Client Server Architecture

What is Client-Server Architecture?

At its core, **client-server architecture** resembles the process of ordering a pizza for delivery. Imagine you call a pizza store to place an order. Someone at the store takes your order and then delivers the pizza to your doorstep. Simple, right? This analogy captures the fundamental principle of client-server architecture:

1. **Server**: The server provides requested services. It hosts, manages, and delivers resources.
2. **Clients**: Clients are the ones who request services from the server.

In the world of computing, client-server architecture refers to a model where most of the resources and services are managed by a central server. Here are the key points:

- **Definition**: Client-server architecture is a computing model in which clients (such as workstations or personal computers) connect to a central server over a network or the internet.
- **Resource Sharing**: In this model, various systems share computing resources via the network.
- **Server Role**: The server hosts and delivers services requested by clients.
- **Network Delivery**: All requests and services flow over the network.

Components of Client-Server Architecture

The architecture consists of the following components:

1. **Clients**: These are the end-user devices (workstations, laptops, or personal computers) that interact with the server to request services.
2. **Server**: The central server manages resources, processes requests, and delivers services.
3. **Network**: The communication medium (wired or wireless) that connects clients and the server.

How Does Client-Server Architecture Work?

1. A client sends a request (such as fetching a web page or accessing a database) to the server.
2. The server processes the request, retrieves the necessary data, and sends it back to the client.
3. The client displays the received information to the user.

Types of Client-Server Architecture

1. **1-Tier Architecture**:
   - Also known as the **single-tier architecture**.
   - In this model, the client and server are tightly coupled, often running on the same machine.
   - Commonly used for standalone applications.
2. **2-Tier Architecture**:

SUNBEAM

   - Also called the **client-server architecture**.
   - The client interacts directly with the server.
   - Commonly used for database-driven applications.
3. **3-Tier Architecture**:
   - Divides the system into three layers:
     - **Presentation Layer**: Handles user interface and interaction (client-side).
     - **Application Layer**: Contains business logic (server-side).
     - **Data Layer**: Manages data storage (database server).
   - Commonly used for web applications.
4. **N-Tier Architecture**:
   - Extends the 3-tier model by adding more layers (such as security, caching, etc.).
   - Scalable and flexible for large-scale applications.

Advantages and Disadvantages

Advantages:

- **Resource Centralization**: Centralized management of resources simplifies maintenance.
- **Scalability**: Easily scale by adding more servers or clients.
- **Security**: Centralized security measures.
- **Load Distribution**: Distributes load across servers.

Disadvantages:

- **Single Point of Failure**: If the server fails, all clients are affected.
- **Network Dependency**: Relies on network connectivity.
- **Complexity**: Managing servers and clients can be complex.

Conclusion

Client-server architecture plays a crucial role in modern computing, enabling efficient resource sharing and service delivery. Whether you're browsing the web, using online banking, or shopping online, you're interacting with client-server systems!

ASP.NET Web Forms VS ASP.NET MVC VS ASP.NET Core MVC

The differences between **ASP.NET Web Forms**, **ASP.NET MVC**, and **ASP.NET Core MVC** to understand their unique features and use cases:

SUNBEAM

1. **ASP.NET Web Forms**:
   - **Development Model**: Web Forms follow a **traditional event-driven development model**. Each page typically has a code-behind file where you write server-side logic.
   - **Server Controls**: Web Forms heavily rely on **server controls** for building UI components.
   - **View State**: Web Forms use **View State** for client-side state management.
   - **Pros**:
     - Familiarity for developers accustomed to desktop application development.
     - Rapid application development using drag-and-drop controls.
     - Rich server controls library.
   - **Cons**:
     - Tightly coupled code-behind model.
     - Complex page lifecycle.
     - Heavy reliance on ViewState can lead to larger page sizes.
2. **ASP.NET MVC**:
   - **Design Pattern**: MVC stands for **Model-View-Controller**. It is a **software design pattern** that emphasizes separation of concerns.
   - **Components**:
     - **Model**: Represents business logic, data manipulation, and database access.
     - **View**: Responsible for UI presentation (HTML, CSS, JavaScript).
     - **Controller**: Acts as an intermediary between Model and View, handling requests and responses.
   - **Features**:
     - **Separation of Concerns**: MVC decouples the UI (View) from the business logic (Model) and request handling (Controller).
     - **SEO-Friendly URLs**: MVC provides clean and customizable URLs.
     - **HTML Helpers**: Offers HTML helpers for form controls (though plain HTML can also be used).
   - **Pros**:
     - Better testability and maintainability.
     - Flexibility in choosing technologies (HTML5, Ajax, Web API, etc.).
     - Good for SEO.
   - **Cons**:
     - Learning curve for developers new to MVC.
     - Requires understanding of the MVC pattern.
3. **ASP.NET Core MVC**:
   - **Cross-Platform**: ASP.NET Core is a **cross-platform framework**, allowing you to build applications that run on Windows, Linux, or macOS.
   - **Lightweight and Modern**:
     - **MVC pattern-based development**.
     - **HTML helpers** for form controls.
     - No reliance on ViewState.
   - **Pros**:
     - Cross-platform compatibility.
     - Lightweight and modular.
     - Keeps you updated with modern web technologies.
   - **Cons**:

SUNBEAM

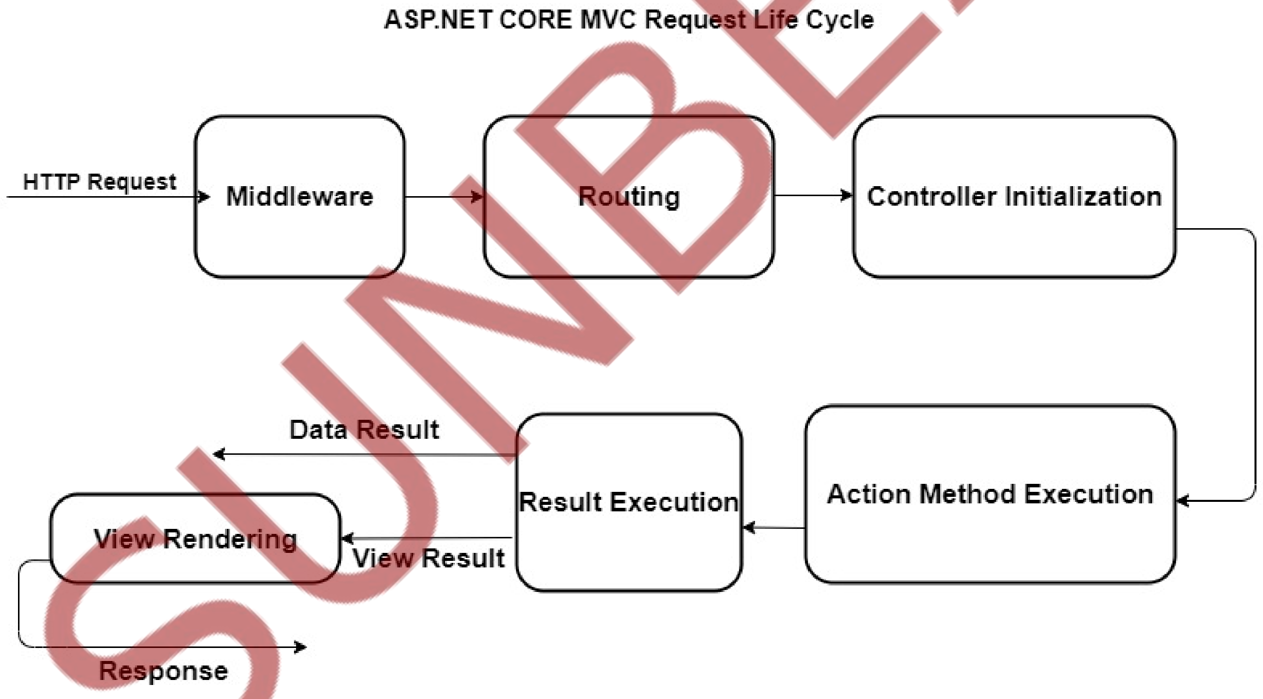
- Learning curve if transitioning from Web Forms.
- Some legacy libraries may not be available.

In summary, choose **ASP.NET Web Forms** if you prefer the traditional event-driven model, **ASP.NET MVC** for better separation of concerns and SEO-friendly URLs, and **ASP.NET Core MVC** for cross-platform development and modern features.

Life Cycle of MVC

Introduction

The ASP.NET Core MVC Request Life Cycle is a sequence of events, stages or components that interact with each other to process an HTTP request and generate a response that goes back to the client. In this article, we will discuss each and every stage of ASP.NET Core MVC Request Life Cycle in detail.



The request life cycle consists of various stages which are:

Middleware

Middleware component forms the basic building block of application HTTP pipeline. These are a series of components that are combined to form a request pipeline in order to handle any incoming request.

SUNBEAM

Routing

Routing is a middleware component that implements MVC framework. The routing Middleware component decides how an incoming request can be mapped to Controllers and actions methods, with the help of convention routes and attribute routes.

Controller Initialization

At this stage of ASP.NET MVC Core Request Life Cycle, the process of initialization and execution of controllers takes place. Controllers are responsible for handling incoming requests. The controller selects the appropriate action methods on the basis of route templates provided.

Action method execution

After the controllers are initialized, the action methods are executed and returns a view which contains Html document to be returned as response to the browser.

Result Execution

During this stage of ASP.NET MVC Core Request Life Cycle, result i.e. the response generated to the original HTTP request, is executed. If an action method returns a view result, the MVC view engine renders a view and returns the HTML response. If result is not of view type, then action method will generate its own response.

Now, we will discuss each stage briefly

Middleware

Middleware component forms the basic building block of the application'a HTTP pipeline. These are aseries of components that are combined to form a request pipeline in order to handle any incoming request. Whenever a new request comes, it is passed to the first middleware component. The component then decides, whether to generate a response after handling that incoming request or to pass it to the next component. The response is sent back along these components, once the request has been handled.

Most middleware components are implemented as standalone classes, which are content generating middleware. Now, we will be creating a content generating middleware component by adding ContentComponent.cs class.

Example

```
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
namespace WebApplication
{
  public class ContentComponent
   {
     private RequestDelegate nextComponent;
     public ContentComponent(RequestDelegate nextMiddleware) =>
```

SUNBEAM

```
nextComponent = nextMiddleware;
     public async Task Invoke(HttpContext httpContext)
      {
        if (httpContext.Request.Path.ToString().ToLower() == "/edit")
         {
           await httpContext.Response.WriteAsync("This is edit URL
component", Encoding.UTF8);
         }
        else
         {
           await nextComponent.Invoke(httpContext);
         }
      }
   }
}
```

This ContentComponent middleware component is defining a constructor, and inside constructor its taking RequestDelegate object. This RequestDelegate object represents the next middleware component. ContentComponent component also defines an Invoke method. When ASP.NET Core application receives an incoming request, the Invoke method is called.

The HttpContext argument of the Invoke method provides information about the incoming HTTP request and the generated response that will be sent back to the client. The invoke method of ContentComponent class checks whether incoming request is sent to /edit URL or not. If the request has been sent to /edit url then, in response a text message is sent.

Routing

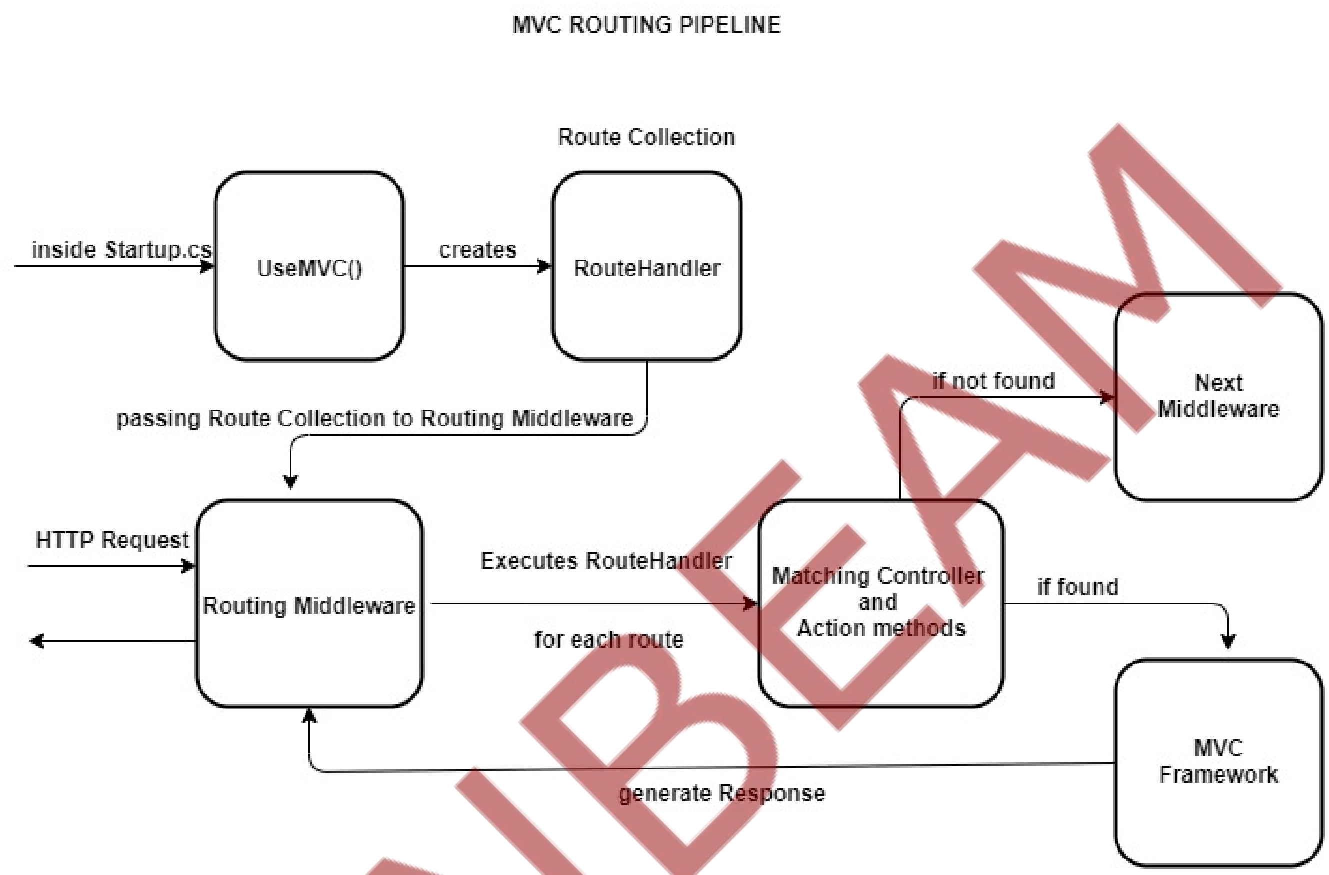
Routing is a middleware component that implements MVC framework. In ASP.NET Core, the routing system is used to handle MVC URLs. The routing Middleware component decides how an incoming request can be mapped to Controllers and actions methods, with the help of convention routes and attribute routes. Routing bridges middleware and MVC framework by mapping incoming request to controller action methods.

In a particular application, MVC registers one or more routes using MapRoute method. MVC provides default routes that are given a name and a template to match incoming request URLs. The Controllers and action variables are placeholders that are replaced by matching segments of the URL.

These routes are passed into and consumed by routing middleware.

SUNBEAM

The MVC routing pipeline



SUNBEAM

In ASP.NET Core, routing maps an incoming request to RouteHandler class, which is then passes as the collection of routes to the routing middleware. The routing middleware executes MVC RouteHandler for each route. If a matching controller and action method is found on a particular route, then the requested data is forwarded to the rest of the MVC framework which will generate a response.

