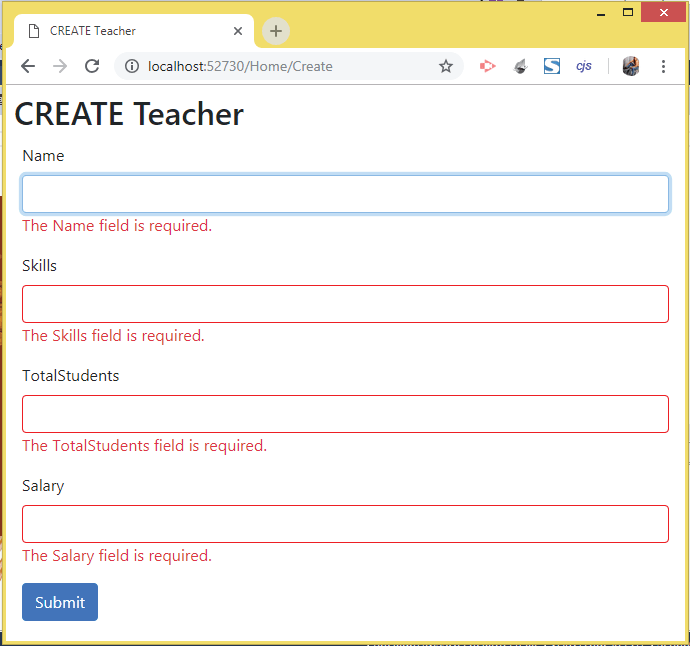
}

JavaScript

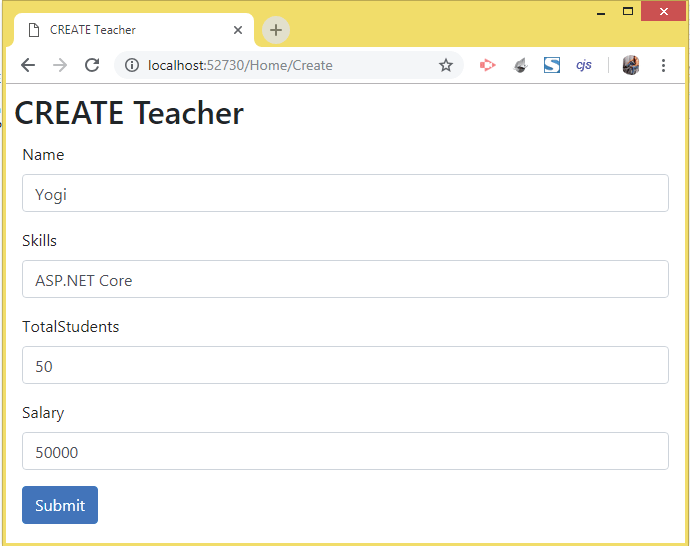
**Testing the Create Teacher functionality**

Run the application and in the browser go to the URL of ‘Create’ View, which is – http://localhost:52730/Home/Create. You will see the Create Teacher form in the browser.

Without filling any fields press the Submit button and you will see the validation errors displayed by jQuery Validation plugins instantly without page reload.



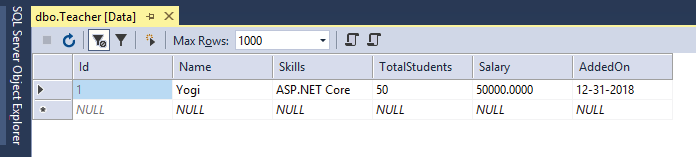
Now fill all the fields (as shown by the below image) and click the submit button.



The teacher record will be created in the database and you will be redirected to the Index View which is currently empty.

You can confirm the record is inserted in the Teacher’s table. For this go to the ‘*SQL Server Object Explorer*’, then right click the ‘*Teacher*’ table and select ‘*View Data*’.

The teacher table will open and you will see the new teacher record in it, see the below image:



Here we have taken a single table only, in your app you can add more tables and create relationships like one-to-many, one-to-one between them, and then perform database inserts and other operations. For this you can refer this [EF Core article](https://www.yogihosting.com/fluent-api-one-to-many-relationship-entity-framework-core/).

