Lab Manual for Cloud Computing

Lab No. 6

REST based CRUD operations with ASP.NET Web API

LAB 06: REST BASED CRUD OPERATIONS WITH ASP.NET WEB API

1. INTRODUCTION:

ASP.NET Web API stands as a cornerstone within the .NET ecosystem, offering developers a robust framework for crafting HTTP services that adhere to RESTful principles. Built upon the bedrock of Representational State Transfer (REST), it streamlines the development of APIs characterized by their scalability, maintainability, and adherence to industry best practices. By standardizing the use of HTTP methods such as GET, POST, PUT, and DELETE for CRUD operations, ASP.NET Web API ensures APIs are easily navigable and comprehensible, fostering seamless integration across a diverse array of client devices ranging from web browsers to mobile applications and desktop software.

A key strength of ASP.NET Web API lies in its ability to transcend platform and device boundaries, offering compatibility with a wide range of clients including web browsers, mobile apps across iOS and Android platforms, desktop software, and even Internet of Things (IoT) devices. This versatility empowers developers to create APIs that cater to a broad audience, enhancing accessibility and promoting interoperability. Moreover, ASP.NET Web API seamlessly integrates with other Microsoft technologies and frameworks, enabling developers to leverage existing skills and infrastructure for increased productivity and efficiency in API development.

In this lab, we will see how to create a simple Web API with all CRUD operations and will connect with an existing MS SQL database. After that, we will create an MVC application andconsume this Web API for CRUD actions

Step 1: Create "Employees" table in MSSQL database

In this Lab, we will see how to create an Employee data entry application. So, we need to create an "Employees" table first.

- Creating database named "CC_Lab_06".

```
create database CC_Lab_06;
```

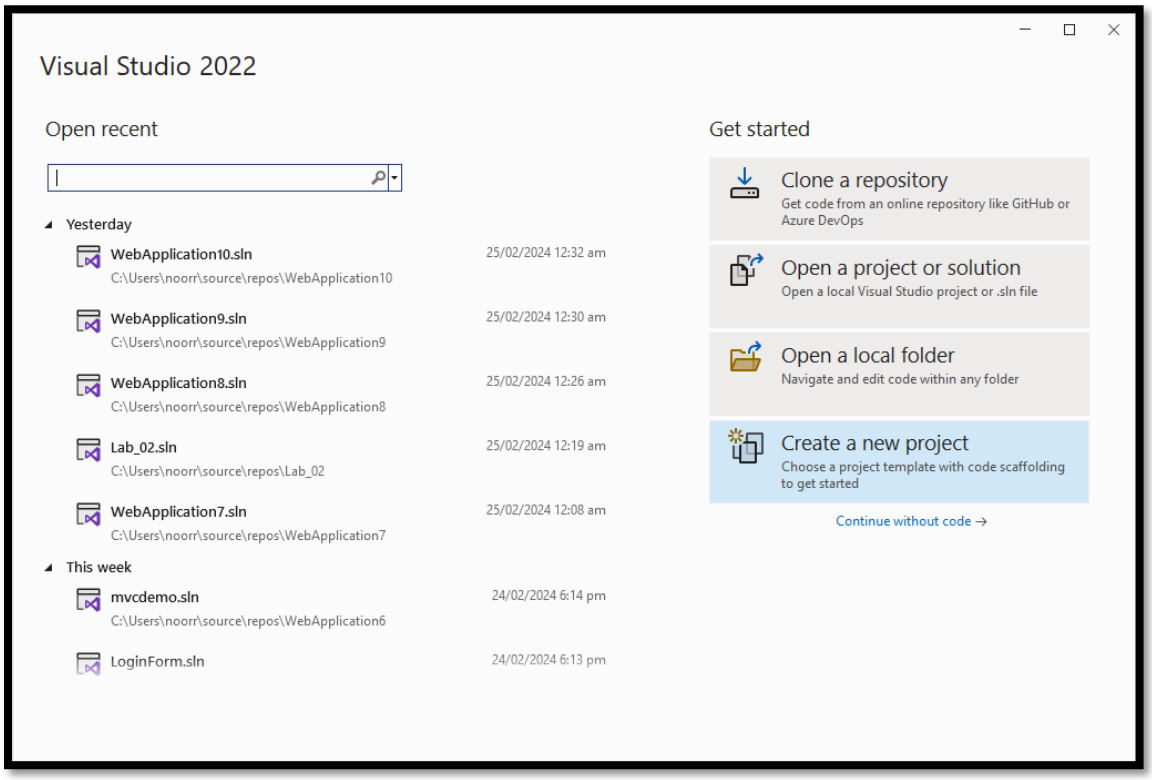
- Using the created database and then design tabel named as Student.

```
use CC_Lab_06;

create table Employees(
   Id varchar(50) not null primary key,
   Name varchar(50),
   Address varchar(50),
   Gender varchar(10),
   Company varchar(50),
   Designation varchar(50));
```

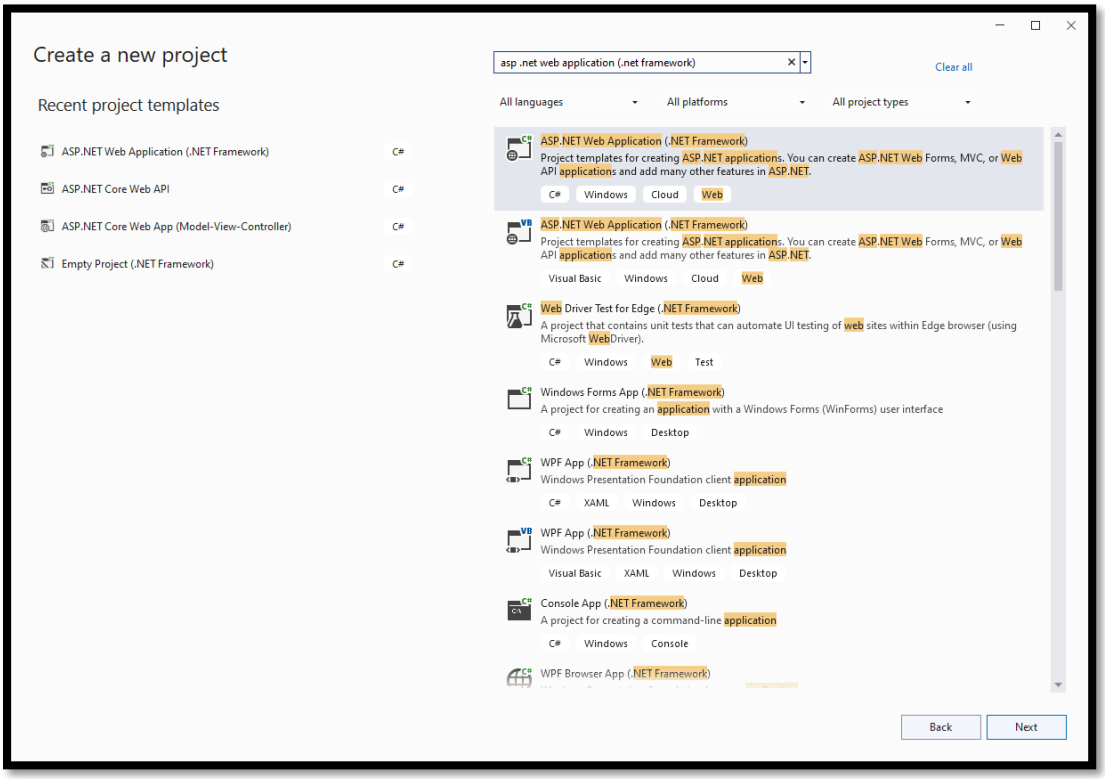
Step 2: Create a new project in Visual Studio 2022

- Open Visual Studio 2022.

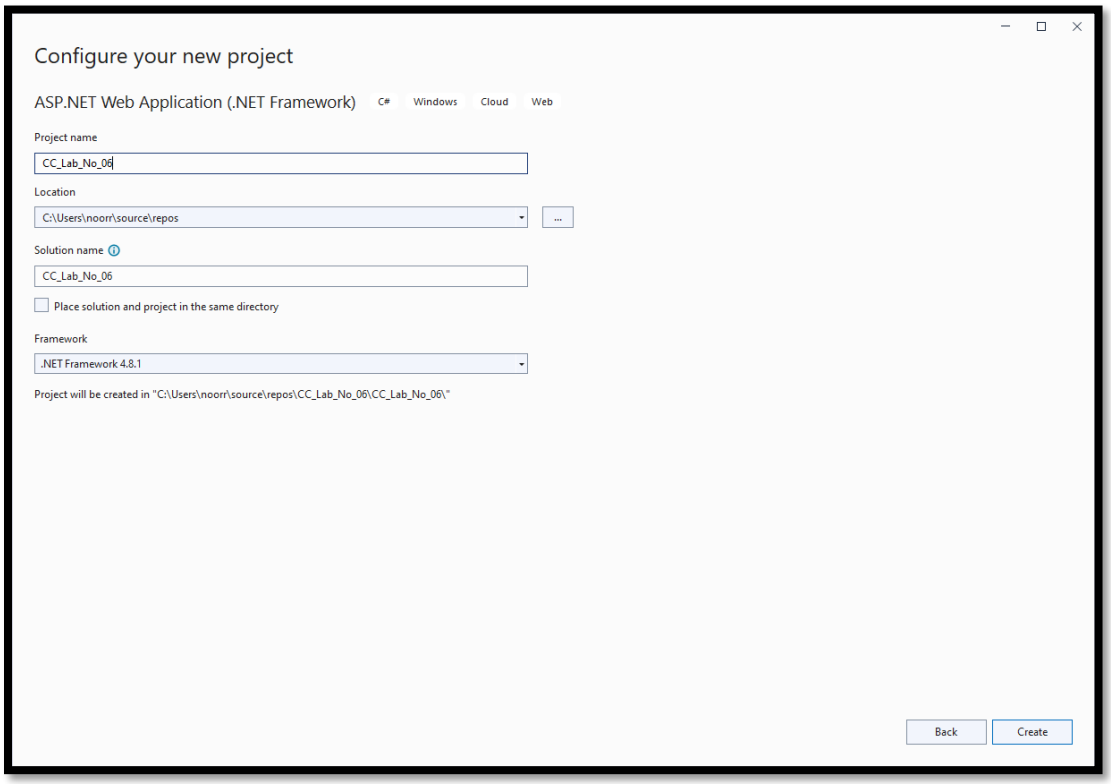
- Click on "Create a new project" in the start window.