There are two types of routing available in MVC which are:

1. **Convention routing**
    - This routing type uses application wide patterns to match a different URL to various controller action methods. Convention routes represent default possible routes that can be defined with the help of controller and action methods.
2. **Attribute routing**
    - This type of routing is implemented through attributes and applied directly to a specific controller or action method.

The convention routing methodology is implemented inside startup.cs file. In startup.cs, the UseMvc() method registers the routes, that are being supplied to it as a parameter inside MapRoute method.

Example

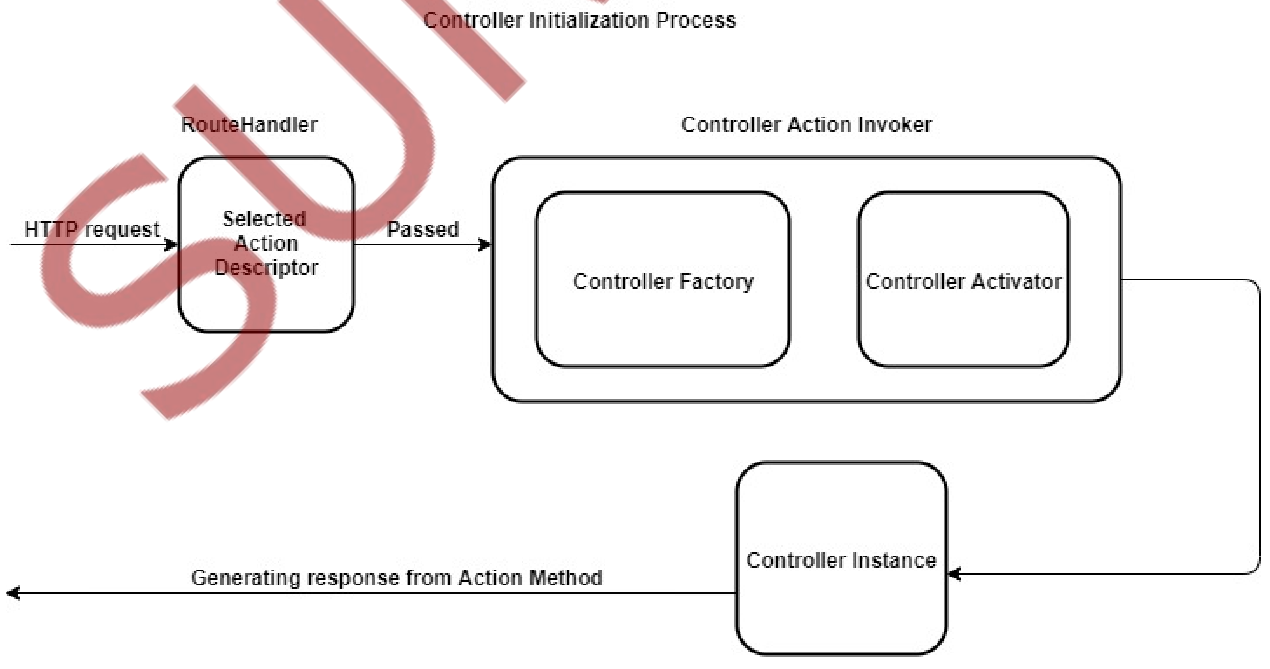
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

 app.UseMvc(routes => {
   routes.MapRoute(
   name: "default",
   template: "{controller=Home}/{action=Index}/{id?}");
 });
}
```

A route attribute is defined on top of an action method inside controller class.

```
public class MoviesController: Controller {
  [Route("movies / released / {year}/{month:regex(\\d{4})}")]
    public ActionResult ByReleaseYear(int year, int month) {
    return Content(year + "/"+month);
    }
}
```

Controller Initialization

At this stage, the process of initialization and execution of controllers takes place. Controllers are responsible for handling incoming requests which is done by mapping request to appropriate action method. The controller selects the appropriate action methods (to generate response) on the basis of route templates provided. A controller class inherits from controller base class. A controller class suffix class name with the Controller keyword.


Controller Initialization Process
RouteHandler
Controller Action Invoker
HTTP request
Selected Action Descriptor
Passed
Controller Factory
Controller Activator
Controller Instance
Generating response from Action Method

SUNBEAM

The MVC RouteHandler is responsible for selecting an action method candidate in the form of action descriptor. The RouteHandler then passes the action descriptor into a class called Controller action invoker. The class called Controller factory creates instance of a controller to be used by controller action method. The controller factory class depends on controller activator for controller instantiation.

After action method is selected, instance of controller is created to handle request. Controller instance provides several features such as action methods, action filters and action result. The activator uses the controller type info property on action descriptor to instantiate the controller by name. once the controller is created, the rest of the action method execution pipeline can run.

The controller factory is the component that is responsible for creating controller instance. The controller factory implements an interface called IControllerFactory. This interface contains two methods which are called CreateController and ReleaseController.

Action Method Execution Process

Authorization Filters

Authorization filters authorize or deny any incoming request.

Controller Creation

Instantiate Controller object.

Model Binding

Model Binding bind incoming request to Action method parameters.

Action Filters

OnActionExecuting method is always called before the action method is invoked.

Action Method Execution

Action method inside controller classes executes logic to create Action Result. Action method returns action results to generate response.

Action Filters

OnActionExecuted method called is always called after the action method has been invoked.

Authorization Filters

In ASP.NET Core MVC, there is an attribute called Authorize. Authorize is a filter, which can be applied to an action and it will be called by a MVC Core framework before and after that action or its result are executed.

SUNBEAM

These filters are used to provide security to application, including user authorization. If authorize attribute is applied to an action method, before the action is executed, the attribute will check if the current user is logged in or not. If not it will redirect the user to the login page. Authorization filters authorize or deny any incoming request.

In the below code example, Authorize Authorization filter(predefined filter) has been used over Index action method.

```
[Authorize]
public ViewResult Index() {
 return View();
}
```

The OnAuthorization method authorizes an HTTP request.

```
namespace WebApplication {
  public interface IAuthorizationFilter: IFilterMetadata {
   void OnAuthorization(AuthorizationFilterContext content);
}
```

Now I have demonstrated, how to create a custom Authorization filter. A class file called OnlyHttpsAttribute.cs has been created which defines the action filter.

```
using System;
  using Microsoft.AspNetCore.Http;
  using Microsoft.AspNetCore.Mvc;
  using Microsoft.AspNetCore.Mvc.Filters;
  namespace WebApplication {
   public class OnlyHttpsAttribute: IAuthorizationFilter {
    public void OnAuthorization(AuthorizationFilterContext approve) {
     if (!approve.HttpContext.Request.IsHttps) {
      approve.Result =
       new StatusCodeResult(StatusCodes.Status403Forbidden);
     }
    }
   } }
```

In the below example, OnlyHttps attribute has been applied to the Home controller

```
using Microsoft.AspNetCore.Mvc;
using WebApplication;
namespace Controllers {
 [OnlyHttps]
 public class HomeController: Controller {
  public ViewResult Index() => View("Message",
  "this URL is working fine");
   } }
```

SUNBEAM

Model Binding

Model binding maps incoming request to action method parameters. Action method parameters are inputs for an action method. Using data values acquired from HTTP request, model binding creates objects that an action method takes as an argument.

The model binders are the components which provide data values from incoming request to invoke action methods. The model binders search for these data values under three scenarios which are:

- Form data values
- Routing variables
- Query strings

Each of these sources are examined in sequence until and unless an argument value is found.

Let's consider a controller class as follows

```
using System.Linq;
using System.Web.Mvc;
namespace project.Controllers {
 public class CustomerController: Controller {
  public ActionResult Edit(int id) {
   return Content("id=", +id);
  }
 }
}
```

Now, suppose I requested an URL as /Customer/Edit/1

Here 1 is the value of id property. MVC translated that part of the URL and used it as the argument when it called the Edit method in the Customer controller class to service the request.To invoke the Edit method, MVC requires a certain type of value for the argument id, therefore model binding provide data values to invoke action methods.

Action Filters

Action filters are used to execute tasks instantly before and after action method execution is performed. In order to apply some logic to action methods, without having to add extra code to the controller class, action filters can be used either above the action method itself or above the Controller class. Action filter implements two types of interfaces which are IActionFilter and IAsyncActionFilter.

The IActionFilter interface which defines action filter is as follows:

```
public interface IActionFilter: IFilterMetadata {
 void OnActionExecuting(ActionExecutingContext content);
 void OnActionExecuted(ActionExecutedContext content); }
```

SUNBEAM

The method OnActionExecuting is always called before the action method is invoked, and the method OnActionExecuted is always called after the action method has been invoked, whenever an action filter is applied to an action method.

Whenever an action filter is created, the data is provided using two different context classes, ActionExecutingContext and ActionExecutedContext.

Now I will demonstrate, how to create a custom action filter by deriving a class from the ActionFilterAttribute class. A class file called ActionAttribute.cs has been created which defines the action filter.

```
using Microsoft.AspNetCore.Mvc.Filters;
namespace WebApplication {
 public class ActionAttribute: ActionFilterAttribute {
  private Stopwatch a;
  public override void OnActionExecuting(ActionExecutingContext content)
{
   a = Stopwatch.StartNew();
  }
  public override void OnActionExecuted(ActionExecutedContext content) {
   a.Stop();
   string output = "<div> calculated time is: " + $"
{a.Elapsed.TotalMilliseconds} ms </div>";
   byte[] bytes = Encoding.ASCII.GetBytes(output);
   content.HttpContext.Response.Body.Write(bytes, 0, bytes.Length);
  }
 }
}
```

In the below example, Action attribute has been applied to the Home controller

```
using Microsoft.AspNetCore.Mvc;
using WebApplication;
namespace Controllers {
 [Action]
 public class HomeController: Controller {
  public ViewResult Index() => View("Message",
   "This index action is executed between action attribute");
 }
}
```

Action Method Execution

Action methods return objects that implements the IActionResult interface from the Microsoft.AspNetCore.Mvc namespace. The various types of response from the controller such as rendering a view or redirecting the client to another URL, all these responses are handled by IActionResult object, commonly known as action result.

SUNBEAM

When an an action result is returned by a specific action method, MVC calls its ExecuteResultAsync method, which generates response on behalf of the action method. The ActionContext argument provides context data for generating the response, including the HttpResponse object.

Controllers contain Action methods whose responsibility is to generate a response to an incoming request. An action method is any public method inside a controller that responds to incoming requests.

```
Public class HomeController: Controller {
 Public IActionResult Edit() {
  Return View();
 }
}
```

Here, Edit action method is defined inside Home controller.

```
using Microsoft.AspNetCore.Mvc;
using WebApplication23;
namespace WebApplication23.Controllers {
 public class HomeController: Controller {
 public ViewResult Index() => View("Form");
  };
 }
}
```

Here, Index method is defined inside Home controller, which is returning a view.

Result Execution

The action method returns action result. This action result is the base class for all action results in asp.net mvc. Depending on the result of an action method, it will return an instance of one of the classes that derive from action result.

Different types of action result are

TYPES	HELPER METHOD	DESCRIPTION
ViewResult	View()	Renders a view and return HTML content
PartialViewResult	PartialView()	Returns partial view
ContentResult	Content()	Returns simple text
RedirectResult	Rediect()	Redirect result to URL
RedirectToRouteResult	RedirectToAction()	Redirect to an action method
JsonResult	Json()	Returns serialized json object

SUNBEAM

FileResult	File()	Returs a file
HttpNotFoundResult	HttpNotFound()	Returns not found or 404 error
EmptyResult		When action does not return values

During this stage of ASP.NET MVC Core Request Life Cycle, result i.e. the response generated to the original HTTP request, is executed. If an action method returns a view result, the MVC view engine renders a view and returns the HTML response. If result is not of view type, then action method will generate its own response.

```
using Microsoft.AspNetCore.Mvc;
using WebApplication23.example;
namespace WebApplication23.Controllers {
  public class HomeController: Controller {
  public ViewResult Index() => View("Form");
   public ViewResult ReceiveForm(string name, string state) =>
View("Details",$ "{name} lives in {state}");
}
```

In the above example Index action method is returning a view result, therefore the MVC view engine renders a view and returns the HTML response, which is a simple form.

Summary

In this article, we have discussed each and every stage of ASP.NET Core MVC Request Life Cycle in detail. All of the stages have been described with proper coding examples and life cycle diagrams.

Services and Dependency Injection

What Are Services?

- A service is a component intended for common consumption within an application.
- Services are made available through dependency injection (DI).

What is Dependency Injection?

Dependency Injection is a software development and design pattern that helps in eliminating the dependency of one class on the other class. Using dependency injection or DI primarily aims to build loosely coupled applications instead of tightly coupled ones.

The following are the top advantages of using dependency injection in .NET Core web API and applications:

SUNBEAM

- **Reusability**: Dependency injection enables developers to reuse a code block, as they can share dependencies between classes.
- **Easy Testing**: During unit testing procedures, dependency injection enables the developers to use mock objects for testing the application. It provides accurate results, leading to patching loopholes, fixing errors, and streamlining code maintenance in .NET Core.
- **Loose Coupling**: DI helps to remove the dependency of one class over another class. This will result in accurate output even if you modify a single class, and it will not impact the dependent class.
- **Extensibility**: With the use of dependency injection, the application becomes more flexible and scalable, allowing it to remove dependencies while retaining uptime.

The Need for Dependency Injection in .NET Core

Let’s see an example below to understand the need for dependency injection.

```
public class Car
{
   private AUDI _tata;
   public Car()
   {
      _tata = new TATA();
   }
}
public class TATA {
   public TATA()
   {
   } }
```

In the above code, two classes are present: class Car and class TATA. The class Car depends on class TATA for providing the output. Therefore, both these classes are tightly coupled, and due to this, each time the program runs, a TATA class object needs to be created.

Now, suppose that you add a ModelName parameter in the TATA class constructor. Then, you need to add and pass the same parameter in the Car class, similar to the following code example. Otherwise, your program will not provide the output, which is, in our case, the model name.

