Lab - How to Perform CRUD Operations Using Blazor and Entity Framework Core 3.0

Entity Framework Core

Entity Framework (EF) Core is the latest version of Entity Framework from Microsoft. It is an open-source, lightweight, extensible, and cross-platform version of Entity Framework. It runs on Windows, Mac, and Linux.

Entity Framework is an object/relational mapping (O/RM) framework. It is an enhancement of ADO.NET that gives developers an automated mechanism for accessing and storing data in a database.

Every web app needs a data source, and EF Core is a great way to interact with a SQL database. So, let's walk through the process of adding EF Core to a Blazor app.

Blazor

In the past, JavaScript was the only programming language available for client-side web applications. Now, we have different choices of client

frameworks, such as Angular, React, and others, but in the end, it always run as JavaScript in the browser.

Recently, WebAssembly (wasm) changes all of that.

According to the WebAssembly.org website:

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

Blazor is a UI web framework built on .NET. It runs on browsers using WebAssembly, and it provides choice for client-side programming language—C# rather than JavaScript.

Now, we will see how to do CRUD operations using a Blazor app with EF Core.

Prerequisites

- Visual Studio 2019 16.3.0 Preview 2.0
- .NET Core 3.0
- SQL Server 2017

Create database

Let’s create a database on our local SQL Server.

1. Open SQL Server 2017.
2. Create a new database named **Management**. Now we have the database in place.
3. Click on our database and choose **New Query**.
4. For this application, I am going to create a table with the name **Employee** with some basic attributes. Paste the following SQL query into the **Query** window to create the **Employee** table.

```
Create Table Employee(

EmployeeId BigInt Identity(1,1) Primary Key,


```

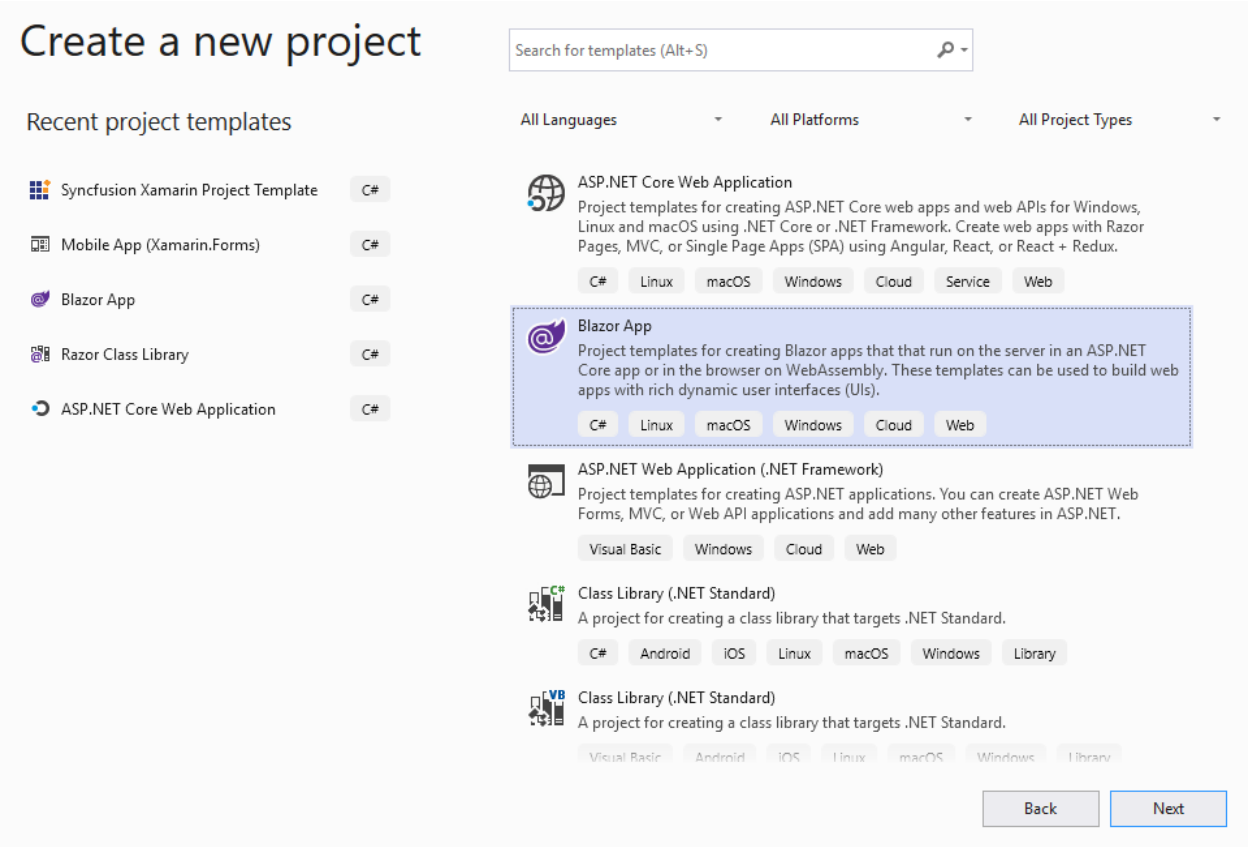
```
Name Varchar(100) Not Null,

Designation Varchar(100),

Email Varchar(20),

Location Varchar(50) Not Null,

PhoneNumber bigint Not Null)
```