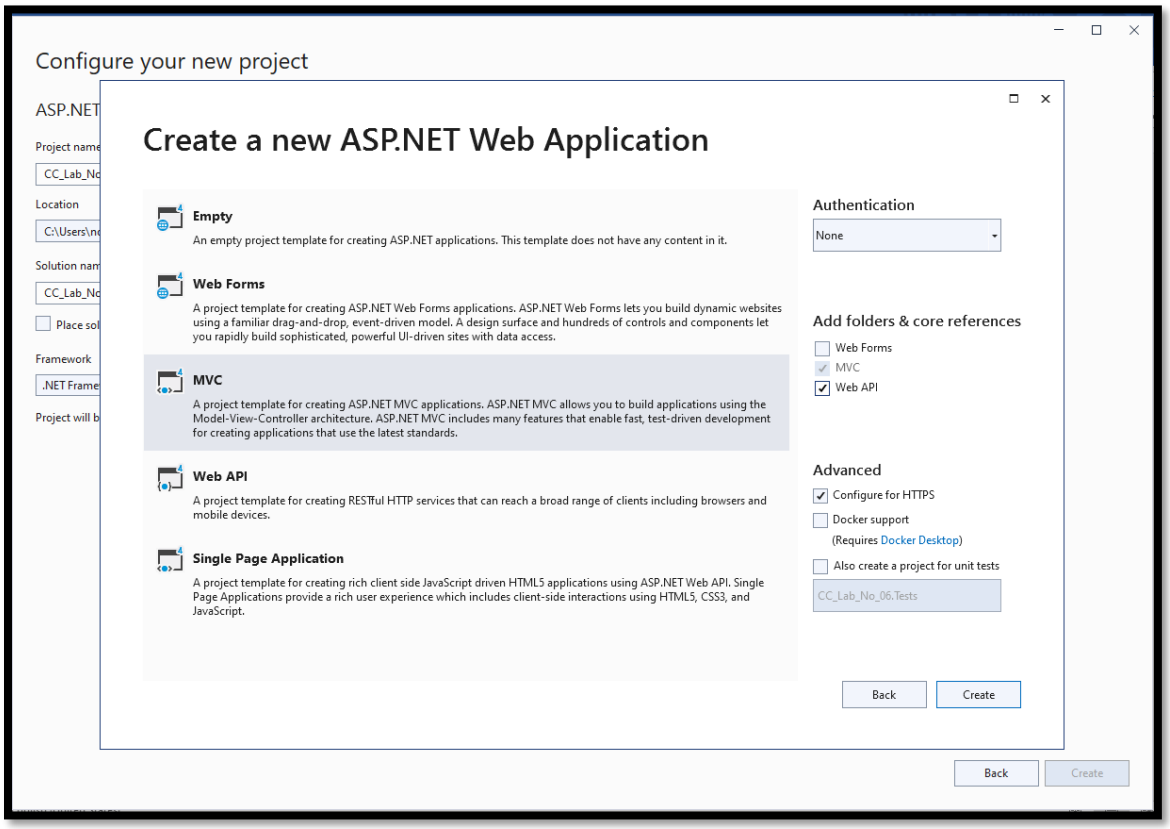
- In the "Create a new project" window, search for **ASP.NET Web Application (.net framework)** in the search bar, and then click NEXT.

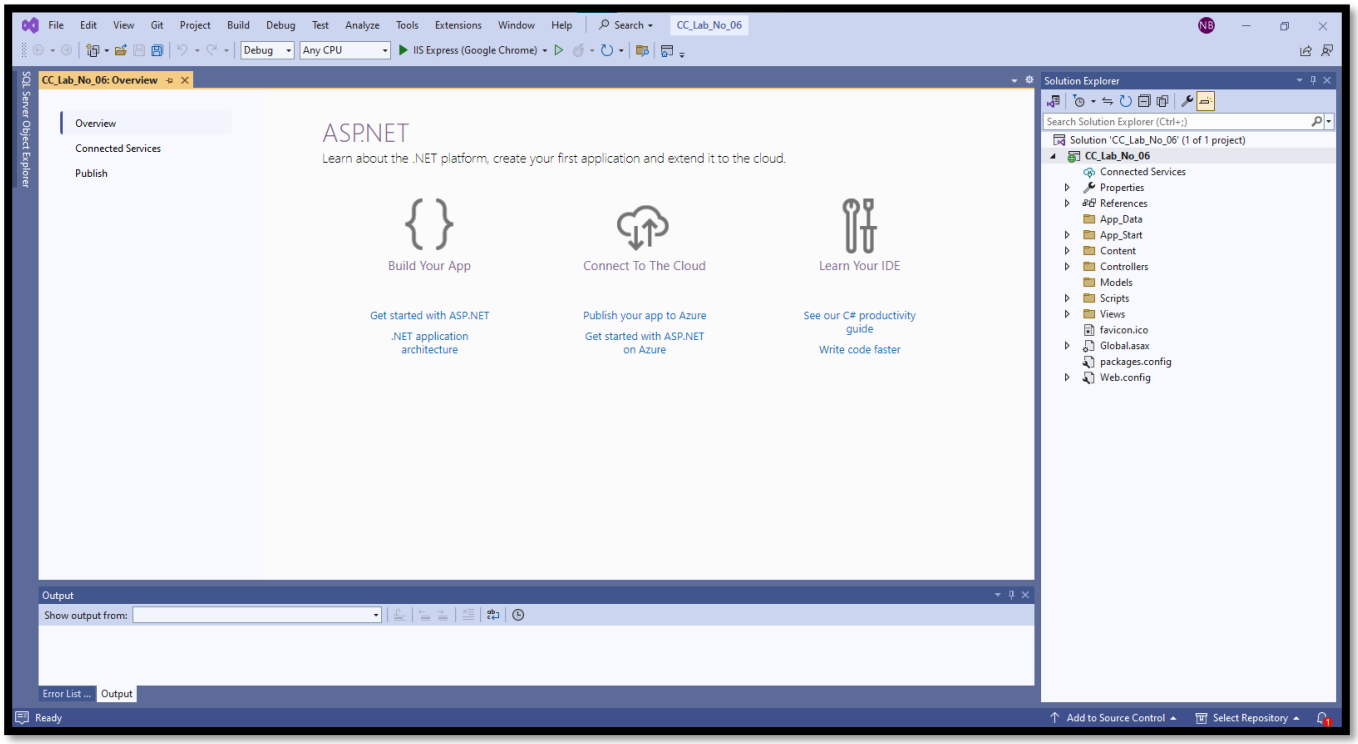


- Provide a name and location for your project and then Click NEXT.



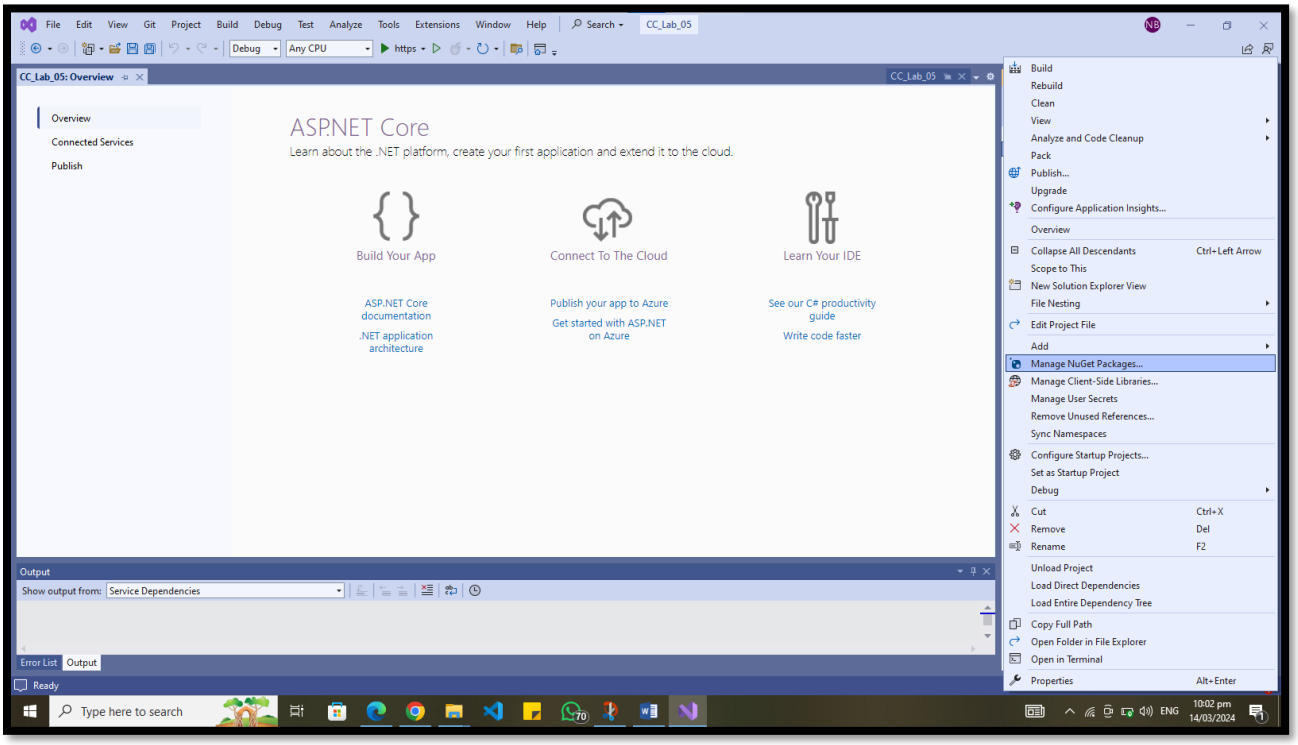
- Select the **MVC template** and check **web API** option from the list of "add folders and core references", and Click CREATE.