```
public class Car {
   private TATA _tata;
   public Car()
   {
      _tata = new TATA("Nexon EV");
   }
}
public class TATA
{
```

SUNBEAM

```
    public TATA(string ModelName)
    {
    }
}
```

Due to the tight coupling of these classes, modification made in one class impacts the other. It’s not a good practice for large projects, as it’s a time, cost, and effort-consuming task. Thus, you need dependency injection to break hard coupling and make your .NET application loosely coupled.

This is where the ASP.NET Cores DI framework comes in. The Responsibility of creating the object is now rests with the framework. The object that does this is called DI Container or Ioc Container.

There are many frameworks are available like Autofac, Unity etc. which you can use with the ASP.NET Core.

It involves the use of an interface or base class to abstract the dependency implementation, and registration of the dependency in a service container

.NET provides a built-in service container, IServiceProvider, and services are typically registered at the app's start-up and appended to an IServiceCollection

When designing services for dependency injection, it is important to avoid stateful, static classes and members, creating global state, and direct instantiation of dependent classes within services

Demo of Dependency Injection

Create Web Application

Create an ASP.NET Core project using the empty template and name it **DependencyInjection**. Add the **HomeController** with index method. You can follow these tutorials

Creating View Model

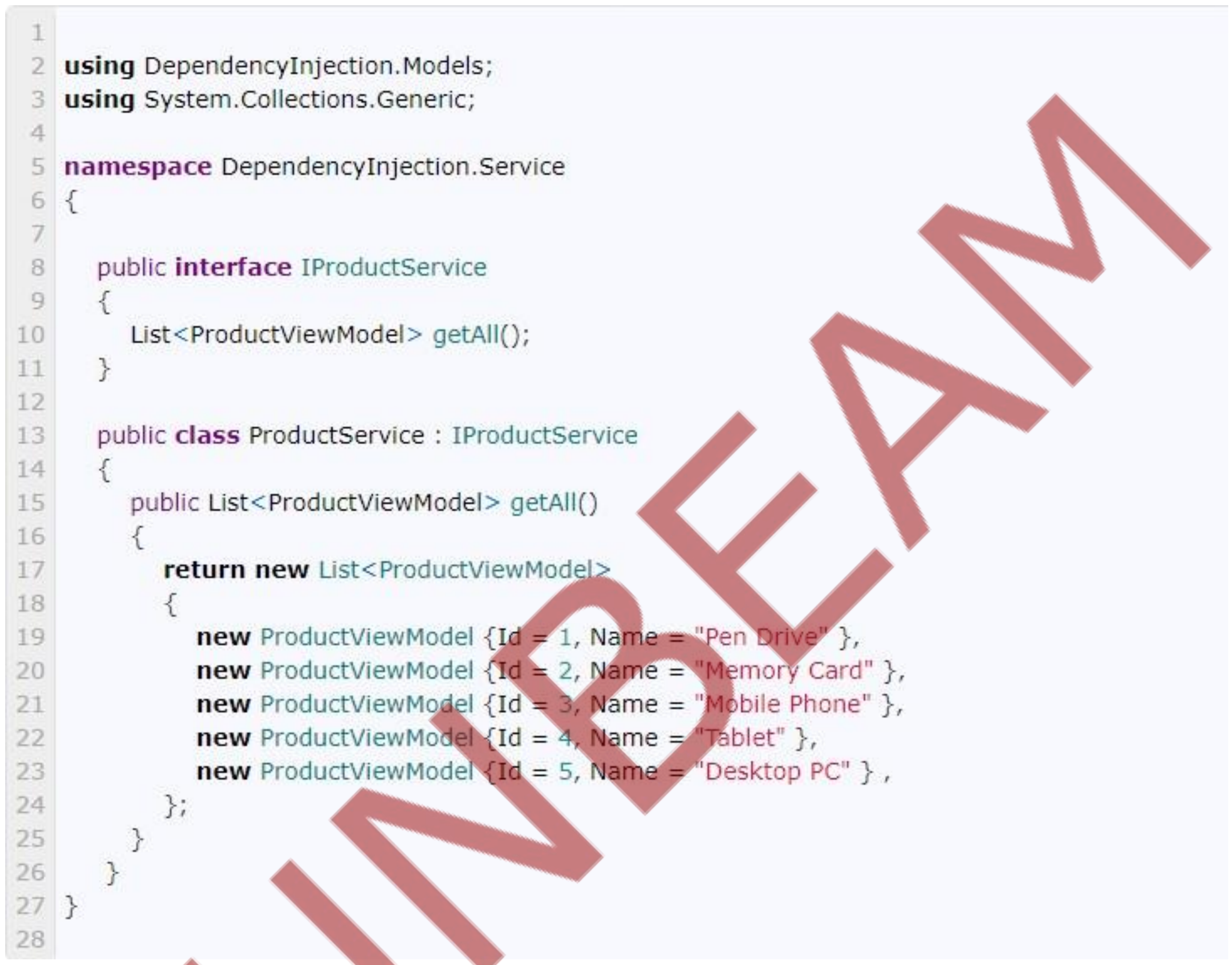
Create the **Models** folder and create the following **viewModel**.

```

public class ProductViewModel
{
    public int Id { get; set; }
    public string Name { get; internal set; }
}

```

SUNBEAM

Adding the Service

Create the folder **Services** and add a class with the name **ProductService.cs**.



```

using DependencyInjection.Models;
using System.Collections.Generic;

namespace DependencyInjection.Service
{

   public interface IProductService
   {
      List<ProductViewModel> getAll();
   }

   public class ProductService : IProductService
   {
      public List<ProductViewModel> getAll()
      {
         return new List<ProductViewModel>
         {
            new ProductViewModel {Id = 1, Name = "Pen Drive" },
            new ProductViewModel {Id = 2, Name = "Memory Card" },
            new ProductViewModel {Id = 3, Name = "Mobile Phone" },
            new ProductViewModel {Id = 4, Name = "Tablet" },
            new ProductViewModel {Id = 5, Name = "Desktop PC" } ,
         };
      }
   }
}

```

First, we added the **IProductService** interface, and then **ProductService** which implements that Interface.

The **ProductService** returns the list of Products. The Product List is hardcoded into the service. In real life, you go into the database to fetch the product List.

SUNBEAM

Using the Service in Controller

Next, Open the HomeController and add the following code.

```
using DependencyInjection.Service;
using Microsoft.AspNetCore.Mvc;

namespace DependencyInjection.Controllers
{
   public class HomeController : Controller
   {
      private IProductService _productService;

      public HomeController(IProductService productService)
      {
         _productService = productService;
      }

      public IActionResult Index()
      {
         return View(_productService.getAll());
      }
   }
}

```

The constructor of the **HomeController** asks for **ProductService** and stores the instance in local variable **_productService**.

The **index** method calls the view with the list of products by invoking the **getAll** method.

Creating View

The View just displays the list of Products as shown below.

```
@model List<DependencyInjection.Models.ProductViewModel>;

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

<h2>Index</h2>

@foreach (var product in Model)
{
   <p>@product.Id @product.Name</p>
}

```

SUNBEAM

Register the Service

The last step is to register the service in the Dependency Injection container.

Open the **startup.cs** and go to **ConfigureServices** method. This is where all services are configured for dependency injection.

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc();
   services.AddTransient<IProductService, ProductService>();
}
```

We have registered the **ProductService** using the **AddTransient** method. The other method available is **AddScoped** & **AddSingleton**.

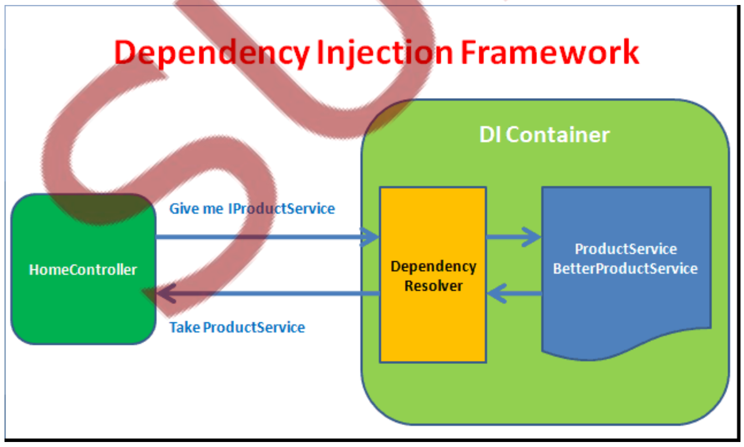
There are three ways, by which you can do that, and it in turn decides how the DI Framework manages the lifecycle of the services.

1. **Transient**: creates a new instance of the service, every time you request it.
2. **Scoped**: creates a new instance for every scope. (Each request is a Scope). Within the scope, it reuses the existing service.
3. **Singleton**: Creates a new Service only once during the application lifetime, and uses it everywhere

Now, run the application and you should be able to see the list of products.

How Dependencies are injected

The following image shows how **ProductService** was injected into the **HomeController**.



When the new **HomeController** is requested, the MVC asks the Dependency injection framework to provide an instance of the **HomeController** class.

The DI Container inspects the constructor of the **HomeController** and determines its dependencies.

It then searches for the dependencies in the services registered in the service collection and finds the matching service and creates an instance of it.

Then it creates the **HomeController** and passes the dependencies in its constructor.

SUNBEAM

ASP.NET Core MVC CRUD Operations using ADO.NET

Database Design

Creating database and Table

First of all, we will create a database named "StudentManagement".

```sql
CREATE DATABASE StudentManagement
```

Then, we will create a table named "Student".

```sql
Create table Student(
    Id int IDENTITY(1,1) NOT NULL,
    FirstName varchar(50) NOT NULL,
    LastName varchar(50) NOT NULL,
    Email varchar(30) NOT NULL,
    Mobile varchar(20) NOT NULL,
    Address varchar(220)  NULL,
)
```

Now, we will create stored procedures to add, delete, update, and get student data.

To Insert a student record

```sql
Create procedure spAddStudent
(
    @FirstName VARCHAR(50),
    @LastName VARCHAR(50),
    @Email VARCHAR(30),
    @Mobile VARCHAR(20),
    @Address VARCHAR(220)
)
as
Begin
    Insert into Student (FirstName,LastName,Email, Mobile,Address)

    Values (@FirstName,@LastName,@Email, @Mobile,@Address)

End
```

To Update a student record

```sql
Create procedure spUpdateStudent
(
    @Id INTEGER ,
```

SUNBEAM

```
    @FirstName VARCHAR(50),
    @LastName VARCHAR(50),
    @Email VARCHAR(30),
    @Mobile VARCHAR(20),
    @Address VARCHAR(220)
)
as
begin
    Update Student
    set FirstName=@FirstName,
    LastName=@LastName,
    Email=@Email,
    Mobile=@Mobile,
    Address=@Address
    where Id=@Id
End
```

To Delete a Student Record

```
Create procedure spDeleteStudent
(
    @Id int
)
as
begin
    Delete from Student where Id=@Id
End
```

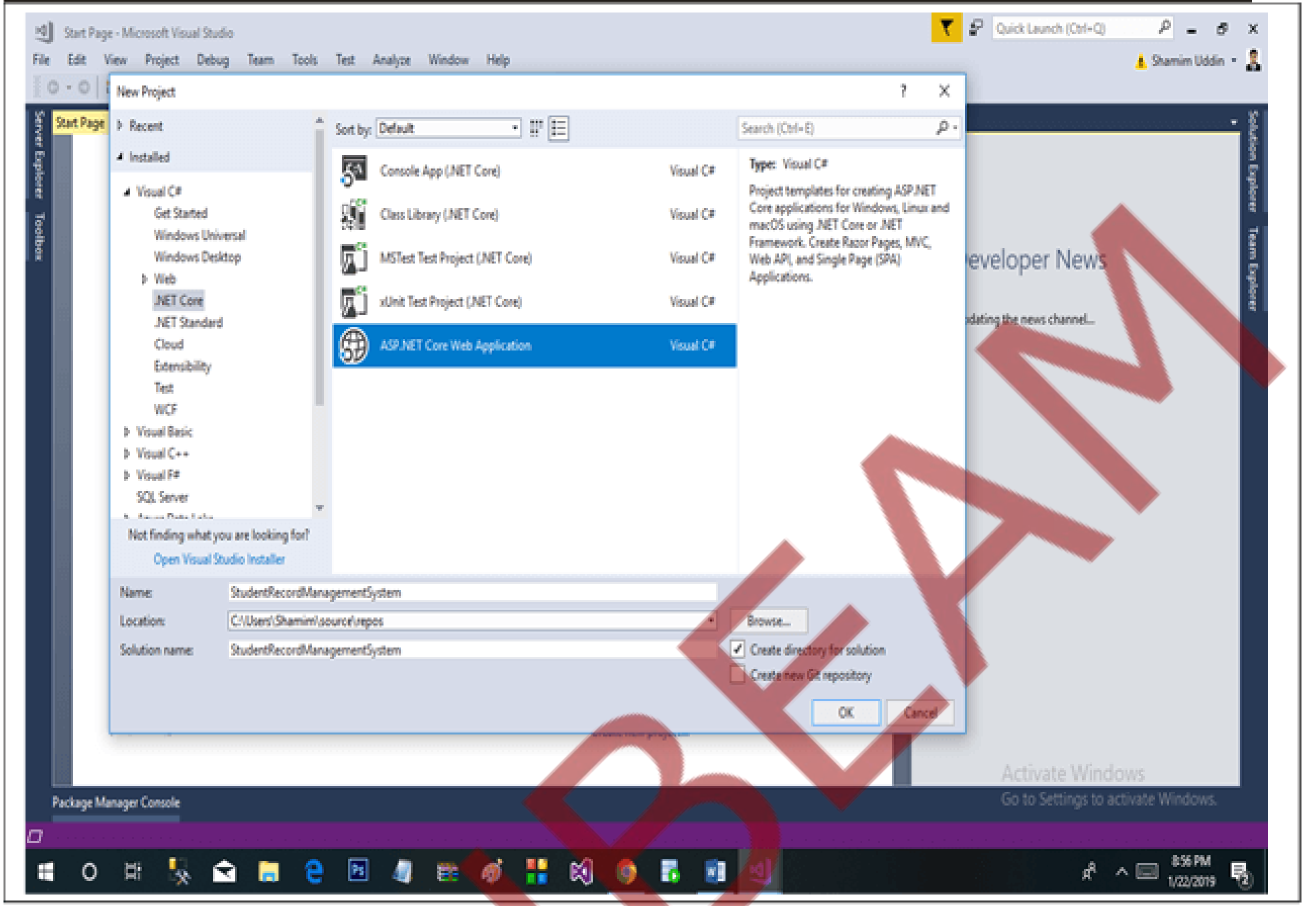
To View all Student Records

```
Create procedure spGetAllStudent
as
Begin
    select *
    from Student
    order by Id
End
```

Create the ASP.NET MVC Web Application

Now, we are going to create an ASP.NET MVC Web Application. The project name is “StudentRecordManagementSystem”.

SUNBEAM



Type: Visual C#

Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.

Name: StudentRecordManagementSystem

Location: C:\Users\Shamim\source\repos

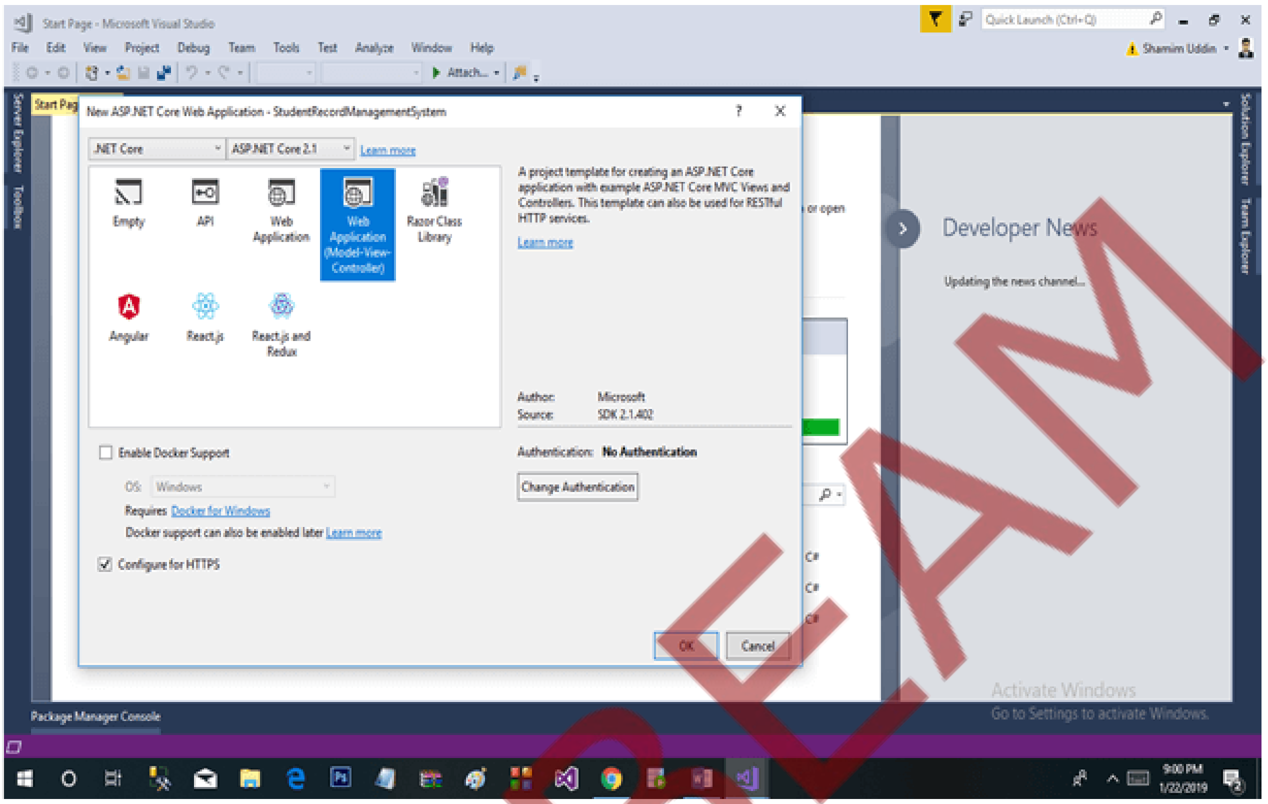
Solution name: StudentRecordManagementSystem

☑ Create directory for solution

☐ Create new Git repository

Click OK. The below window will appear on the screen.