Create Blazor application

Follow these steps to create a Blazor application:

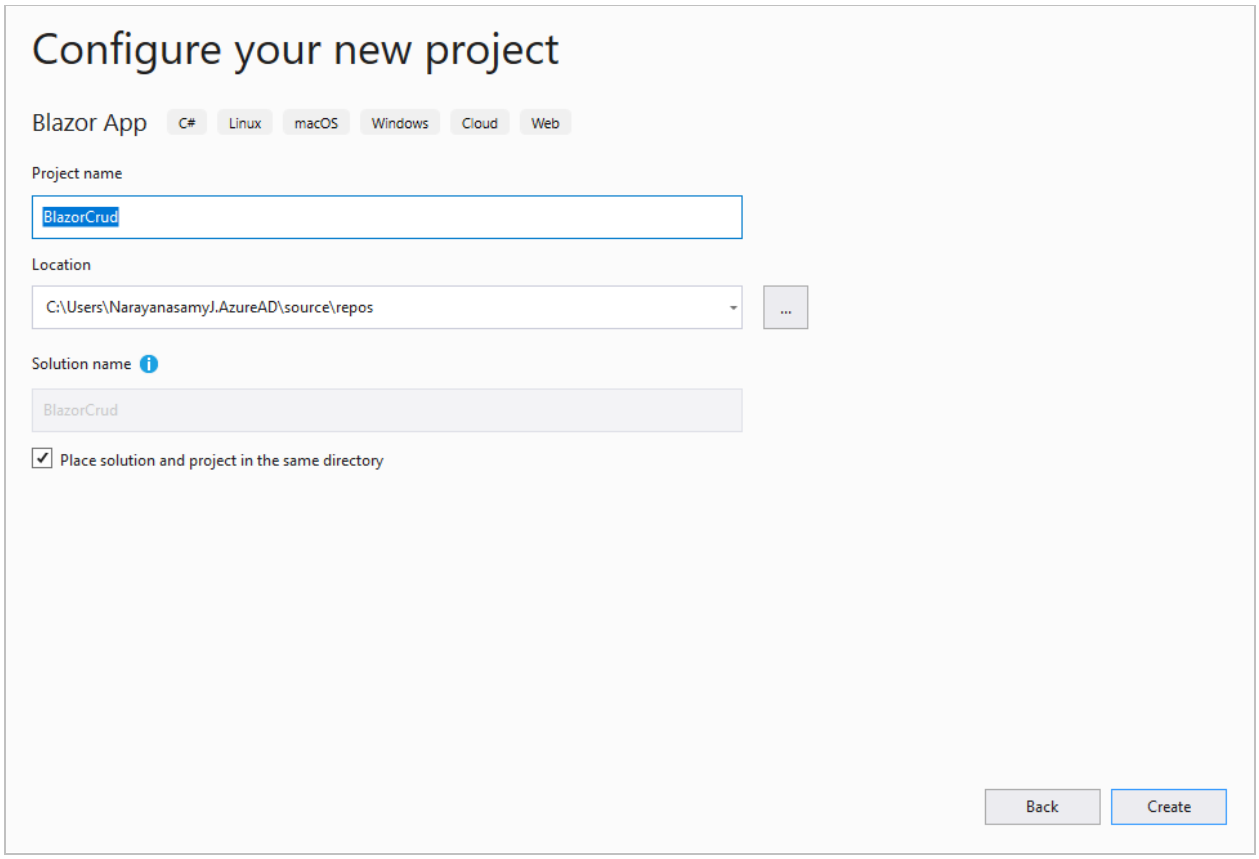
1. In Visual Studio 2019, go to **File** > **New** > **Project**.
2. Choose the **Create a new project**.