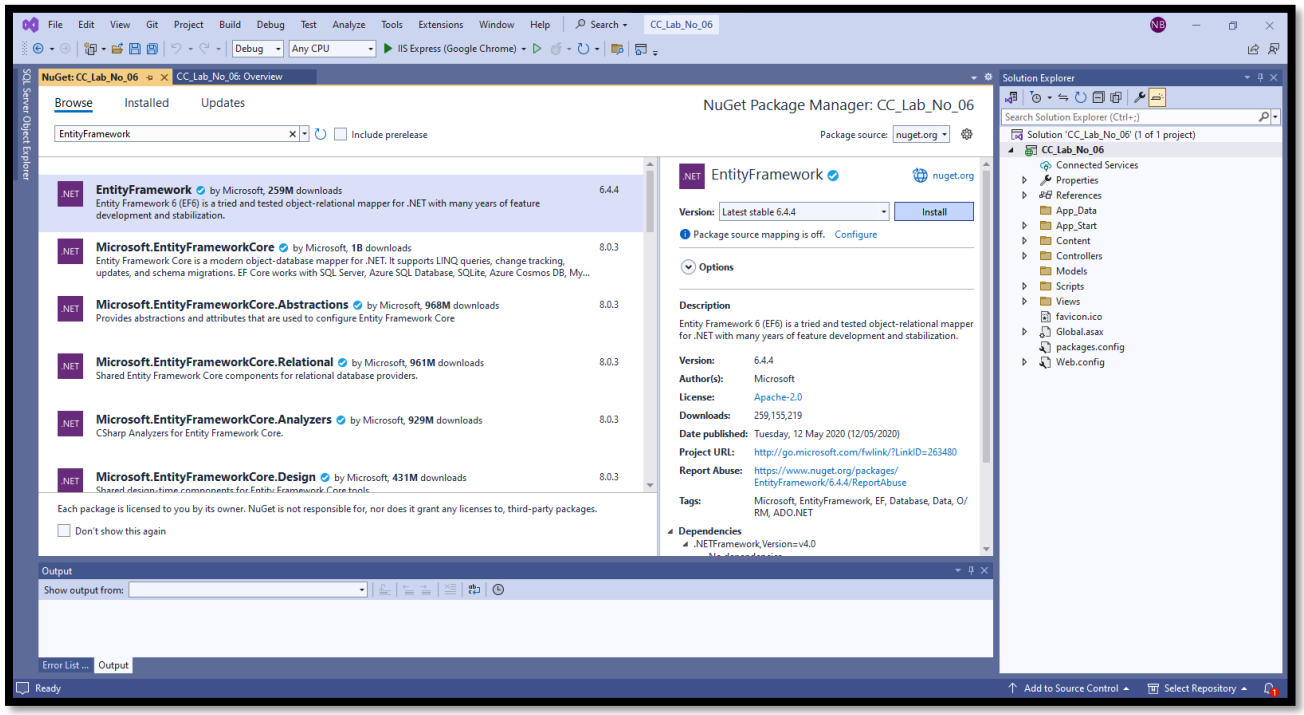


Step 2: Adding a NuGet package

- Right-click on your project in Solution Explorer, and then Choose "Manage Nugget Packages".

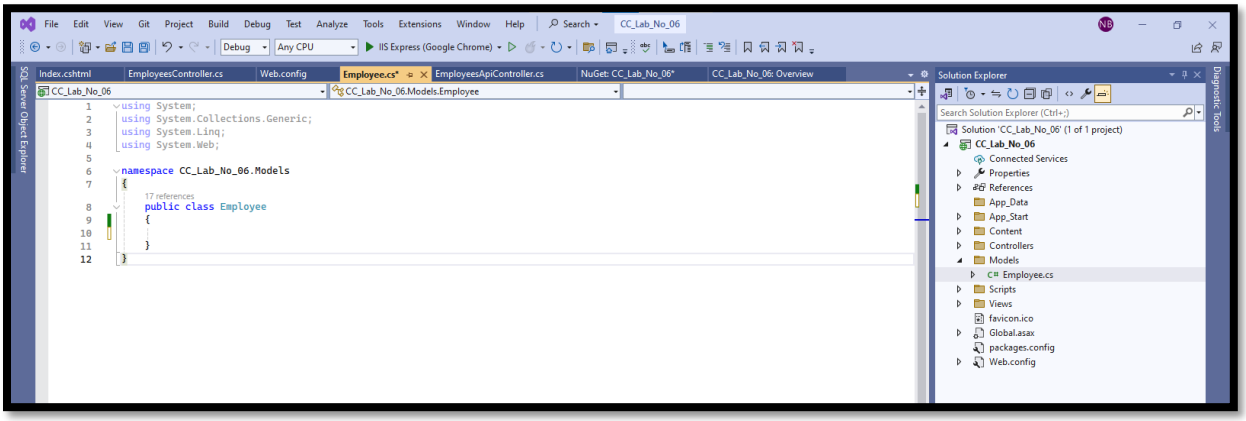


- Select the **Browse** tab. Enter **EntityFramework** in the search box, and then select it, then click install.



Step 3: Creating Business Classes in Models' Folder

- In **Solution Explorer**, right-click the Models folder and select **Add** > **Class**. Name the class Employee and add it.

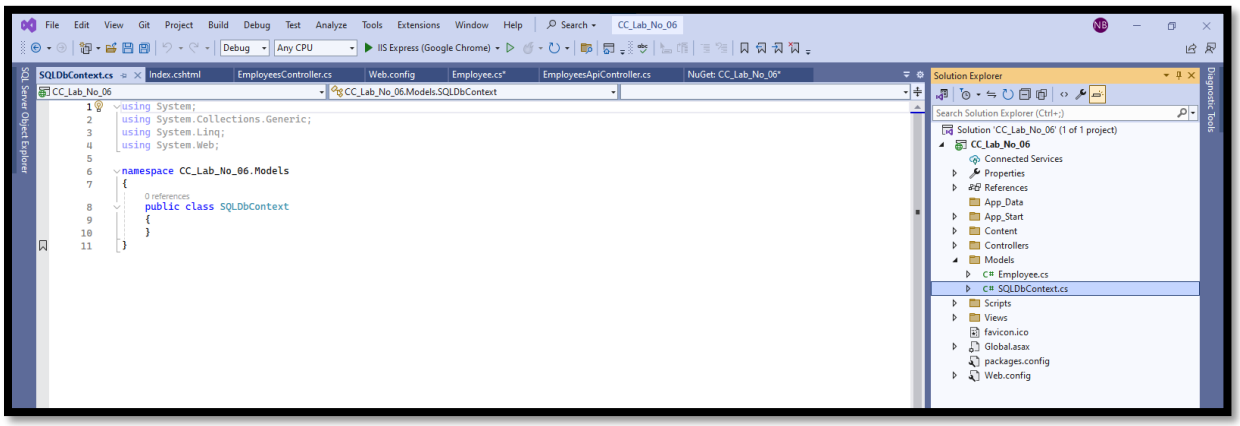


After creating the class add the following code:

```
public class Employee
{
      public string Id { get; set; }
      public string Name { get; set; }
      public string Address { get; set; }
      public string Gender { get; set; }
      public string Company { get; set; }
      public string Designation { get; set; }
}
```

- After that create a **"SQLDbContext"** class in the folder of models for database connectivity.

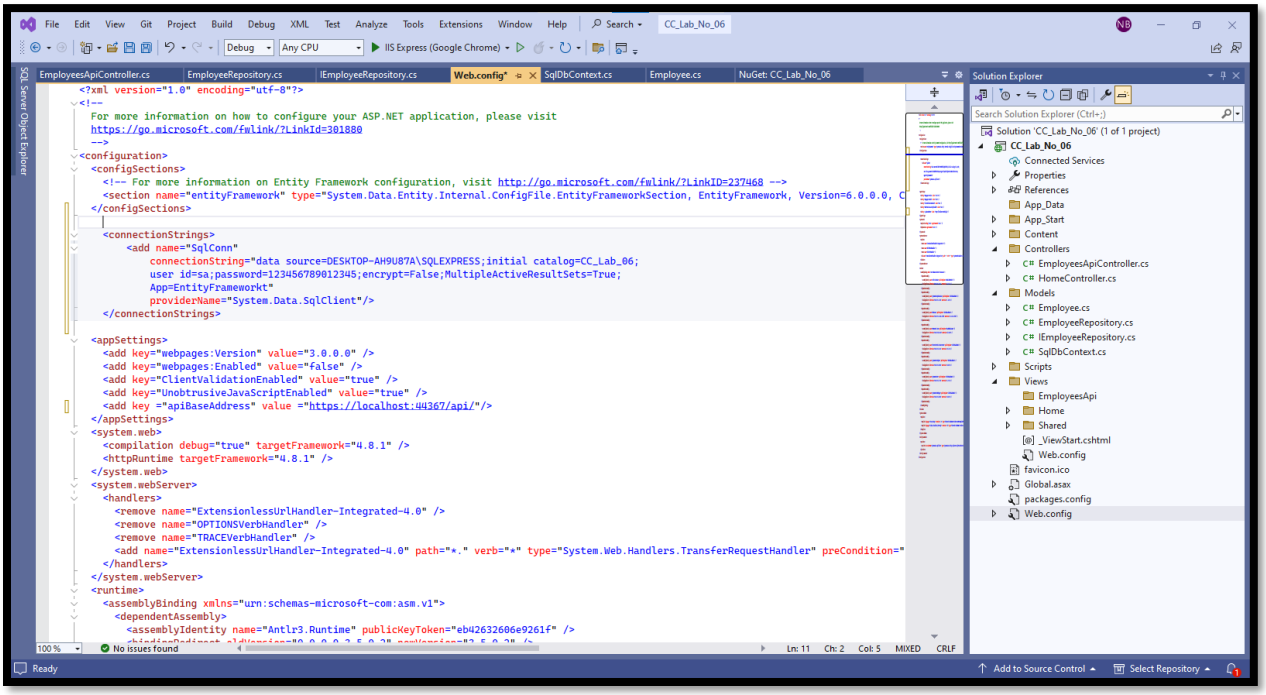
(The class that derives DbContext is called context class in entity framework. DbContext is an important class in Entity Framework API. It is a bridge between domain or entity classes and the database. DbContext is the primary class that is responsible for interacting with the database.)