SUNBEAM



New ASP.NET Core Web Application - StudentRecordManagementSystem

.NET Core | ASP.NET Core 2.1 | Learn more

Empty | API | Web Application | Web Application (Model-View-Controller) | Razor Class Library | Angular | React.js | React.js and Redux

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.
Learn more

Author: Microsoft
Source: SDK 2.1.402

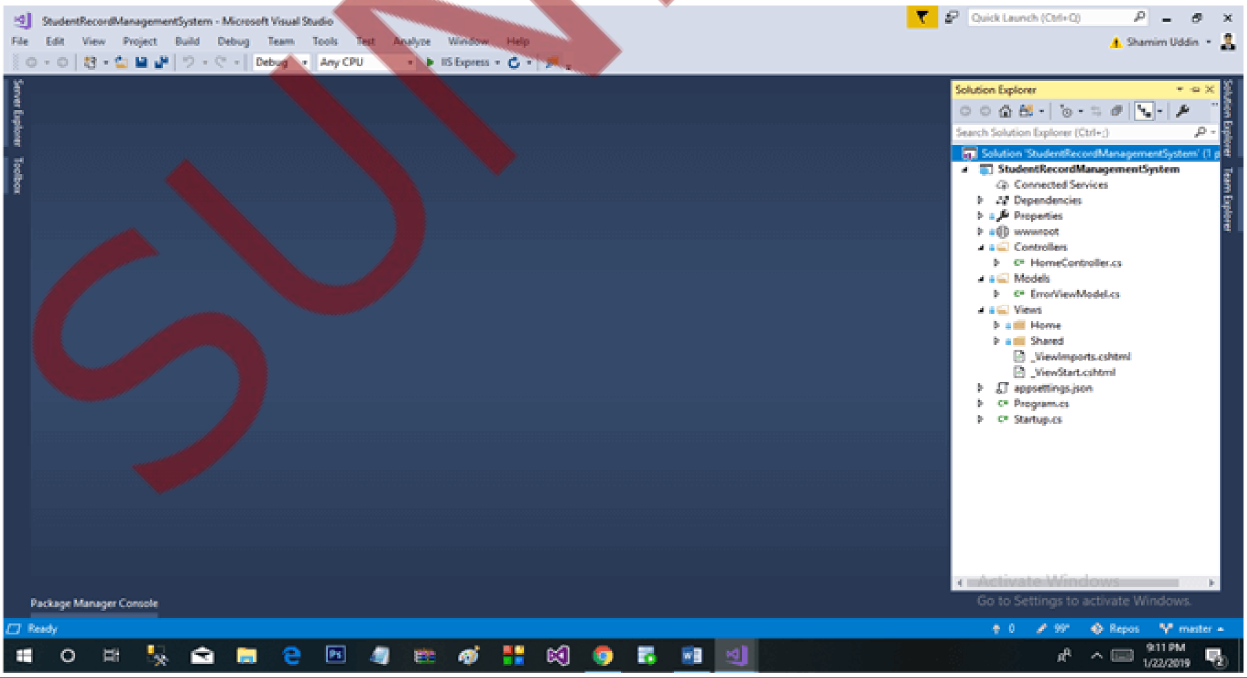
Authentication: **No Authentication**
Change Authentication

☐ Enable Docker Support
OS: Windows
Requires Docker for Windows
Docker support can also be enabled later Learn more

☑ Configure for HTTPS

OK | Cancel

Click OK. The solution creation is done with the loading of all needed files. Given below is the picture of the solution structure.



```
Solution 'StudentRecordManagementSystem' (1 p
  StudentRecordManagementSystem
    Connected Services
    Dependencies
    Properties
    wwwroot
    Controllers
      HomeController.cs
    Models
      ErrorViewModel.cs
    Views
      Home
      Shared
      _ViewImports.cshtml
      _ViewStart.cshtml
    appsettings.json
    Program.cs
    Startup.cs
```

Create Utility folder in your project

SUNBEAM

Create a folder named “Utility” in the project. Now, we will create a class named “ConnectionString” within the Utility folder.

```
public static class ConnectionString
{
    private static string cName = "Data Source=.; Initial Catalog=StudentManagement;User ID=sa;Password=123";
    public static string CName { get => cName;
 }
}
```

Creating a Model

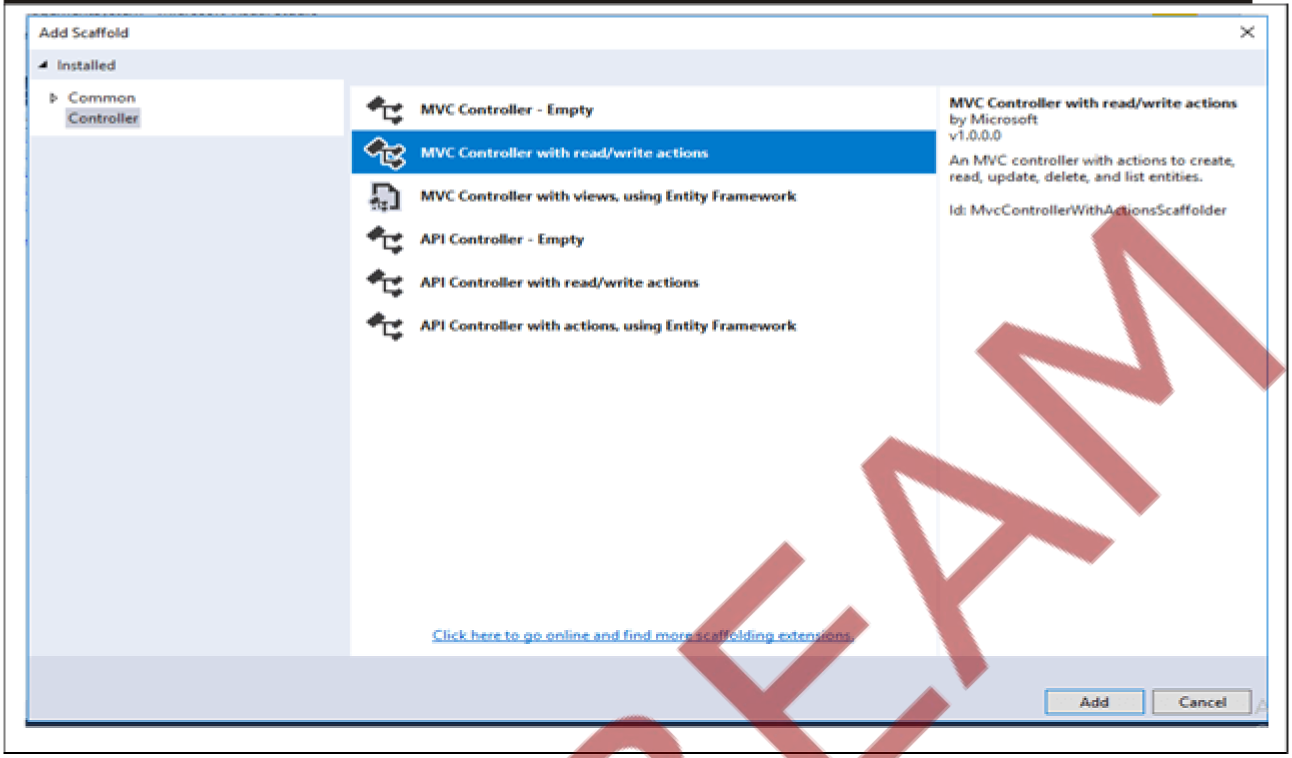
After that, we will create a class named “Student” within model folder.

```
public  class  Student
    {
        public int Id { set; get; }
        [Required]
        public string FirstName { set; get; }
        [Required]
        public string LastName { set; get; }
        [Required]
        public string Email { set; get; }
          [Required]
          public string Mobile { set; get; }
          public string Address { set; get; }
     }
```

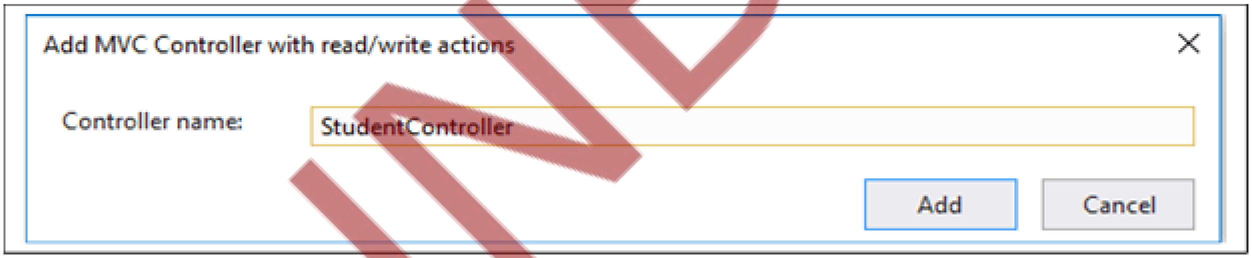
Creating a Controller

We will Rebuild our solution and create a “StudentController” within Controller folder. Right-Add-Controller-Select MVC Controller with read/write actions and click add.

SUNBEAM


Add Scaffold
Installed
Common
Controller
MVC Controller - Empty
MVC Controller with read/write actions
MVC Controller with views, using Entity Framework
API Controller - Empty
API Controller with read/write actions
API Controller with actions, using Entity Framework
MVC Controller with read/write actions
by Microsoft
v1.0.0.0
An MVC controller with actions to create, read, update, delete, and list entities.
Id: MvcControllerWithActionsScaffolder
Click here to go online and find more scaffolding extensions.
Add Cancel

Another window will appear on screen


Add MVC Controller with read/write actions
Controller name: StudentController
Add Cancel

Creating Data Access Layer

We have to work with the database so we will create a data access layer class within model folder named "StudentDataAccessLayer"

```
public class StudentDataAccessLayer
    {
       string connectionString = ConnectionString.CName;

         public IEnumerable<Student> GetAllStudent()
         {
             List<Student> lstStudent = new List<Student>();
             using (SqlConnection con = new SqlConnection(connectionSt
ring))
             {
```

SUNBEAM

```
                SqlCommand cmd = new SqlCommand("spGetAllStudent", con);
                cmd.CommandType = CommandType.StoredProcedure;
                con.Open();
                SqlDataReader rdr = cmd.ExecuteReader();

                while (rdr.Read())
                {
                    Student student = new Student();
                    student.Id = Convert.ToInt32(rdr["Id"]);
                    student.FirstName = rdr["FirstName"].ToString();
                    student.LastName = rdr["LastName"].ToString();
                    student.Email = rdr["Email"].ToString();
                    student.Mobile = rdr["Mobile"].ToString();
                    student.Address = rdr["Address"].ToString();

                    lstStudent.Add(student);
                }
                con.Close();
            }
            return lstStudent;
        }
        public void AddStudent(Student student)
        {
            using (SqlConnection con = new SqlConnection(connectionString))
            {
                SqlCommand cmd = new SqlCommand("spAddStudent", con);
                cmd.CommandType = CommandType.StoredProcedure;

                cmd.Parameters.AddWithValue("@FirstName", student.FirstName);
                cmd.Parameters.AddWithValue("@LastName", student.LastName);
                cmd.Parameters.AddWithValue("@Email", student.Email);
                cmd.Parameters.AddWithValue("@Mobile", student.Mobile);
                cmd.Parameters.AddWithValue("@Address", student.Address);
                con.Open();
                cmd.ExecuteNonQuery();
                con.Close();
            }
```

SUNBEAM

```
        }

        public void UpdateStudent(Student student)
        {
            using (SqlConnection con = new SqlConnection(connectionString))
            {
                SqlCommand cmd = new SqlCommand("spUpdateStudent", con);
                cmd.CommandType = CommandType.StoredProcedure;

                cmd.Parameters.AddWithValue("@Id", student.Id);
                cmd.Parameters.AddWithValue("@FirstName", student.FirstName);
                cmd.Parameters.AddWithValue("@LastName", student.LastName);
                cmd.Parameters.AddWithValue("@Email", student.Email);
                cmd.Parameters.AddWithValue("@Mobile", student.Mobile);
                cmd.Parameters.AddWithValue("@Address", student.Address);
                con.Open();
                cmd.ExecuteNonQuery();
                con.Close();
            }
        }

        public Student GetStudentData(int? id)
        {
            Student student = new Student();

            using (SqlConnection con = new SqlConnection(connectionString))
            {
                string sqlQuery = "SELECT * FROM Student WHERE Id= " + id;
                SqlCommand cmd = new SqlCommand(sqlQuery, con);
                con.Open();
                SqlDataReader rdr = cmd.ExecuteReader();

                while (rdr.Read())
                {
                    student.Id = Convert.ToInt32(rdr["Id"]);
                    student.FirstName = rdr["FirstName"].ToString();
```

SUNBEAM

```
                        student.LastName = rdr["LastName"].ToString();
                        student.Email = rdr["Email"].ToString();
                        student.Mobile = rdr["Mobile"].ToString();
                        student.Address = rdr["Address"].ToString();
                    }
                }
                return student;
            }

            public void DeleteStudent(int? id)
            {
                using (SqlConnection con = new SqlConnection(connectionS
tring))
                {
                    SqlCommand cmd = new SqlCommand("spDeleteStudent", c
on);
                    cmd.CommandType = CommandType.StoredProcedure;
                    cmd.Parameters.AddWithValue("@Id", id);
                    con.Open();
                    cmd.ExecuteNonQuery();
                    con.Close();
                }
            }
    }
```

Action Methods

Create Action

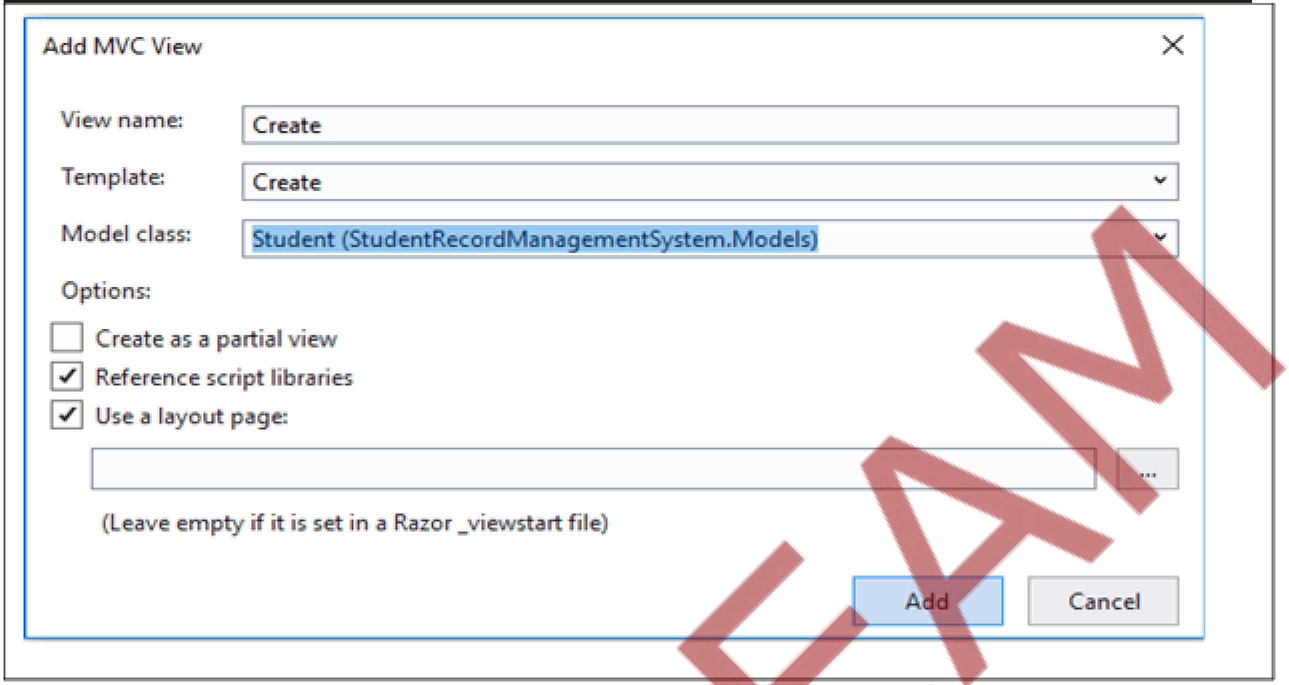
Now we will work with Create Action within Student Controller. There are two Create Actions one is GET and another is POST. Now we will create a view for creating action.