Performing the READ Teacher Functionality

Now we will create the READ Teacher Functionality. So change the **Index Action** in the Home Controller to return all the teachers to the View as shown below:

public IActionResult Index()

{

return View(schoolContext.Teacher);

}

C#

The code – **schoolContext.Teacher** will get all the teachers entity from Entity Framework Core.

Next, to the ***Index.cshtml***view given inside the ‘Views/Home/’ folder, add the following code,

@model IEnumerable<Teacher>

@{

Layout = "\_Layout";

var title = "READ Teacher";

ViewData["Title"] = title;

}

<h2>@title</h2>

<h3><a asp-action="Create" class="btn btn-sm btn-secondary">Create</a></h3>

<table class="table table-bordered table-sm table-striped">

<thead>

<tr><th>Id</th><th>Name</th><th>Skills</th><th>Total Students</th><th>Salary</th><th>Added On</th><th>Update</th><th>Delete</th></tr>

</thead>

<tbody>

@if (Model == null)

{

<tr><td colspan="7" class="text-center">No Model Data</td></tr>

}

else

{

@foreach (var p in Model)

{

<tr>

<td>@p.Id</td>

<td>@p.Name</td>

<td>@p.Skills</td>

<td>@p.TotalStudents</td>

<td>@string.Format(new System.Globalization.CultureInfo("en-US"), "{0:C2}", p.Salary)</td>

<td>@string.Format("{0:dddd, dd MMMM yyyy}", p.AddedOn)</td>

<td><a asp-action="Update" asp-route-id="@p.Id">Update</a></td>

<td>

<form asp-action="Delete" method="post" asp-route-id="@p.Id">

<button>Delete</button>

</form>

</td>

</tr>

}

}

</tbody>

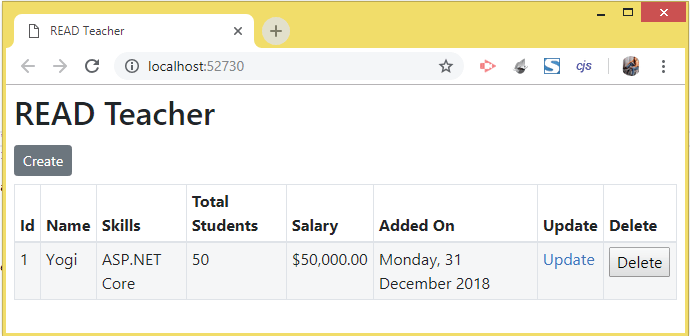
</table>

C#

Notice that I have also created the columns for Update and Delete functionality in the table’s thread and body elements, We will create the Update and Delete Functionality in the next sections.

**Testing the Read Teacher functionality**

Run your application and you will see the Teacher’s table records get displayed in the Index View. This is shown by the below image,



Performing the UPDATE Teacher functionality

Add the 2 Update Actions to the Home Controller as shown below:

public IActionResult Update(int id)

{

return View(schoolContext.Teacher.Where(a => a.Id == id).FirstOrDefault());

}

[HttpPost]

[ActionName("Update")]

public IActionResult Update\_Post(Teacher teacher)

{

schoolContext.Teacher.Update(teacher);

schoolContext.SaveChanges();

return RedirectToAction("Index");

}

C#

The HttpGet version of Update action method takes the ‘id’ of the Teacher records in its parameter and fetches the record from the database using EF Core like:

schoolContext.Teacher.Where(a => a.Id == id).FirstOrDefault());

C#

The HttpPost version of Update action method takes the Teacher class in it’s parameter which is bound with updated values from the Update View. The Updated values are saved in the database table by EF core like this:

schoolContext.Teacher.Update(teacher);

schoolContext.SaveChanges();

C#

Next, create the Update View inside the **Views/Home/** folder with the following code,

@model Teacher

@{

Layout = "\_Layout";

var title = "UPDATE Teacher";

ViewData["Title"] = title;

}

<style>

.input-validation-error {

border-color: red;