Add the following code.

```
public class SqlDbContext : DbContext
{
    public SqlDbContext() : base("name=SqlConn")
    {
    }

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

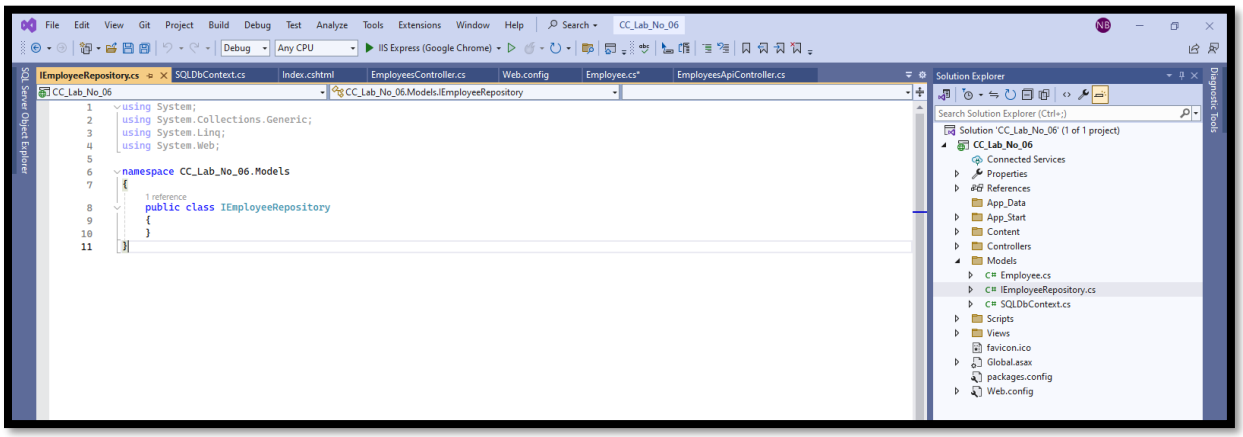
- As we've used a connection "SqlConn" in above DbContext class. Hence, we can create the connection string in Web.Config file.




```
<connectionStrings>

    <add name="SqlConn"

        connectionString = "data source=DESKTOP-AH9U87A\SQLEXPRESS;
                            initial catalog = CC_Lab_06;
                            user  id=sa;
                            password = 123456789012345;
                            encrypt = False;
                            MultipleActiveResultSets=True;
                             App=EntityFrameworkt"

        providerName="System.Data.SqlClient"/>
</connectionStrings>
```

- We are following the repository pattern in this application. So, we can create a "IEmployeeRepository" interface and define all functions there. So create this class in folder of models.

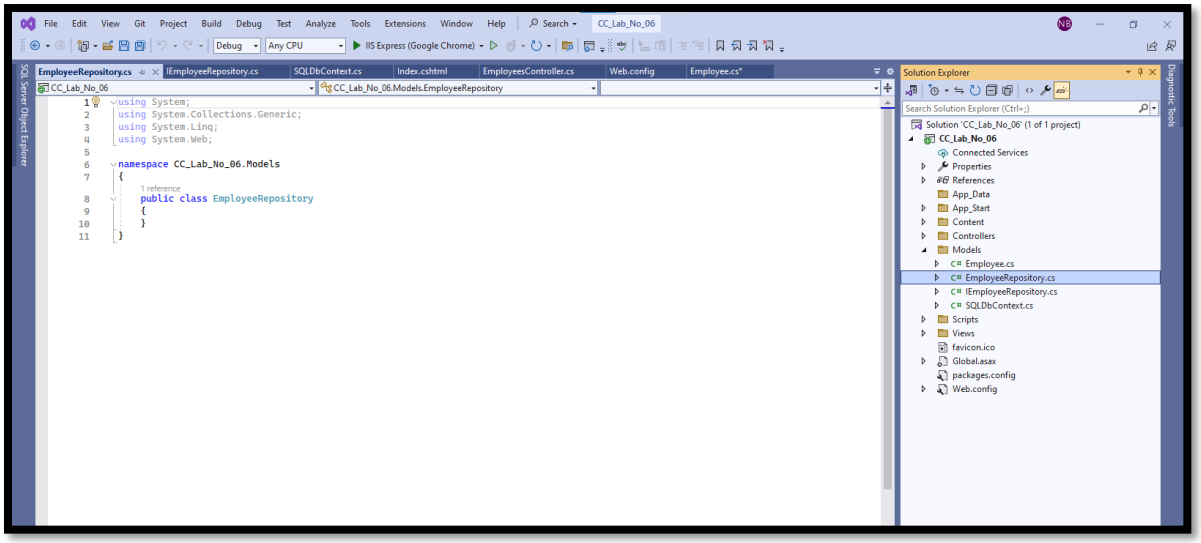
(The **Repository Pattern** abstracts data access logic, separating it from the rest of the application. It encapsulates CRUD operations, providing a clean interface for interacting with the data source. By promoting modularity and testability, it enhances code maintainability and flexibility in handling different data storage technologies.)



Add the following code in the created Interface.

```
public interface IEmployeeRepository
{
    Task Add(Employee employee);
    Task Update(Employee employee);
    Task Delete(string id);
    Task<Employee> GetEmployee(string id);
    Task<IEnumerable<Employee>> GetEmployees();
}
```

- Now you have to implement the exact logic for CRUD actions in "EmployeeRepository" class.



We will implement IEmployeeRepository interface in this class.

```
public class EmployeeRepository : IEmployeeRepository
{
    private readonly SqlDbContext db = new SqlDbContext();
}
```

'Implementation of ADD'

```
public async Task Add(Employee employee)
{
    employee.Id = Guid.NewGuid().ToString();
    db.Employees.Add(employee);
    try
    {
        await db.SaveChangesAsync();
    }
    catch
    {
        throw;
    }
}
```

'Implementation of UPDATE'

```
public async Task Update(Employee employee)
{
    try
    {
        db.Entry(employee).State = EntityState.Modified;
        await db.SaveChangesAsync();
    }
    catch
    {
        throw;
    }
}
```

'Implementation of DELETE'

```
public async Task Delete(string id)
{
    try
    {
        Employee employee = await db.Employees.FindAsync(id);
        db.Employees.Remove(employee);
        await db.SaveChangesAsync();
    }
    catch
    {
        throw;
    }
}
```

'Implementation of Employee Exist Method'

```
private bool EmployeeExists(string id)
{
    return db.Employees.Count(e => e.Id == id) > 0;
}
```

'Implementation of GET EMPLOYEE (using ID)'

```
public async Task<Employee> GetEmployee(string id)
 {
     try
     {
         Employee employee = await db.Employees.FindAsync(id);
         if (employee == null)
         {
             return null;
         }
         return employee;
     }
     catch
     {
         throw;
     }
 }
```

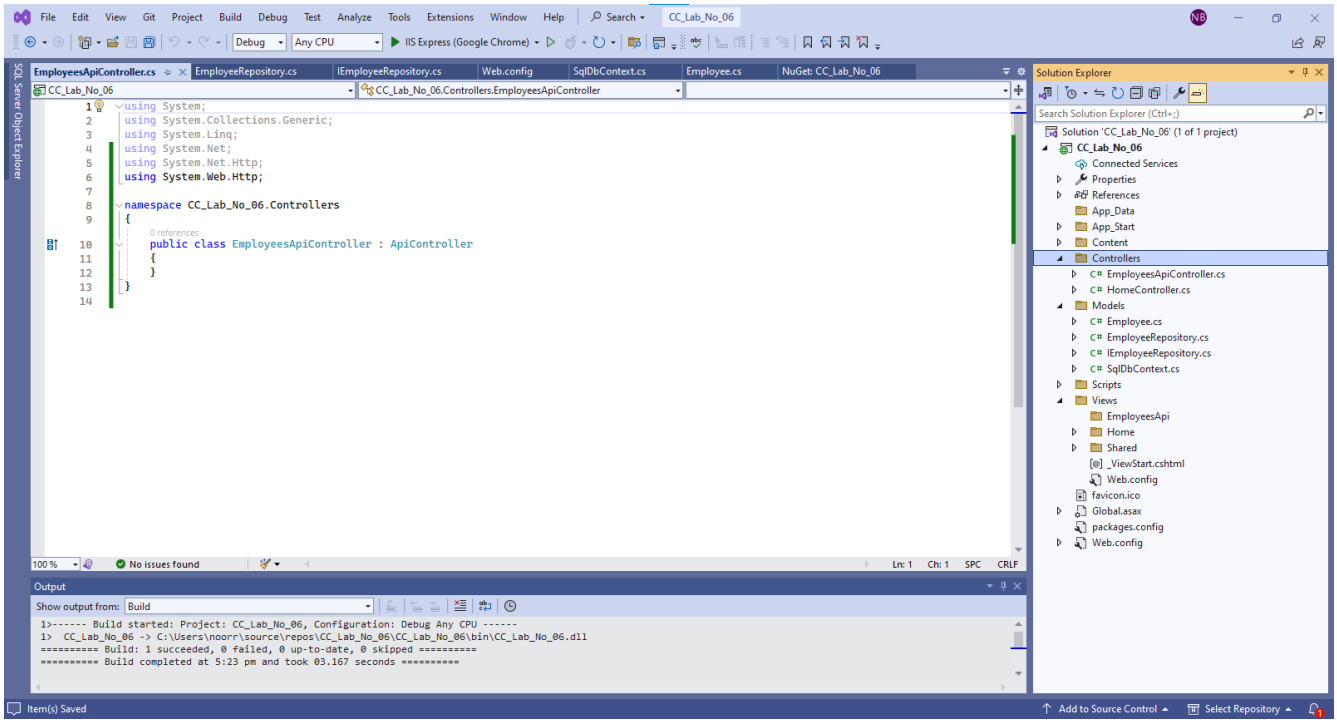
'Implementation of GET EMPLOYEE '