Before creating a view we will create a constructor

```
StudentDataAccessLayer studentDataAccessLayer = null;
public StudentController()
 {
   studentDataAccessLayer = new StudentDataAccessLayer();
 }
```

Right click on create (GET) action then click add view; the below window will appear on the screen.

SUNBEAM


Add MVC View

View name: Create

Template: Create

Model class: Student (StudentRecordManagementSystem.Models)

Options:

- [ ] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page:

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

Click Add

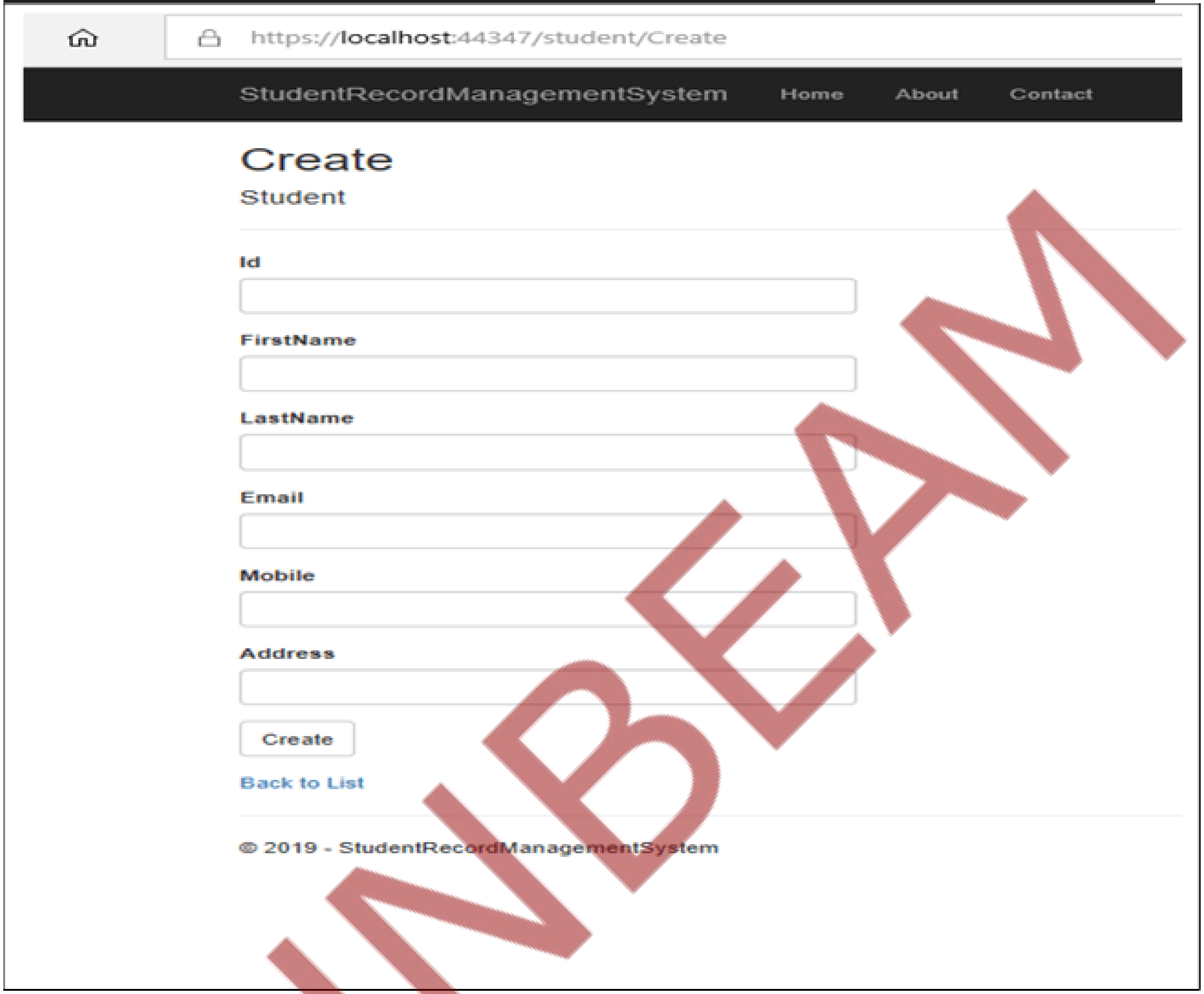
```
 @model  StudentRecordManagementSystem.Models.Student
@{
    ViewData["Title"] = "Create";
}
<h2>Create</h2>

<h4>Student</h4>
<hr />
<div class="row">
     <div class="col-md-4">
         <form asp-action="Create">
             <div asp-validation-summary="ModelOnly" class="text-
danger"></div>
             <div class="form-group">
                 <label asp-for="Id" class="control-label"></label>
                 <input asp-for="Id" class="form-control" />
                 <span asp-validation-for="Id" class="text-
danger"></span>
             </div>
             <div class="form-group">
                 <label asp-for="FirstName" class="control-
label"></label>
                 <input asp-for="FirstName" class="form-control" />
                 <span asp-validation-for="FirstName" class="text-
danger"></span>
```

SUNBEAM

```
            </div>
            <div class="form-group">
                <label asp-for="LastName" class="control-label"></label>
                <input asp-for="LastName" class="form-control" />
                <span asp-validation-for="LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Email" class="control-label"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Mobile" class="control-label"></label>
                <input asp-for="Mobile" class="form-control" />
                <span asp-validation-for="Mobile" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Address" class="control-label"></label>
                <input asp-for="Address" class="form-control" />
                <span asp-validation-for="Address" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

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

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Now run our application,

SUNBEAM


https://localhost:44347/student/Create

StudentRecordManagementSystem Home About Contact

Create

Student

Id

FirstName

LastName

Email

Mobile

Address

Create

Back to List

© 2019 - StudentRecordManagementSystem

We will remove Id field from view. We have done auto increment of Id field in database.

Now we will work with Create (POST),

```
 [HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(Student student)
{
    try
    {
        // TODO: Add insert logic here
        studentDataAccessLayer.AddStudent(student);

          return RedirectToAction(nameof(Index));
      }
      catch(Exception ex)
      {
```

SUNBEAM

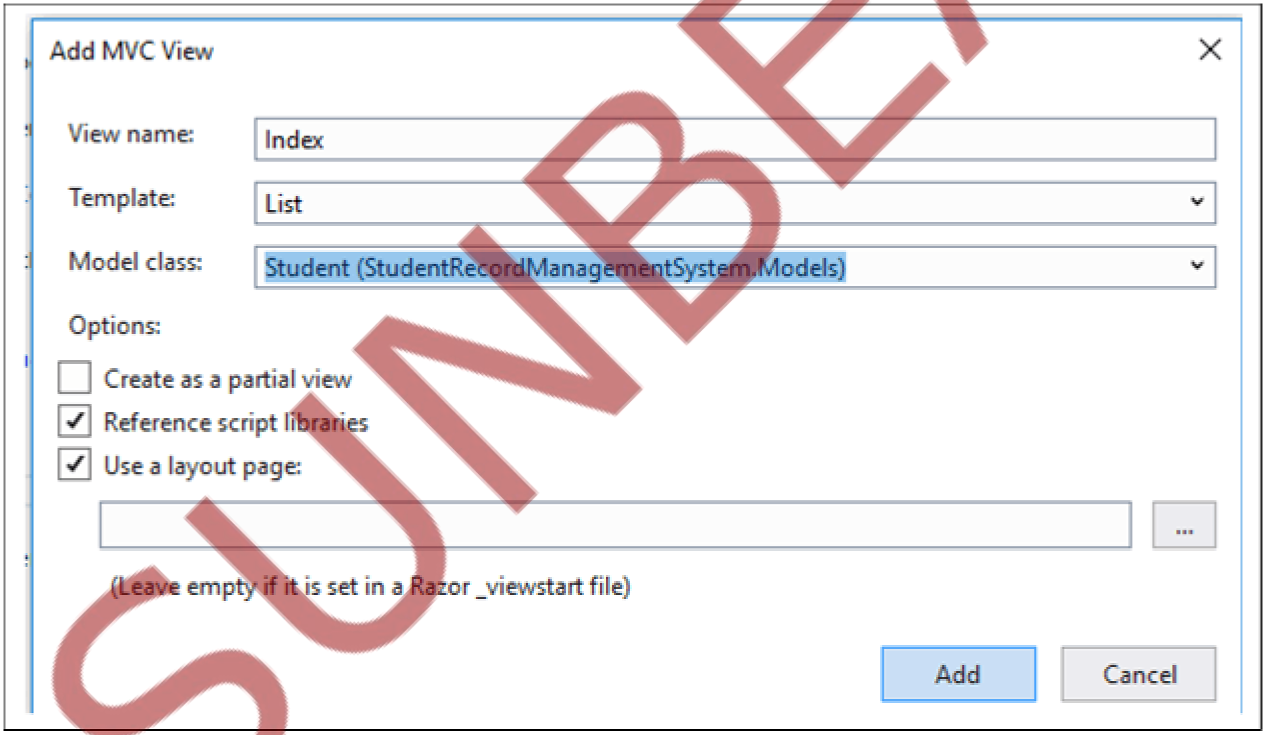
```
        return View();
    }
}
```

Index Action

We have to call GetAllStudent method from StudentDataAccessLayer class for getting all students in Index action,

```
public ActionResult Index()
        {
            IEnumerable<Student> students = studentDataAccessLayer.GetAllStudent();
            return View(students);
        }
```

Right click on Index action then click add view and the below window will appear on screen.



Add MVC View

View name: Index

Template: List

Model class: Student (StudentRecordManagementSystem.Models)

Options:

- [ ] Create as a partial view
- [x] Reference script libraries
- [x] Use a layout page:

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

Click add

```
@model IEnumerable<StudentRecordManagementSystem.Models.Student>
@{
    ViewData["Title"] = "Index";
}
<h2>Index</h2>

<p>
```

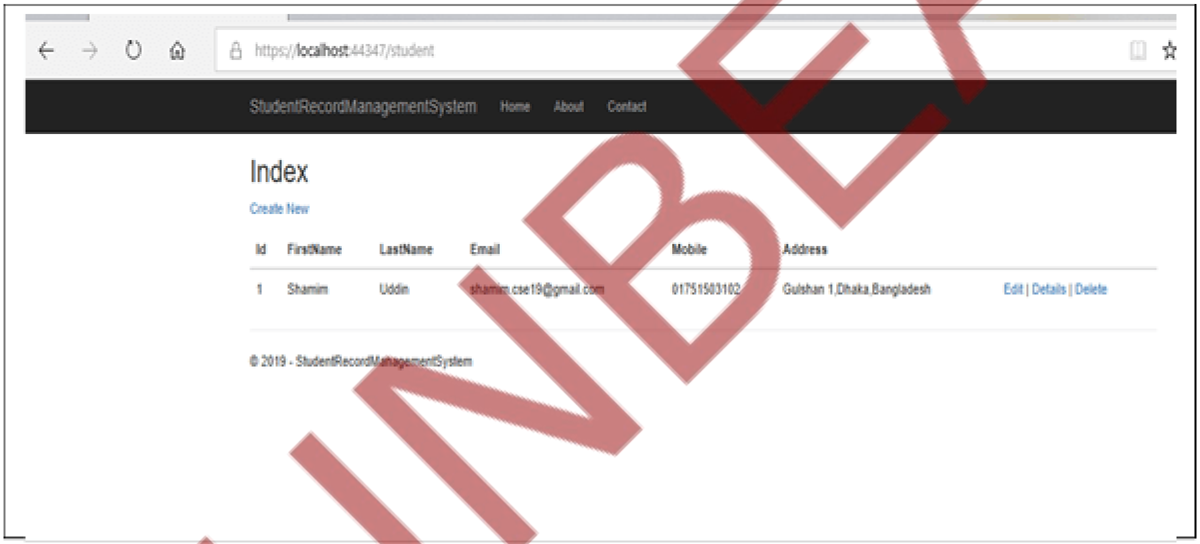
SUNBEAM

```
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Id)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.LastName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Email)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Mobile)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Address)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Id)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Email)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Mobile)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Address)
```

SUNBEAM