}

</style>

<h2>@title</h2>

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

<form class="m-1 p-1" method="post">

<div class="form-group">

<label asp-for="Id"></label>

<input asp-for="Id" type="text" readonly class="form-control" />

</div>

<div class="form-group">

<label asp-for="Name"></label>

<input asp-for="Name" type="text" class="form-control" />

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

</div>

<div class="form-group">

<label asp-for="Skills"></label>

<input asp-for="Skills" type="text" class="form-control" />

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

</div>

<div class="form-group">

<label asp-for="TotalStudents"></label>

<input asp-for="TotalStudents" type="text" class="form-control" />

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

</div>

<div class="form-group">

<label asp-for="Salary"></label>

<input asp-for="Salary" type="text" class="form-control" />

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

</div>

<div class="form-group">

<label asp-for="AddedOn"></label>

<input asp-for="AddedOn" type="text" class="form-control" asp-format="{0:d}" />

</div>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

@section scripts {

<script src="~/lib/jquery/dist/jquery.min.js"></script>

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

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

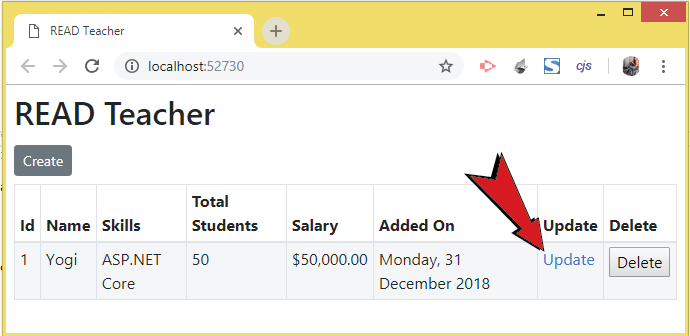
}

C#

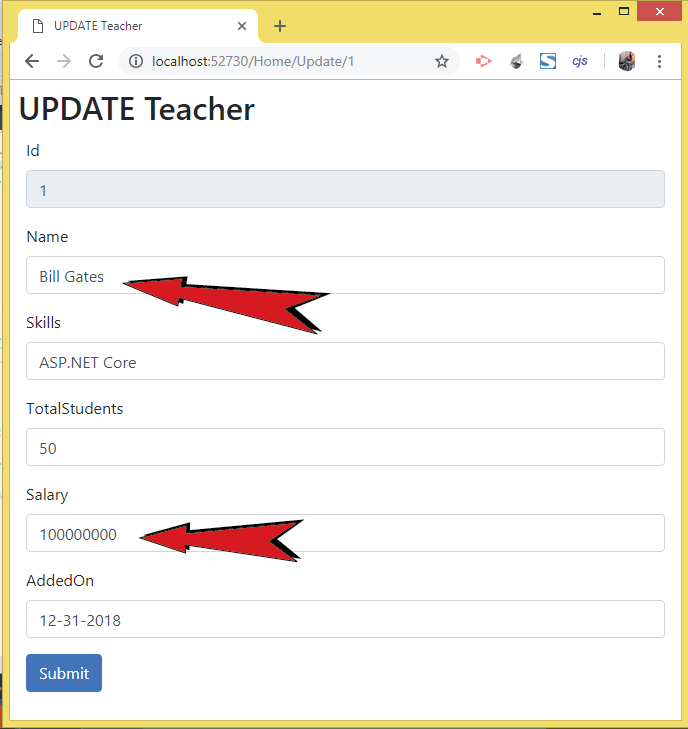
The View is similar to the **Create.cshtml** view we created earlier. I have made the ‘Id’ field as readonly so that user cannot change it.