```
public async Task<IEnumerable<Employee>> GetEmployees()
{
    try
    {
        var employees = await db.Employees.ToListAsync();
        return employees.AsQueryable();
    }
    catch
    {
        throw;
    }
}
```

So, all the five methods (for CRUD) has been implemented in this class.

Step 4: Creating Web API Controller

- Right-click the *Controllers* folder. Select **Add** > **Controller.** Select **API > Web API 2 Controller – Empty,** add it and name it as **EmployeesApiController** .



Now add the follwing methods in it:

```
public class EmployeesApiController : ApiController
{
    private readonly IEmployeeRepository _iEmployeeRepository =
                     new EmployeeRepository();
}
```

‘Implementation of GET Method ’

```
[HttpGet]
[Route("api/Employees/Get")]
public async Task<IEnumerable<Employee>> Get()
{
     return await _iEmployeeRepository.GetEmployees();
}
```

‘Implementation of Deatails (by ID) ’

```
[HttpGet]
[Route("api/Employees/Details/{id}")]
public async Task<Employee> Details(string id)
{
    var result = await _iEmployeeRepository.GetEmployee(id);
    return result;
}
```

'Implementation of CREATE Method'

```
[HttpPost]
[Route("api/Employees/Create")]
public async Task CreateAsync([FromBody] Employee employee)
{
      if (ModelState.IsValid)
      {
          await _iEmployeeRepository.Add(employee);
      }
}
```

'Implementation of EDIT Method'

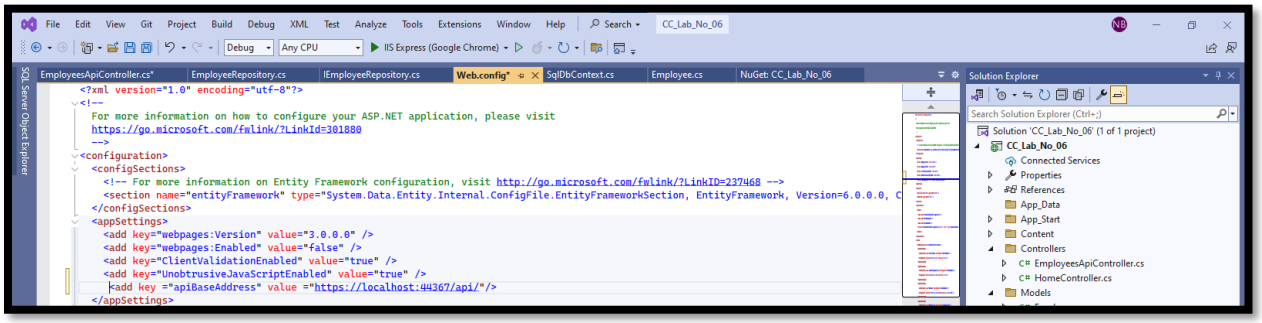
```
[HttpPut]
[Route("api/Employees/Edit")]
public async Task EditAsync([FromBody] Employee employee)
{
      if (ModelState.IsValid)
      {
           await _iEmployeeRepository.Update(employee);
      }
}
```

'Implementation of DELETE Method'

```
[HttpDelete]
[Route("api/Employees/Delete/{id}")]
public async Task DeleteConfirmedAsync(string id)
{
      await _iEmployeeRepository.Delete(id);
}
```

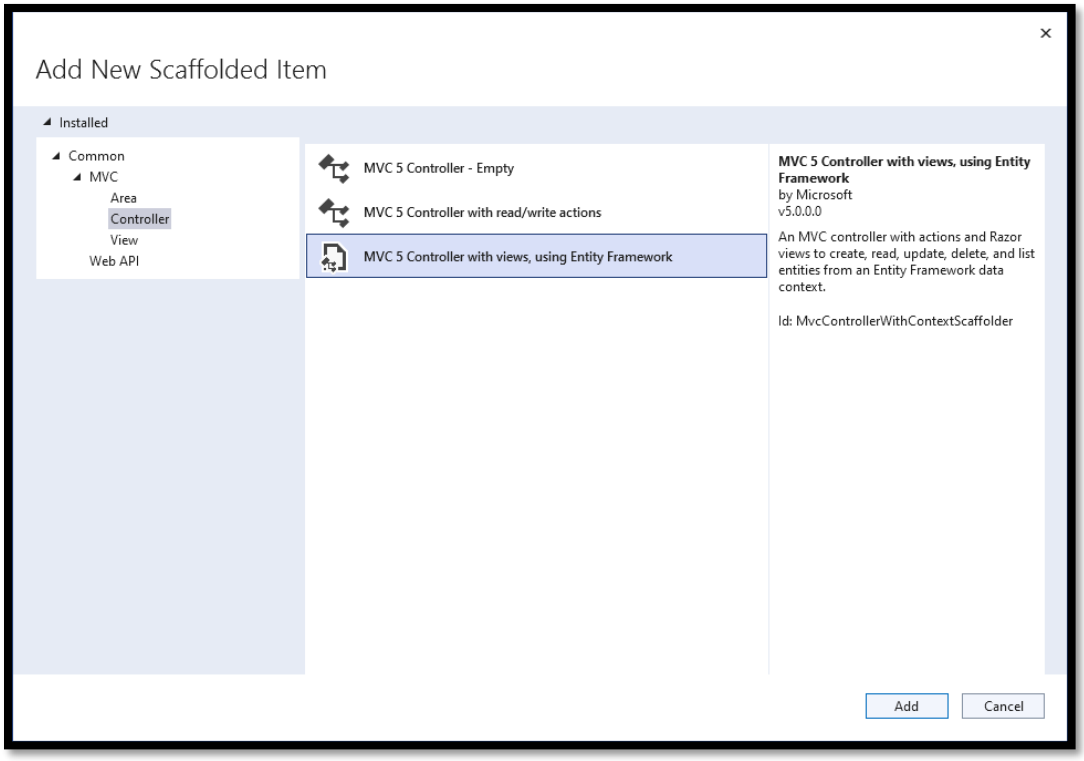
All the CRUD actions are derived in this API class. An instance of the EmployeeRepository class has been created, and with the help of this instance, all the methods from the EmployeeRepository class have been accessed in our API class.

- Now, it is needed to add this base URL in the Web.Config file because it will be used in MVC controllers. Let's create a new key-value pair in Web.Config file under "appSettings" section.

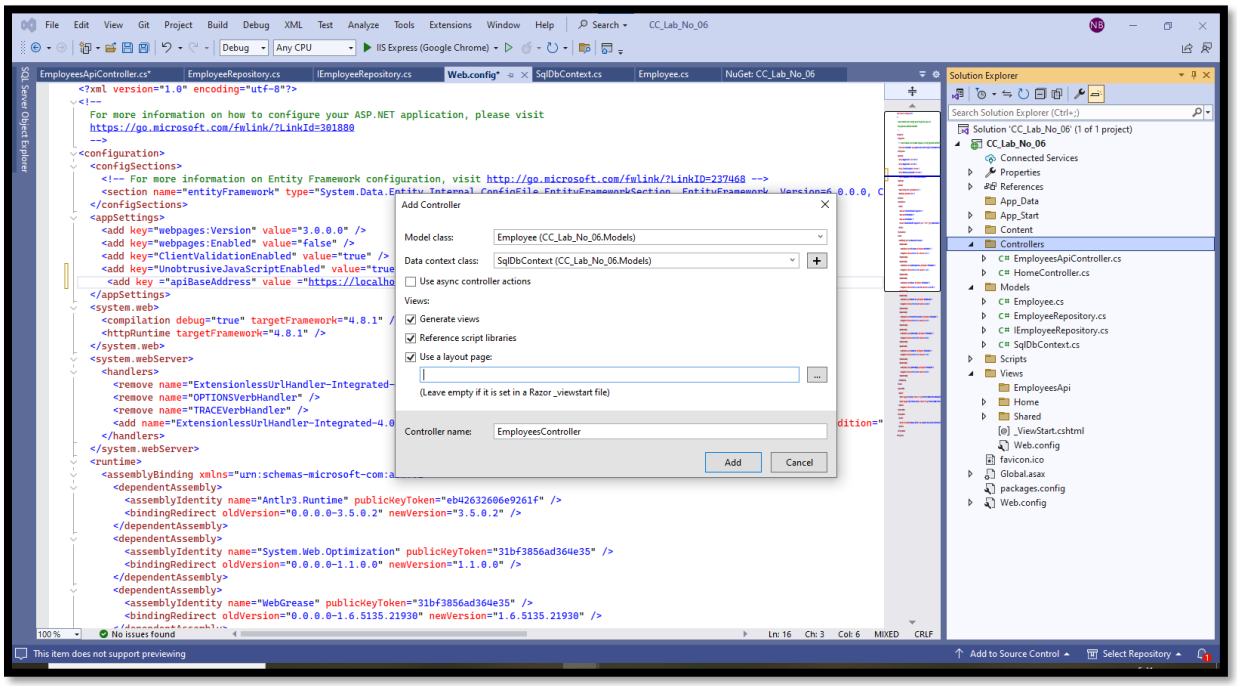


Step 5: Scaffold a controller

- Right-click the *Controllers* folder. Select **Add** > **Controller**. Select **MVC 5 Controller with views, using Entity Framework,** and click add**.**



- In the **Add Controller** dialog box, select the following configurations**.**



Now add the following methods in the the controller created.