```
                </td>
                <td>
                    @Html.ActionLink("Edit", "Edit", new { id=item.Id  }) |
                    @Html.ActionLink("Details", "Details", new {  id=item.Id }) |
                    @Html.ActionLink("Delete", "Delete", new {  id=item.Id  })
                </td>
            </tr>
    }
        </tbody>
    </table>
```

Show student list:



https://localhost:44347/student

StudentRecordManagementSystem Home About Contact

Index

Create New

Id	FirstName	LastName	Email	Mobile	Address	
1	Shamim	Uddin	shamim.cse19@gmail.com	01751503102	Gulshan 1,Dhaka,Bangladesh	Edit | Details | Delete

© 2019 - StudentRecordManagementSystem

Edit Action

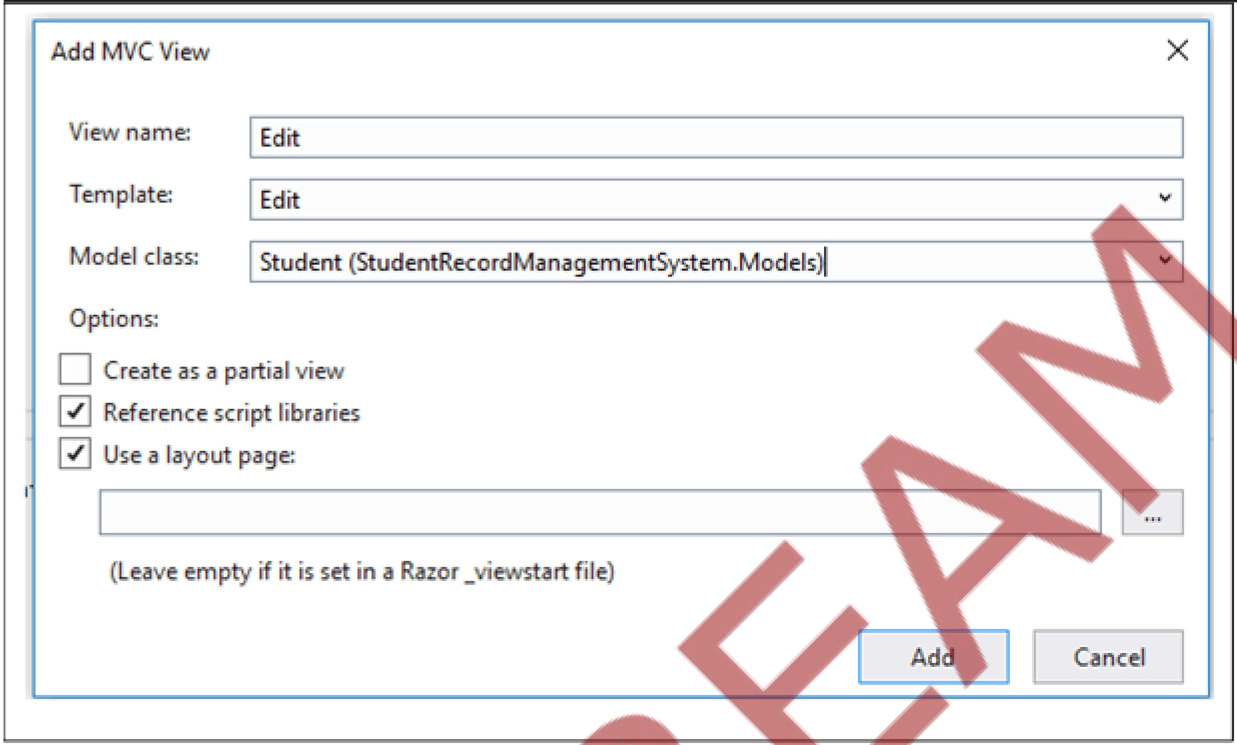
Now we will work with Edit Action within Student Controller. There are two Edit Actions; one is GET and another is POST. Now we will create a view for creating action.

We have to call GetStudentData method from StudentDataAccessLayer class for getting student by Id.

```
public ActionResult Edit(int id)
{
    Student student = studentDataAccessLayer.GetStudentData(id);
    return View(student);
}
```

Right click on Edit (GET) action then click add view; the below window will appear on screen.

SUNBEAM


Add MVC View
View name: Edit
Template: Edit
Model class: Student (StudentRecordManagementSystem.Models)
Options:
Create as a partial view
Reference script libraries
Use a layout page:
(Leave empty if it is set in a Razor _viewstart file)
Add
Cancel

Click Add

```
 @model  StudentRecordManagementSystem.Models.Student
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>

<h4>Student</h4>
<hr />
<div class="row">
       <div class="col-md-4">
              <form asp-action="Edit">
                     <div asp-validation-summary="ModelOnly" class="text-danger"></div>
                     <div class="form-group">
                            <label asp-for="Id" class="control-label"></label>
                            <input asp-for="Id" class="form-control" readonly/>
                            <span asp-validation-for="Id" class="text-danger"></span>
                     </div>
                     <div class="form-group">
                            <label asp-for="FirstName" class="control-label"></label>
                            <input asp-for="FirstName" class="form-control" />
```

SUNBEAM

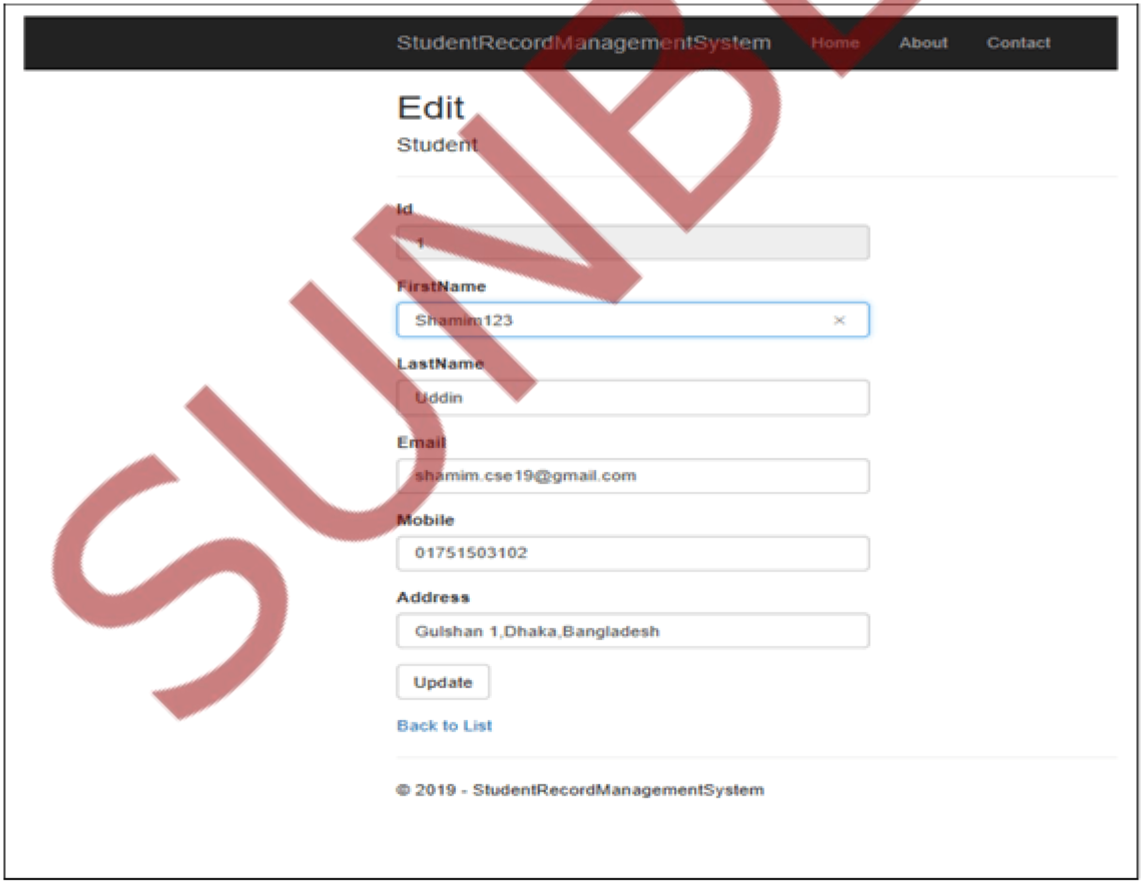
```
                        <span asp-validation-for="FirstName" class="text-
danger"></span>
                    </div>
                    <div class="form-group">
                        <label asp-for="LastName" class="control-
label"></label>
                        <input asp-for="LastName" class="form-control" />
                        <span asp-validation-for="LastName" class="text-
danger"></span>
                    </div>
                    <div class="form-group">
                        <label asp-for="Email" class="control-
label"></label>
                        <input asp-for="Email" class="form-control" />
                        <span asp-validation-for="Email" class="text-
danger"></span>
                    </div>
                    <div class="form-group">
                        <label asp-for="Mobile" class="control-
label"></label>
                        <input asp-for="Mobile" class="form-control" />
                        <span asp-validation-for="Mobile" class="text-
danger"></span>
                    </div>
                    <div class="form-group">
                        <label asp-for="Address" class="control-
label"></label>
                        <input asp-for="Address" class="form-control" />
                        <span asp-validation-for="Address" class="text-
danger"></span>
                    </div>
                    <div class="form-group">
                        <input type="submit" value="Update" class="btn btn-
default" />
                    </div>
                </form>
            </div>
        </div>

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

        @section Scripts {
            @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
        }
```

SUNBEAM

Now we will work with Edit (POST):

```
[HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Edit(Student student)
        {
            try
            {
                // TODO: Add update logic here
                studentDataAccessLayer.UpdateStudent(student);
                return RedirectToAction(nameof(Index));
             }
             catch
             {
                 return View();
             }
         }
```

Edit option has been done. Now we will test if it works or not.



Edit

Student

Id: 1

FirstName: Shamim123

LastName: Uddin

Email: shamim.cse19@gmail.com

Mobile: 01751503102

Address: Gulshan 1,Dhaka,Bangladesh

© 2019 - StudentRecordManagementSystem

SUNBEAM

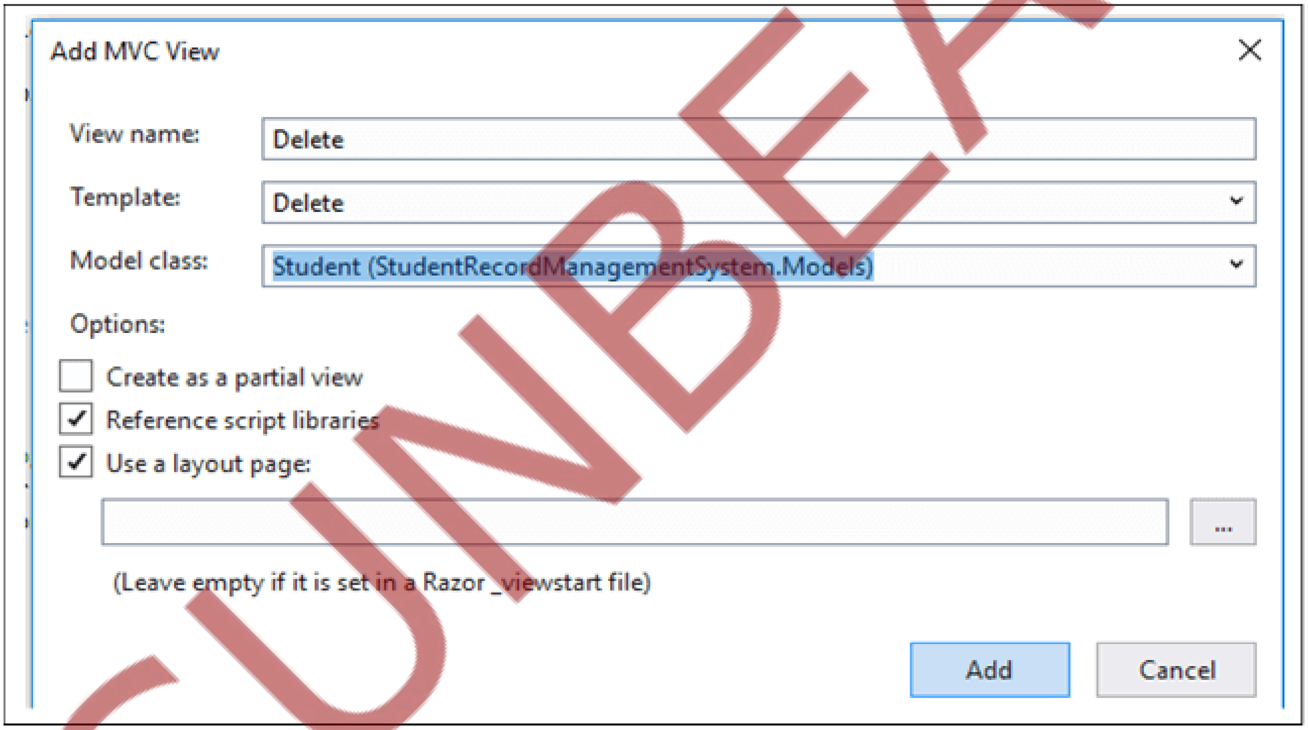
Delete Action

Now we will work with Delete Action within Student Controller. There are two Delete Actions, one is GET and another is POST. Now we will create a view for deleting action.

We have to call GetStudentData method from StudentDataAccessLayer class for getting student by Id.

```
public ActionResult Delete(int id)
        {
            Student student = studentDataAccessLayer.GetStudentData(id);
            return View(student);
        }
```

Right click on Delete (GET) action then click add view; the below window will appear on screen.


Add MVC View
View name: Delete
Template: Delete
Model class: Student (StudentRecordManagementSystem.Models)
Options:
☐ Create as a partial view
☑ Reference script libraries
☑ Use a layout page:
(Leave empty if it is set in a Razor _viewstart file)

Click Add

```
@model StudentRecordManagementSystem.Models.Student

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
```

SUNBEAM

```
    <h4>Student</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Id)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Id)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.FirstName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.FirstName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.LastName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.LastName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Email)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Email)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Mobile)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Mobile)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Address)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Address)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-action="Index">Back to List</a>
    </form>
```

SUNBEAM

```
</div>
```

Now we will work with Delete (POST)

```
[HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Delete(Student student)
        {
            try
            {
                // TODO: Add delete logic here
                studentDataAccessLayer.DeleteStudent(student.Id);
                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View();
            }
        }
```

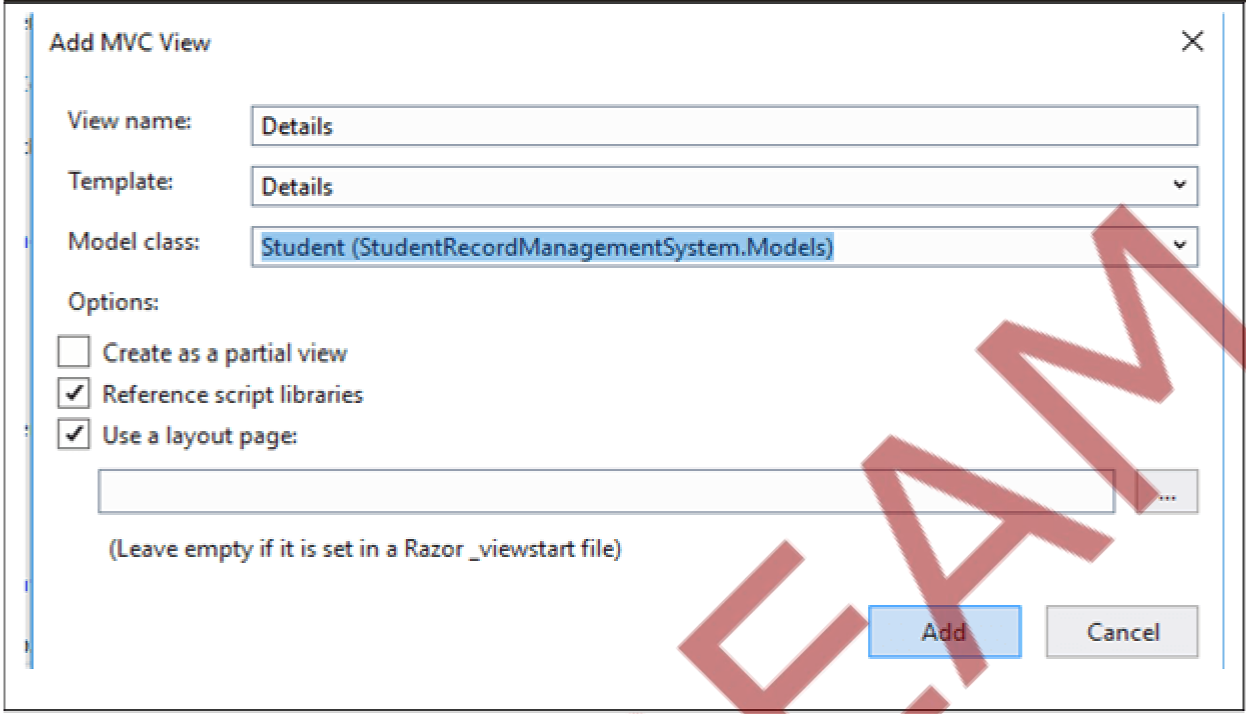
Details Action

We have to call GetStudentData method from StudentDataAccessLayer class for getting student by Id in Index action