**Testing the Update Teacher functionality**

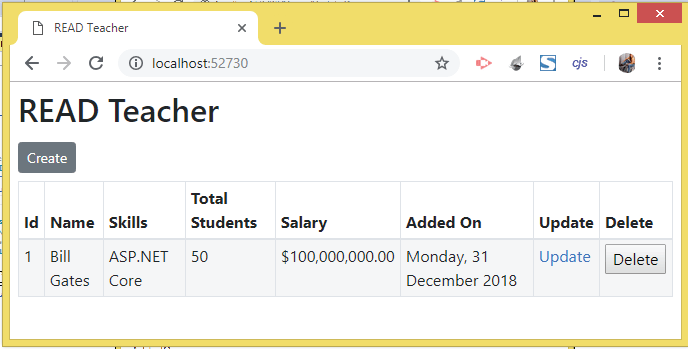
Run your application and click the ‘Update’ link for the 1st teacher records, see below image,



The record will open for performing updation. Change name to ‘Bill Gates’ and salary to ‘*100000000*’. Finally click the submit button as shown by the below image,



Record will be updated and you will be redirected to the Index View where you can see the updated record fields as shown by the below image,



Performing the DELETE Teacher Functionality

Create Delete Action method in the Home Controller whose code is given below:

[HttpPost]

public IActionResult Delete(int id)

{

var teacher = schoolContext.Teacher.Where(a => a.Id == id).FirstOrDefault();

schoolContext.Teacher.Remove(teacher);

schoolContext.SaveChanges();

return RedirectToAction("Index");

}

C#

This method takes the id of the teacher’s record in its parameter and fetches that record from the database table using the below EF Core code:

var teacher = schoolContext.Teacher.Where(a => a.Id == id).FirstOrDefault();

C#

Finally it deletes that record from the database table like:

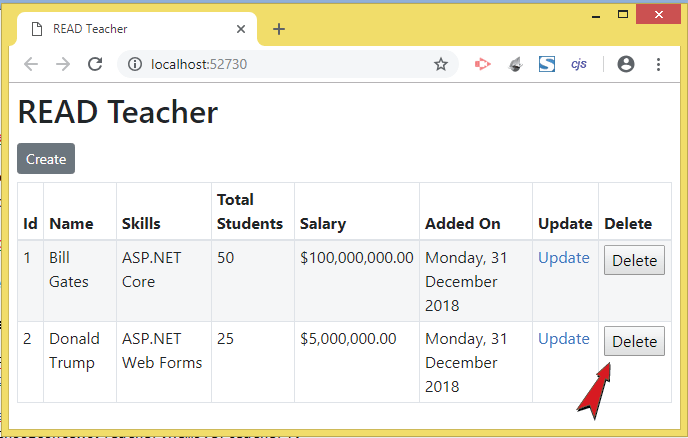
schoolContext.Teacher.Remove(teacher);

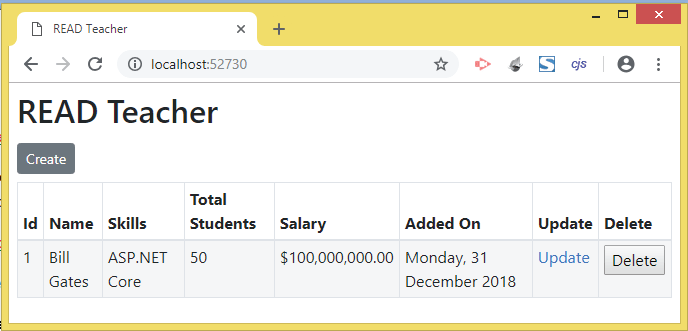
schoolContext.SaveChanges();

C#

**Testing the Delete Teacher functionality**

Run your application and click the **Delete** button given against each record. Check the below images which shows the record deletion process,





Conclusion

This completes this CRUD operations tutorial in ASP.NET Core and Entity Framework core. I hope you loved reading this and now understand how the codes are working. If you have any confusion go through all the *Source Codes* which you can download from this tutorial itself. **Remember that when you run the source codes do make the database first by running the migration commands otherwise you will encounter errors**.

I have also written a related tutorial in C# Corner which you should read - [5 Quick Ways To Delete Duplicate Records From Database Which Every Developer Must Know](https://www.c-sharpcorner.com/article/5-quick-ways-to-delete-duplicate-records-from-database-which-every-developer-mus/)

Please like and share this tutorial on your Facebook and Twitter accounts and let your friends learn ASP.NET Core. It hardly takes a few seconds of your time.

Thank you!