```
public class EmployeesController : Controller
{
    readonly string apiBaseAddress = ConfigurationManager.AppSettings["apiBaseAddress"];

    public async Task<ActionResult> Index()
    {
        IEnumerable<Employee> employees = null;
        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri(apiBaseAddress);
            var result = await client.GetAsync("employees/get");
            if (result.IsSuccessStatusCode)
            {
                employees = await result.Content.ReadAsAsync<IList<Employee>>();
            }
            else
            {
                employees = Enumerable.Empty<Employee>();
                ModelState.AddModelError(string.Empty, "Error try after some time.");
            }
        }
        return View(employees);
    }


    public async Task<ActionResult> Details(string id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        Employee employee = null;
        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri(apiBaseAddress);
            var result = await client.GetAsync($"employees/details/{id}");
            if (result.IsSuccessStatusCode)
            {
                employee = await result.Content.ReadAsAsync<Employee>();
            }
            else
            {
                ModelState.AddModelError(string.Empty, "Error try after some time.");
            }
        }
        if (employee == null)
        {
            return HttpNotFound();
        }
        return View(employee);
    }


    public ActionResult Create()
    {
        return View();
    }
```

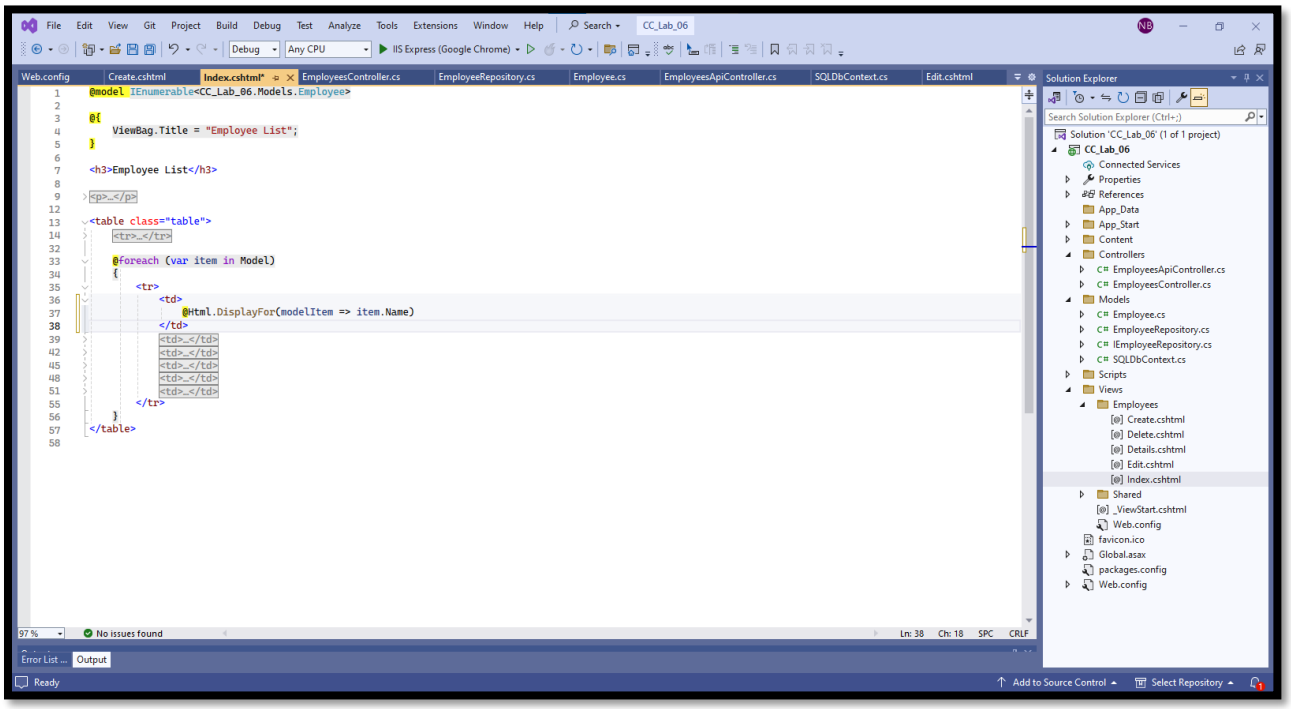
```
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> Create([Bind(Include =
"Name,Address,Gender,Company,Designation")] Employee employee)
    {
        if (ModelState.IsValid)
        {
            using (var client = new HttpClient())
            {
                client.BaseAddress = new Uri(apiBaseAddress);

                var response = await client.PostAsJsonAsync("employees/Create", employee);
                if (response.IsSuccessStatusCode)
                {
                    return RedirectToAction("Index");
                }
                else
                {
                  ModelState.AddModelError(string.Empty, "Server error try after some time.");
                }
            }
        }
        return View(employee);
    }


    public async Task<ActionResult> Edit(string id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        Employee employee = null;

        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri(apiBaseAddress);

            var result = await client.GetAsync($"employees/details/{id}");

            if (result.IsSuccessStatusCode)
            {
                employee = await result.Content.ReadAsAsync<Employee>();
            }
            else
            {
                ModelState.AddModelError(string.Empty, "Server error try after some time.");
            }
        }

        if (employee == null)
        {
            return HttpNotFound();
        }
        return View(employee);
    }
```

```
    [HttpPost]
    [ValidateAntiForgeryToken]

    public async Task<ActionResult> Edit([Bind(Include =
"Id,Name,Address,Gender,Company,Designation")] Employee employee)
    {
        if (ModelState.IsValid)
        {
            using (var client = new HttpClient())
            {
                client.BaseAddress = new Uri(apiBaseAddress);

                var response = await client.PutAsJsonAsync("employees/edit", employee);

                if (response.IsSuccessStatusCode)
                {
                    return RedirectToAction("Index");
                }
                else
                {
                    ModelState.AddModelError(string.Empty, "Error try after some time.");
                }
            }
            return RedirectToAction("Index");
        }
        return View(employee);
    }


    public async Task<ActionResult> Delete(string id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        Employee employee = null;
        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri(apiBaseAddress);

            var result = await client.GetAsync($"employees/details/{id}");
            if (result.IsSuccessStatusCode)
            {
                employee = await result.Content.ReadAsAsync<Employee>();
            }
            else
            {
                ModelState.AddModelError(string.Empty, "Server error try after some time.");
            }
        }

        if (employee == null)
        {
            return HttpNotFound();
        }
        return View(employee);
    }
```