```
public  ActionResult  Details(int  id)
        {
            Student  student  =  studentDataAccessLayer.GetStudentData(id);
            return  View(student);
        }
```

Right click on Details action then click add view; the below window will appear on screen.

SUNBEAM


Add MVC View

View name: Details

Template: Details

Model class: Student (StudentRecordManagementSystem.Models)

Options:

- [ ] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page:

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

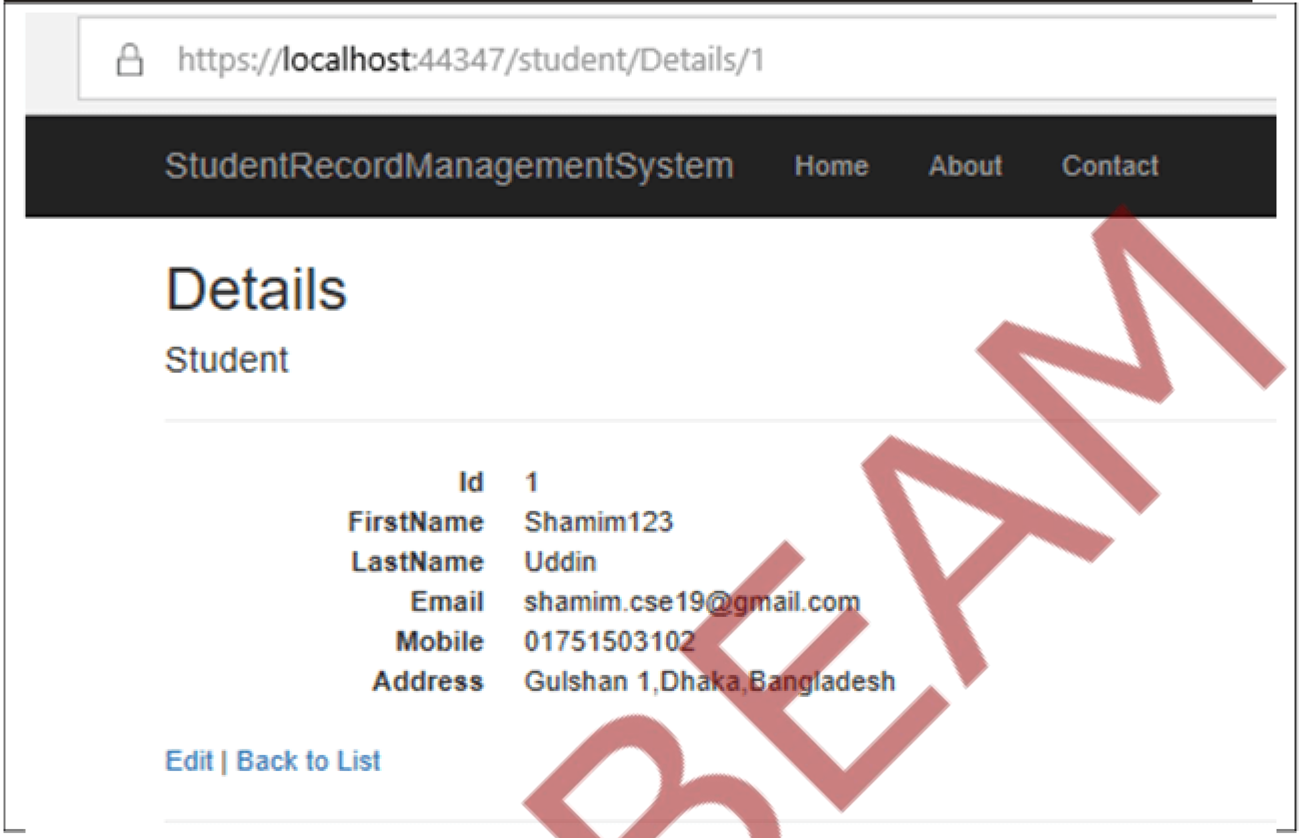
Click add

```
@model  StudentRecordManagementSystem.Models.Student
@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Id)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Id)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.FirstName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.FirstName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.LastName)
```

SUNBEAM

```
            </dt>
            <dd>
                @Html.DisplayFor(model => model.LastName)
            </dd>
            <dt>
                @Html.DisplayNameFor(model => model.Email)
            </dt>
            <dd>
                @Html.DisplayFor(model => model.Email)
            </dd>
            <dt>
                @Html.DisplayNameFor(model => model.Mobile)
            </dt>
            <dd>
                @Html.DisplayFor(model => model.Mobile)
            </dd>
            <dt>
                @Html.DisplayNameFor(model => model.Address)
            </dt>
            <dd>
                @Html.DisplayFor(model => model.Address)
            </dd>
        </dl>
</div>
<div>
    @Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
    <a asp-action="Index">Back to List</a>
</div>
```

SUNBEAM


https://localhost:44347/student/Details/1

Details

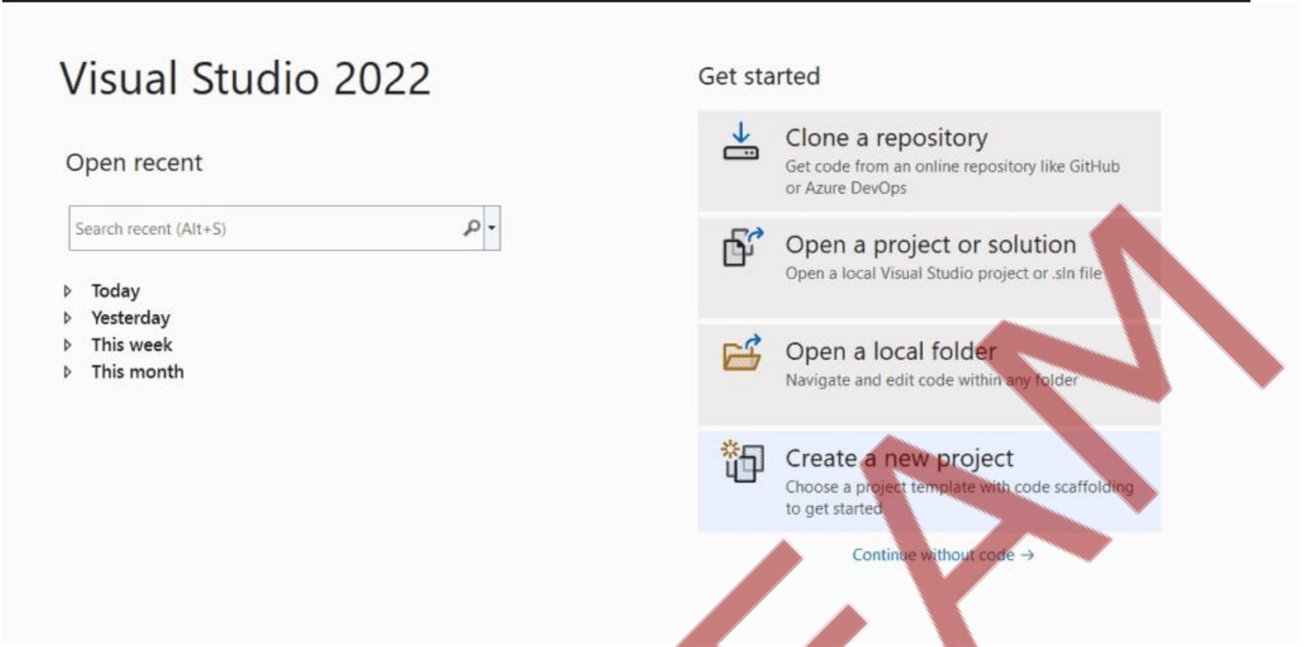
Student

Id	1
FirstName	Shamim123
LastName	Uddin
Email	shamim.cse19@gmail.com
Mobile	01751503102
Address	Gulshan 1,Dhaka,Bangladesh

SUNBEAM

ASP.NET Core MVC CRUD Operations using EF

Open Visual Studio and click on Create New Project



Visual Studio 2022

Open recent

Search recent (Alt+S)

- Today
- Yesterday
- This week
- This month

Get started

Clone a repository
Get code from an online repository like GitHub or Azure DevOps

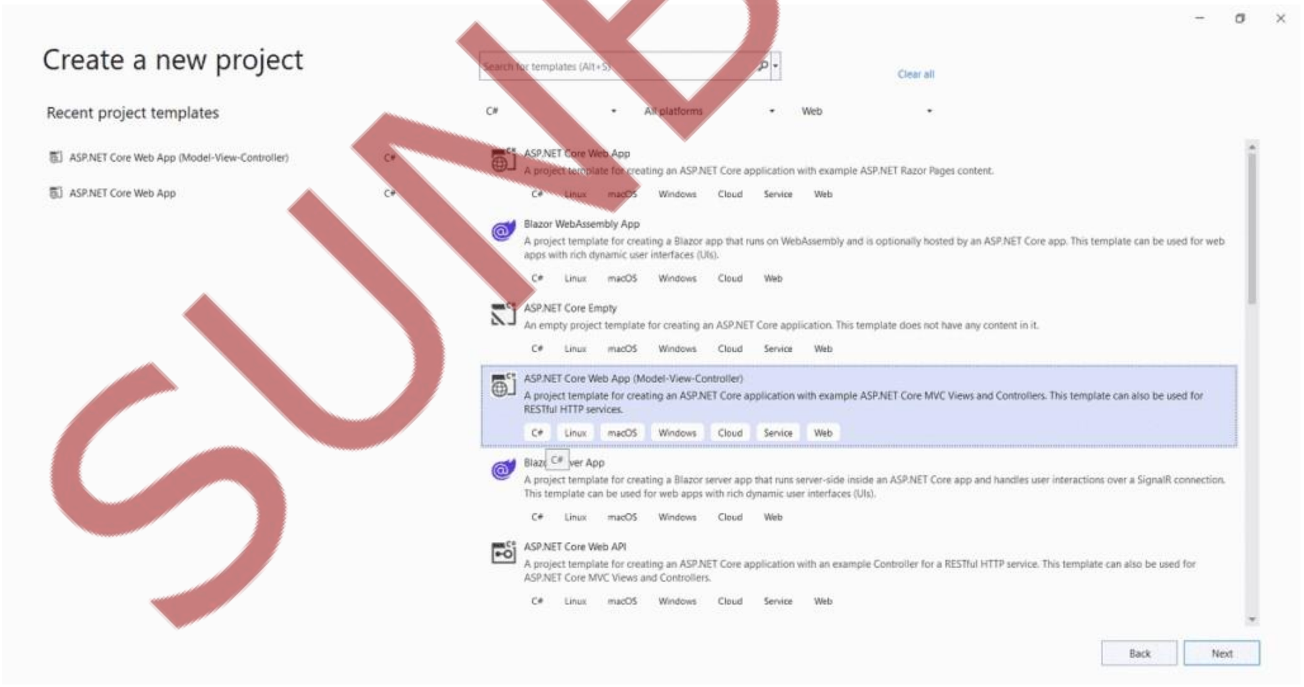
Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

Create a new project
Choose a project template with code scaffolding to get started

Continue without code →

Select ASP.NET Core Web App (Model-View-Controller) – [C#] and click next button



Create a new project

Search for templates (Alt+S)

Clear all

Recent project templates

- ASP.NET Core Web App (Model-View-Controller) C#
- ASP.NET Core Web App C#

C# | All platforms | Web

ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.
C# Linux macOS Windows Cloud Service Web

Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly and is optionally hosted by an ASP.NET Core app. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
C# Linux macOS Windows Cloud Service Web

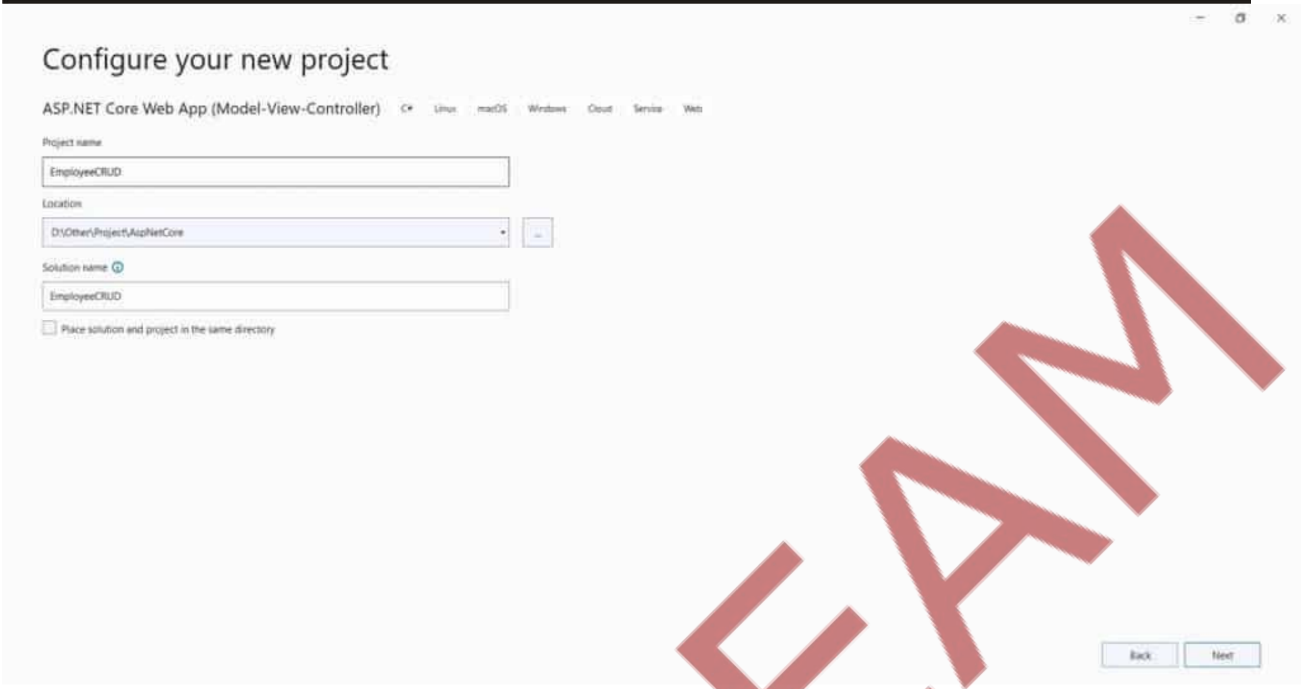
ASP.NET Core Web App (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.
C# Linux macOS Windows Cloud Service Web

Blazor Server App
A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

ASP.NET Core Web API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
C# Linux macOS Windows Cloud Service Web

Enter the project name and click on next button

SUNBEAM



Configure your new project

ASP.NET Core Web App (Model-View-Controller)

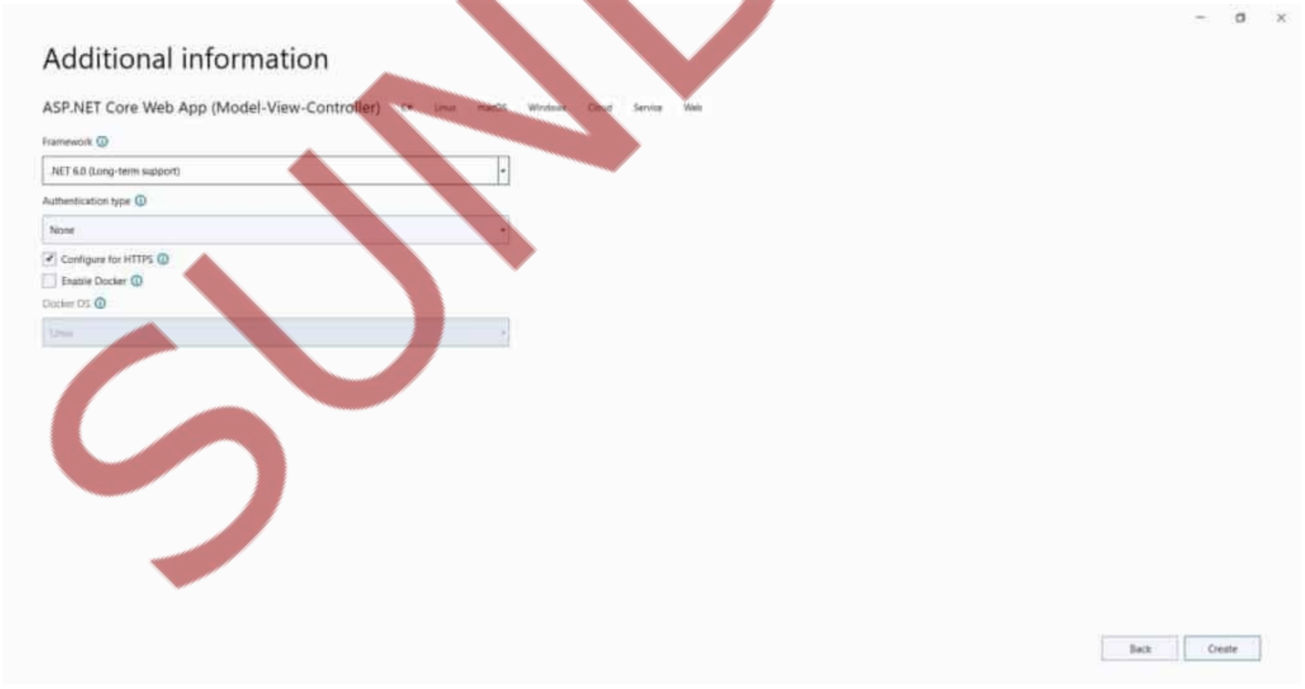
Project name: EmployeeCRUD

Location: D:\Other\Project\AspNetCore

Solution name: EmployeeCRUD

☐ Place solution and project in the same directory

Select .Net 6.0 , authentication type None and click on create button



Additional information

ASP.NET Core Web App (Model-View-Controller)

Framework: .NET 6.0 (Long-term support)

Authentication type: None

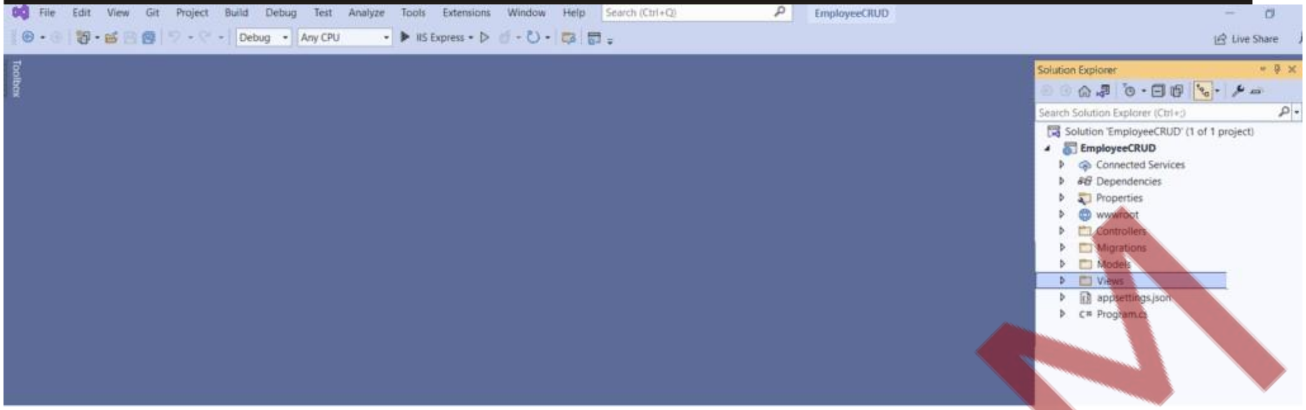
☑ Configure for HTTPS

☐ Enable Docker

Docker OS: Linux

Once created, your project will look like following

SUNBEAM



Open Models folder and create an Employee class

Enter the following code in the employee class