3. Select the **Blazor App**.



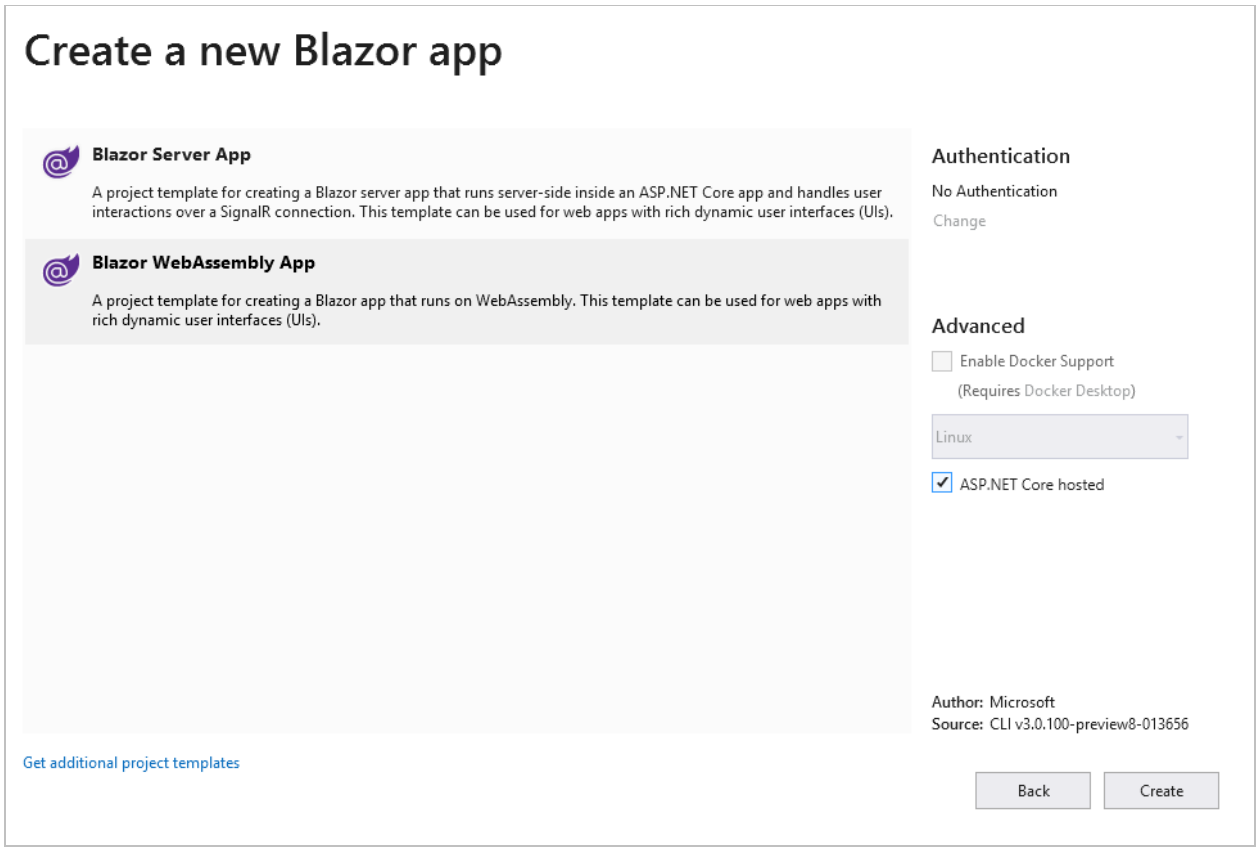
Selecting a Project Template in Visual Studio

4. Enter a project name and click **Create**. In my example, I used the name **BlazorCrud**.



Configuring the New Project in Visual Studio

5. Select **Blazor WebAssembly App**, choose the **ASP.NET Core Hosted** template, and then click **Create**. The sample Blazor application will be created.