```
    [HttpPost, ActionName("Delete")]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> DeleteConfirmed(string id)
    {
        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri(apiBaseAddress);
            var response = await client.DeleteAsync($"employees/delete/{id}");
            if (response.IsSuccessStatusCode)
            {
                return RedirectToAction("Index");
            }
            else
            {
                ModelState.AddModelError(string.Empty, "Server error try after some time.");
            }
        }
        return View();
    }
}
```

Step 6: Update the Views

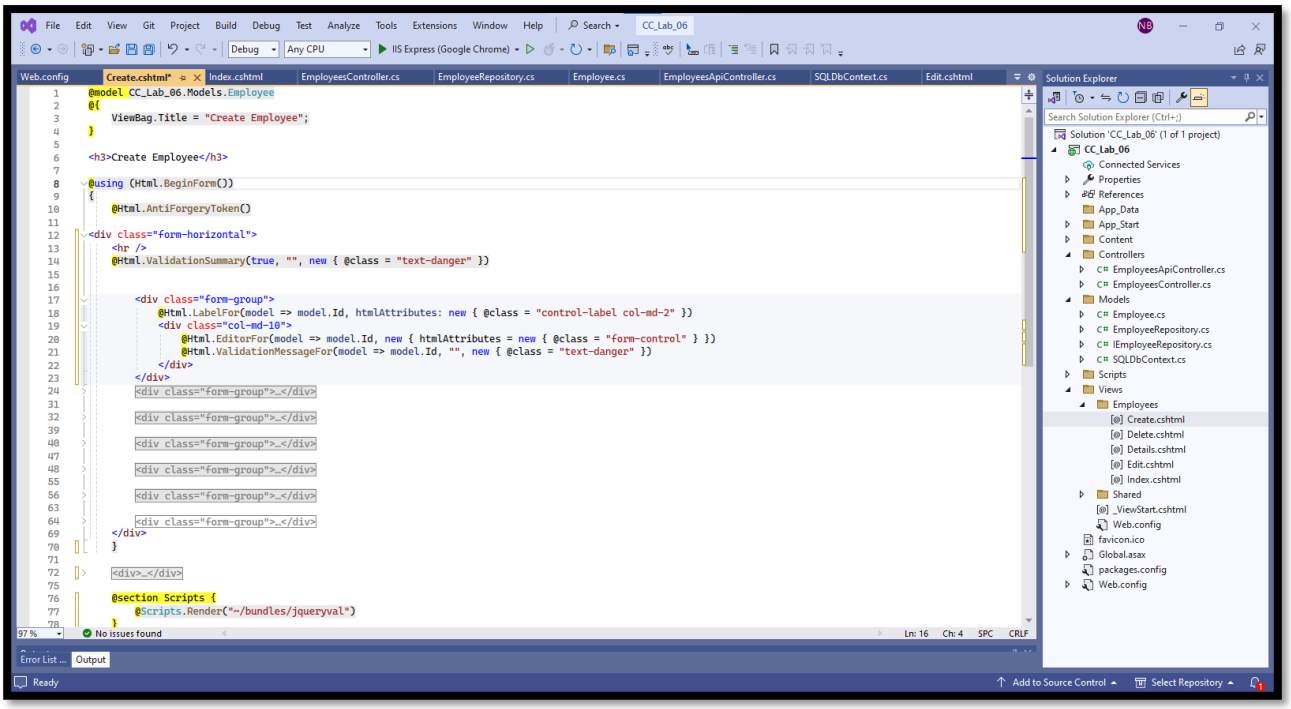
- Open the views created in the Employees folder, and then update the index.cshtml view as:



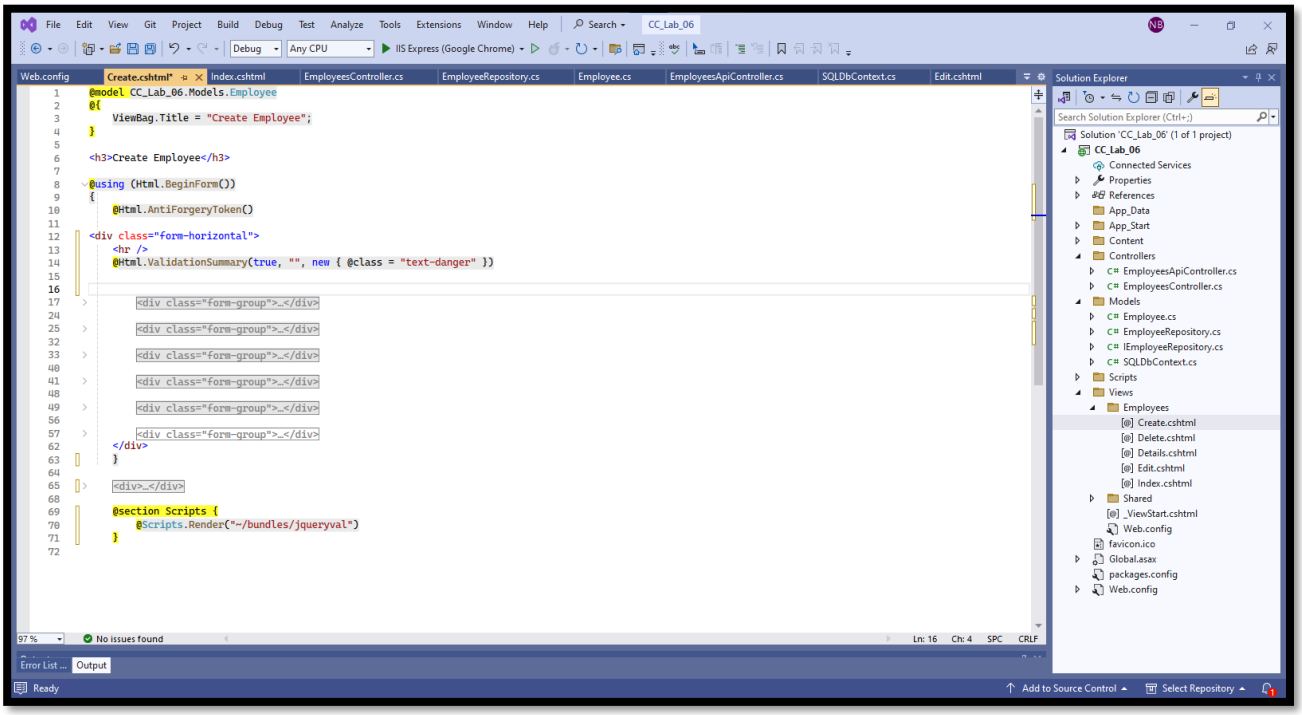