```
using System.ComponentModel.DataAnnotations;
namespace EmployeeCRUD.Models
{
    public class Employee
    {
        [Key]
        public int Id { get; set; }
        [Required]
        [Display(Name ="Employee Name")]
        public string Name { get; set; }
        public string Designation { get; set; }
        [DataType(DataType.MultilineText)]
        public string Address { get; set; }
        public DateTime? RecordCreatedOn { get; set; }
    }
}
```

Install the following packages according to your .NET Core version

```
<PackageReference Include="Microsoft.EntityFrameworkCore"
Version="6.0.1" />
```

SUNBEAM

```
    <PackageReference
Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.1"
/>
    <PackageReference
Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.1">
```

Create a class ApplicationDbContext of DBContext Class to link the database with data model class

```
using EmployeeCRUD.Models;
using Microsoft.EntityFrameworkCore;

namespace EmployeeCRUD.Data
{
    public class ApplicationDbContext:DbContext
    {
        public
ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options):base(options)
        {

        }

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

Open appsettings.json and configure the connection string

```
{
    "ConnectionStrings": {
      "DefaultConnection": "Server=EnterServerName;
Database=EmployeeDatabase; User Id=sa; Password=EnterPassword;
Trusted_Connection=True; MultipleActiveResultSets=true"
    },
```

SUNBEAM

```
    "Logging": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft.AspNetCore": "Warning"
      }
    },
    "AllowedHosts": "*"
  }
```

Open the program.cs file and add the required services

```
using EmployeeCRUD.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

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

// add
builder.Services.AddDbContext<ApplicationDbContext>(
    options => options.UseSqlServer(

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

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
```

SUNBEAM

```
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

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

app.Run();
```

Run the migration

Open the package manager console and execute following commands

- add-migration 'initial'
- update-database

Add Employee Controller

1. Right click on Controllers folder
2. click on add
3. click on empty controller and name file as EmployeeController.cs file

```
using EmployeeCRUD.Data;
using EmployeeCRUD.Models;
using Microsoft.AspNetCore.Mvc;

namespace EmployeeCRUD.Controllers
{
    public class EmployeeController : Controller
    {
        private readonly ApplicationDbContext _context;
        public EmployeeController(ApplicationDbContext context)
        {
            _context = context;
        }
        public IActionResult Index()
```

SUNBEAM

```
        {
            IEnumerable<Employee> objCatlist =
_context.Employees;
            return View(objCatlist);
        }

        public IActionResult Create()
        {
            return View();
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public IActionResult Create(Employee empobj)
        {
            if (ModelState.IsValid)
            {
                _context.Employees.Add(empobj);
                _context.SaveChanges();
                TempData["ResultOk"] = "Record Added Successfully
!";
                return RedirectToAction("Index");
            }

            return View(empobj);
        }

        public IActionResult Edit(int? id)
        {
            if (id == null || id == 0)
            {
                return NotFound();
            }
            var empfromdb = _context.Employees.Find(id);

            if (empfromdb == null)
```

SUNBEAM

```
            {
                return NotFound();
            }
            return View(empfromdb);
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public IActionResult Edit(Employee empobj)
        {
            if (ModelState.IsValid)
            {
                _context.Employees.Update(empobj);
                _context.SaveChanges();
                TempData["ResultOk"] = "Data Updated Successfully
!";
                return RedirectToAction("Index");
            }

            return View(empobj);
        }

        public IActionResult Delete(int? id)
        {
            if (id == null || id == 0)
            {
                return NotFound();
            }
            var empfromdb = _context.Employees.Find(id);

            if (empfromdb == null)
            {
                return NotFound();
            }
            return View(empfromdb);
        }
```

SUNBEAM

```
        [HttpPost]
        [ValidateAntiForgeryToken]
        public IActionResult DeleteEmp(int? id)
        {
            var deleterecord = _context.Employees.Find(id);
            if (deleterecord == null)
            {
                return NotFound();
            }
            _context.Employees.Remove(deleterecord);
            _context.SaveChanges();
            TempData["ResultOk"] = "Data Deleted Successfully !";
            return RedirectToAction("Index");
        }
    }
}
```

Add view files

Create one Employee folder under View folder and create the following files

1. Index.cshtml
2. Create.cshtml
3. Edit.cshtml
4. Delete.cshtml

Code of Index.cshtml

```
@model IEnumerable<Employee>

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

@if (TempData["ResultOk"] != null)
{
    <h1 class="alert-success">@TempData["ResultOk"]</h1>
}
```

SUNBEAM

```
<div class="container shadow p-5">

    <h1 class="text-center mb-3">CRUD Operations Using .NET Core
6 & Microsoft.EntityFrameworkCore </h1>

    <div class="col mb-3">
        <a asp-controller="Employee" asp-action="Create"
class="btn btn-lg btn-primary"><i class="bi bi-file-plus-
fill"></i>Add Employee</a>
    </div>
    <table class="table table-bordered table-hover">
        <thead>
            <tr>
                <th scope="col">Employee Name</th>
                <th scope="col">Designation</th>
                <th scope="col">Address</th>
                <th scope="col">CreatedOn</th>
                <th></th>
            </tr>
        </thead>
        <tbody>

            @foreach (var item in Model)
            {
                <tr>
                    <td width="20%">
                        @item.Name
                    </td>
                    <td width="20%">
                        @item.Designation
                    </td>
                    <td width="25%">
                        @item.Address
                    </td>
                    <td width="20%">
```

SUNBEAM

```
                        @item.RecordCreatedOn
                    </td>
                    <td>
                        <div role="group" class="w-60 btn-group">
                            <a asp-controller="Employee" asp-
action="Edit" asp-route-id="@item.Id" class=" btn btn-sm btn-
primary"><i class="bi bi-pencil-square"></i>Edit</a>&nbsp;
                            <a asp-controller="Employee" asp-
action="Delete" asp-route-id="@item.Id" class="btn btn-sm btn-
danger"><i class="bi bi-trash-fill"></i>Delete</a>
                        </div>
                    </td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

Code of Create.cshtml

```
@model Employee
<div class="container shadow p-5">
    <div class="row pb-2">
        <h2>Add Employee</h2>
    </div>
    <form method="post">
        <div asp-validation-summary="All"></div>
        <div class="form-row">
            <div class="form-group col-md-6">
                <label asp-for="Name">Employee Name</label>
                <input type="text" class="form-control mb-3" asp-
for="Name" placeholder="Enter Name">
                <span asp-validation-for="Name" class=" alert-
danger"></span>
            </div>
            <div class="form-group col-md-6">
                <label asp-for="Designation">Designation</label>
                <input type="text" class="form-control mb-3" asp-
```

SUNBEAM

```
for="Designation" placeholder="Enter Designation">
                <span asp-validation-for="Designation" class="
alert-danger"></span>
            </div>
        </div>
        <div class="form-row">
            <div class="form-group col-md-6">
                <label asp-for="Address">Address</label>
                <input type="text" class="form-control mb-3" asp-
for="Address" placeholder="Enter Address">
                <span asp-validation-for="Address" class=" alert-
danger"></span>
            </div>
            <div class="form-group col-md-6 mb-3">
                <label asp-for="RecordCreatedOn">Created
On</label>
                <input type="datetime-local" class="form-control"
asp-for="RecordCreatedOn">
                <span asp-validation-for="RecordCreatedOn"
class=" alert-danger"></span>
            </div>
        </div>
        <button type="submit" class="btn btn-lg btn-primary p-
2"><i class="bi bi-file-plus-fill"></i>Save</button>
        <a asp-controller="Employee" asp-action="Index"
class="btn btn-lg btn-warning p-2">Back To List</a>
    </form>
</div>
@section Scripts{
    @{
    <partial name="_ValidationScriptsPartial" />
    }
}
```

Code of Edit.cshtml

```
@model Employee
<div class="container shadow p-5">
```

SUNBEAM

```
    <div class="row pb-2">
        <h2>Edit Employee</h2>
    </div>
    <form method="post" asp-action="Edit">
        <div asp-validation-summary="All"></div>
        <div class="form-row">
            <div class="form-group col-md-6">
                <label asp-for="Name">Employee Name</label>
                <input type="text" class="form-control mb-3" asp-
for="Name">
                <span asp-validation-for="Name" class=" alert-
danger"></span>
            </div>
            <div class="form-group col-md-6">
                <label asp-for="Designation">Designation</label>
                <input type="text" class="form-control mb-3" asp-
for="Designation">
                <span asp-validation-for="Designation" class="
alert-danger"></span>
            </div>
        </div>
        <div class="form-row">
            <div class="form-group col-md-6">
                <label asp-for="Address">Address</label>
                <input type="text" class="form-control mb-3" asp-
for="Address">
                <span asp-validation-for="Address" class=" alert-
danger"></span>
            </div>
            <div class="form-group col-md-6 mb-3">
                <label asp-for="RecordCreatedOn">Created
On</label>
                <input type="datetime-local" class="form-control"
asp-for="RecordCreatedOn">
                <span asp-validation-for="RecordCreatedOn"
class=" alert-danger"></span>
```

SUNBEAM

```
                </div>
            </div>
            <button type="submit" class="btn btn-lg btn-primary p-
2"><i class="bi bi-file-plus-fill"></i>Update</button>
            <a asp-controller="Employee" asp-action="Index"
class="btn btn-lg btn-warning p-2">Back To List</a>
        </form>
</div>
@section Scripts{
    @{
    <partial name="_ValidationScriptsPartial" />
    }
}
```

Code of Delete.cshtml

```
@model Employee
<div class="container shadow p-5">
    <div class="row pb-2">
        <h2>Delete Employee</h2>
    </div>
    <form method="post" asp-action="DeleteEmp">
        <input asp-for="Id" hidden />
        <div asp-validation-summary="All"></div>
        <div class="form-row">
            <div class="form-group col-md-6">
                <label asp-for="Name">Employee Name</label>
                <input type="text" class="form-control mb-3" asp-
for="Name" disabled>
                <span asp-validation-for="Name" class=" alert-
danger"></span>
            </div>
            <div class="form-group col-md-6">
                <label asp-for="Designation">Designation</label>
                <input type="text" class="form-control mb-3" asp-
for="Designation" disabled>
                <span asp-validation-for="Designation" class="
alert-danger"></span>
```

SUNBEAM

```
            </div>
        </div>
        <div class="form-row">
            <div class="form-group col-md-6">
                <label asp-for="Address">Address</label>
                <input type="text" class="form-control mb-3" asp-
for="Address" disabled>
                <span asp-validation-for="Address" class=" alert-
danger"></span>
            </div>
            <div class="form-group col-md-6 mb-3">
                <label asp-for="RecordCreatedOn">Created
On</label>
                <input type="datetime-local" class="form-control"
asp-for="RecordCreatedOn" disabled>
                <span asp-validation-for="RecordCreatedOn"
class=" alert-danger"></span>
            </div>
        </div>
        <button type="submit" class="btn btn-lg btn-danger p-
2"><i class="bi bi-trash-fill"></i>Delete</button>
        <a asp-controller="Employee" asp-action="Index"
class="btn btn-lg btn-warning p-2">Back To List</a>
    </form>

</div>

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

SUNBEAM