```
<td>
        @Html.ActionLink(item.Name, "Details", new { id = item.Id })
</td>
```

The existing "Index" view has been modified. The "Details" link has been removed from this view, and instead, a hyperlink has been provided in the employee name itself for details.

- Now update the create view. The "Create" view can be modified by removing the Id field. For us, the employee Id will be created automatically during the insertion of new data. We have utilized a system GUID for this purpose.

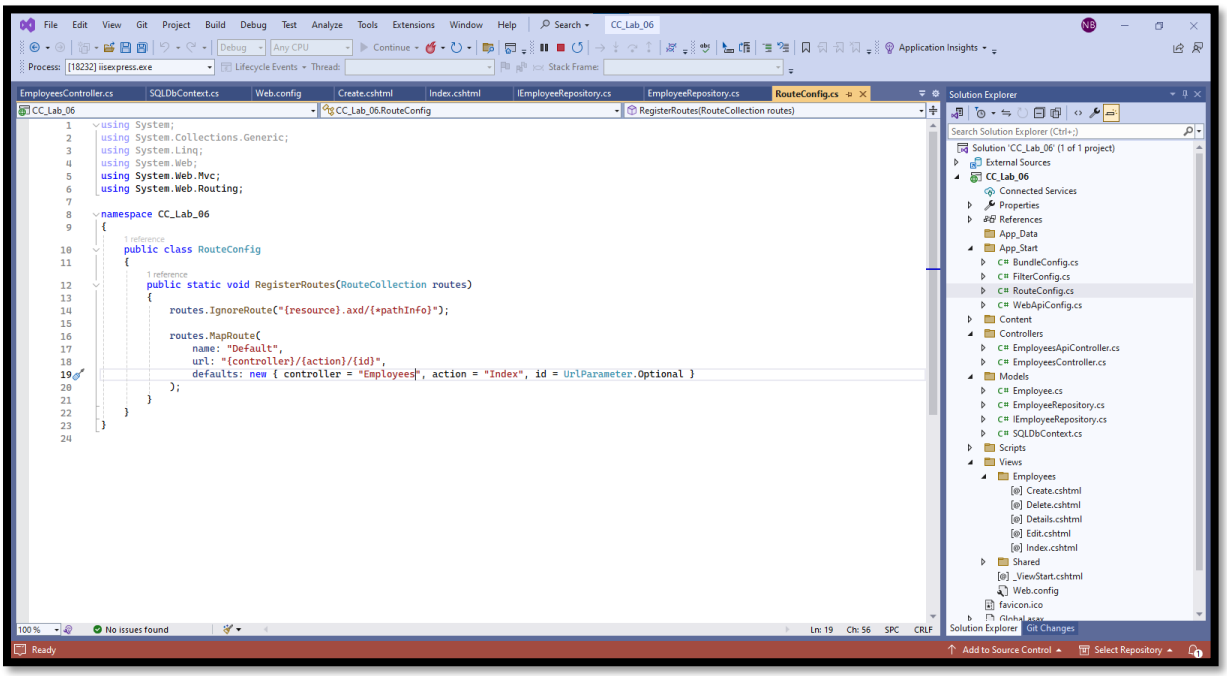


The id field has been removed.

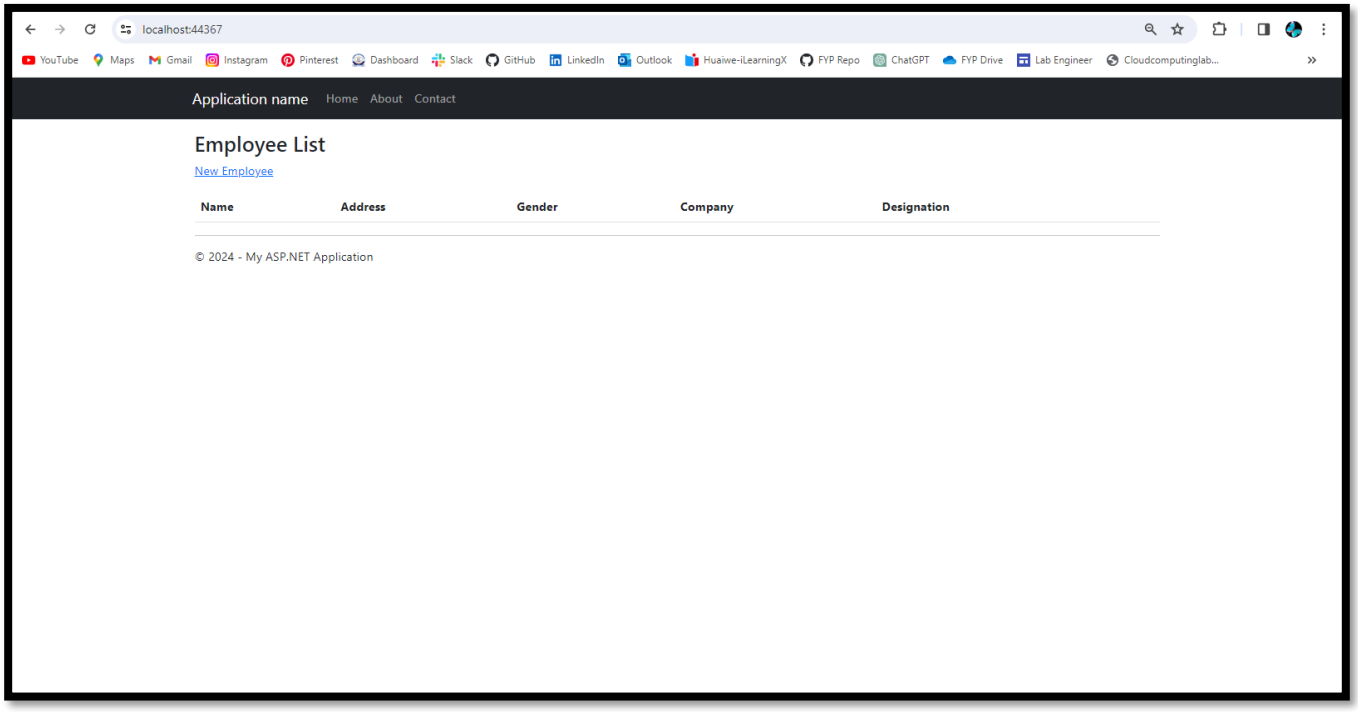


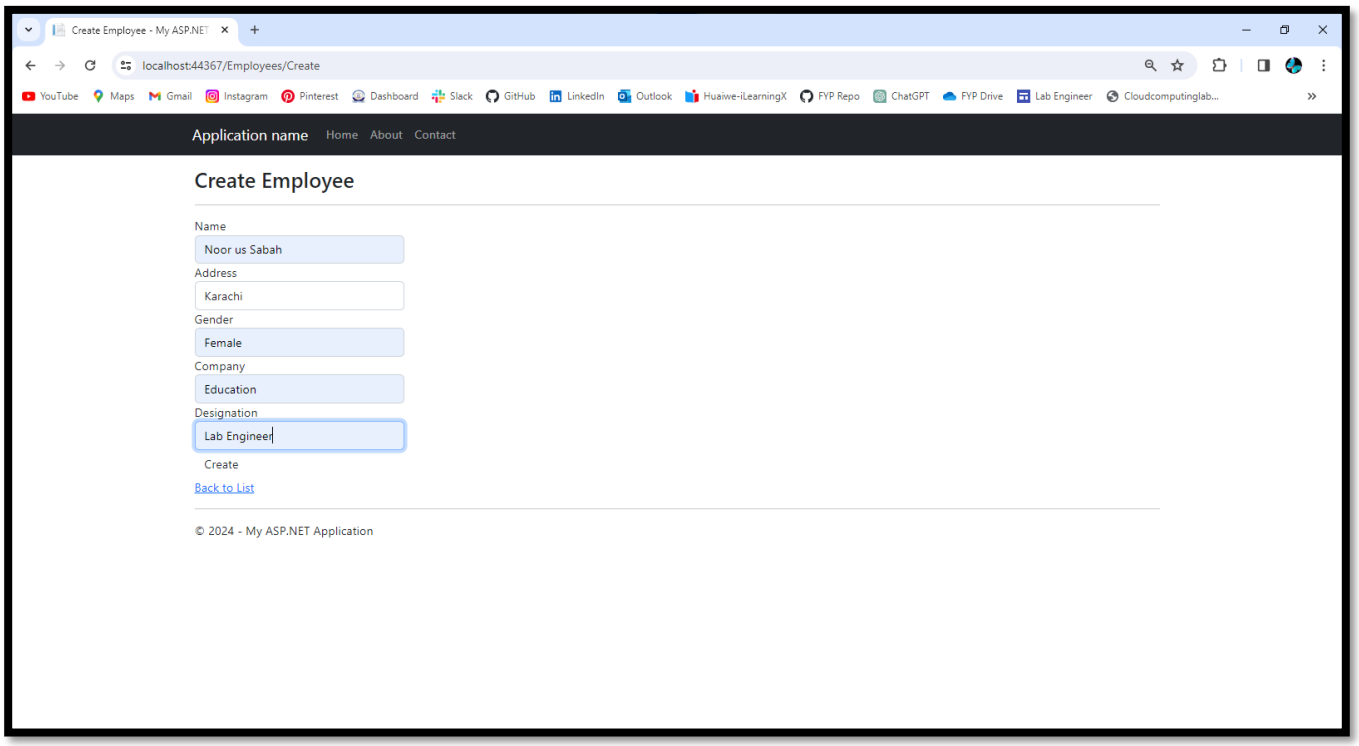
Step 6: Configure the route.

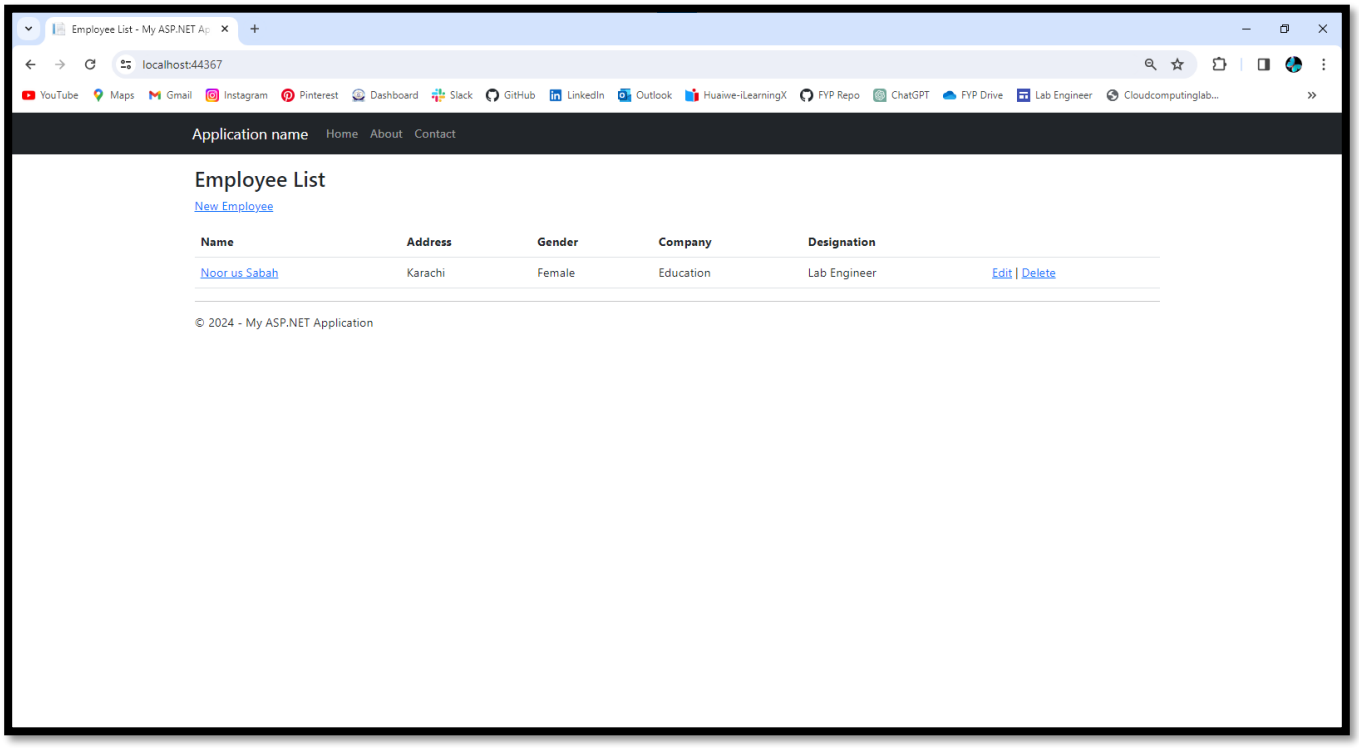
- Locate the RouteConfig.cs file in the App_Start folder.



Step 7: Run the Project




Create Employee - My ASP.NET
localhost:44367/Employees/Create
YouTube Maps Gmail Instagram Pinterest Dashboard Slack GitHub LinkedIn Outlook Huaiwe-iLearningX FYP Repo ChatGPT FYP Drive Lab Engineer Cloudcomputinglab...
Application name Home About Contact
Create Employee
Name
Noor us Sabah
Address
Karachi
Gender
Female
Company
Education
Designation
Lab Engineer
Create
Back to List
© 2024 - My ASP.NET Application


Employee List - My ASP.NET Ap
localhost:44367
YouTube Maps Gmail Instagram Pinterest Dashboard Slack GitHub LinkedIn Outlook Huaiwe-iLearningX FYP Repo ChatGPT FYP Drive Lab Engineer Cloudcomputinglab...
Application name Home About Contact
Employee List
New Employee
Name Address Gender Company Designation
Noor us Sabah Karachi Female Education Lab Engineer Edit | Delete
© 2024 - My ASP.NET Application

2. Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Visual studio Environment	3 mints + 5 mints	8 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	178 mints

3. Objectives

After completing this lab the student should be able to:

a. *Understand the concept of REST-based APIs and their pivotal role in enabling communication between diverse software applications.*

b. *Develop fundamental ASP.NET Core Web API endpoints tailored for RESTful interactions, adept at managing HTTP requests and responses.*

c. *Acquire a deep understanding of the core principles governing REST-based ASP.NET Core Web API development, encompassing routing, controllers, and data serialization strategies.*

4. Lab Tasks/Practical Work

1. Implement an ASP.NET Core Web API employing RESTful principles, incorporating endpoints for GET (to fetch products), POST (to add products), PUT (to update products), and DELETE (to remove products) methods, streamlining the administration of a product inventory system.

2. Construct an ASP.NET Core Web API adhering to REST conventions, designed to manage student records through CRUD operations (GET, POST, PUT, and DELETE), enabling the seamless addition, retrieval, modification, and deletion of student information.