Selecting a Blazor WebAssembly App

Now, we have three project files created inside this solution:

- **Client**: Contains the client-side code and the pages that will be rendered on the browser.
- **Server**: Contains the server-side code such as DB-related operations and web API.
- **Shared**: Contains the shared code that can be accessed by both client and server.

Install NuGet packages

The following NuGet packages should be added to work with the SQL Server database and scaffolding. Run these commands in the **Package Manager Console**:

- **Install-Package Microsoft.EntityFrameworkCore.Tools -Version 3.0.0:** This package creates database context and model classes from the database.
- **Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 3.0.0:** The database provider that allows Entity Framework Core to work with SQL Server.

Creating a model for an existing database

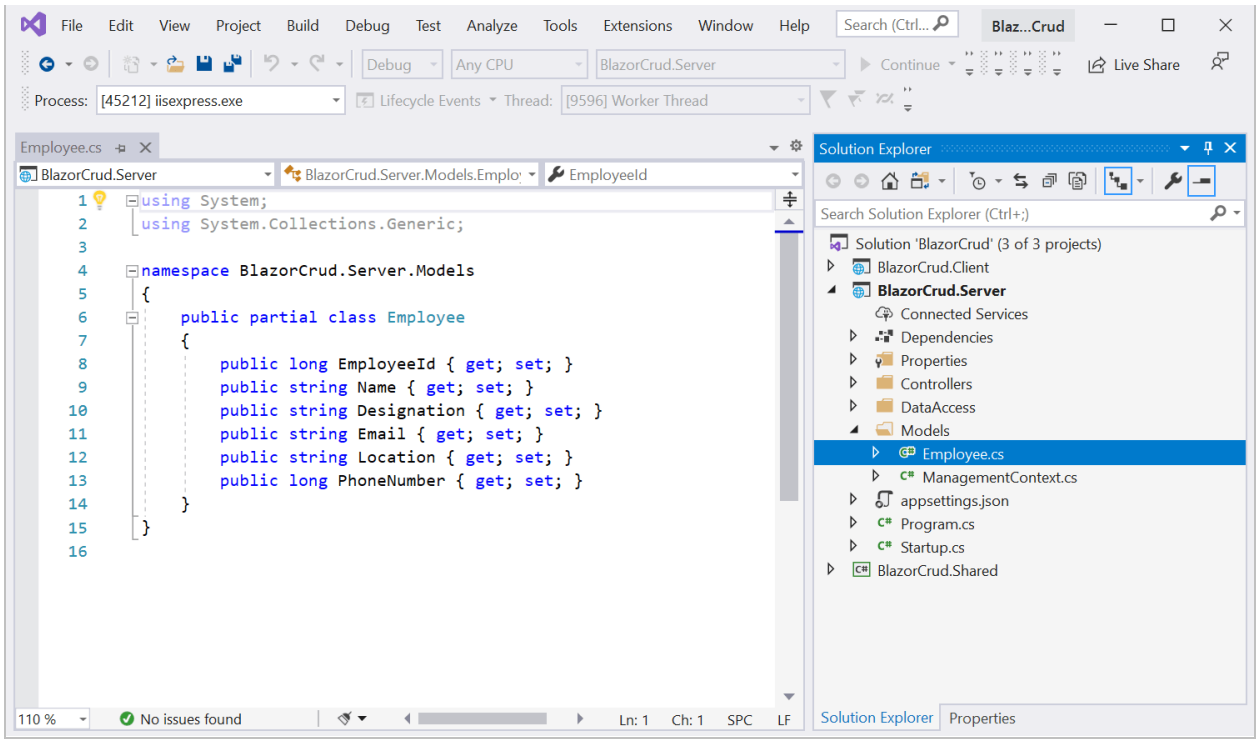
Let's create entity and context classes for the **Management** database in the local SQL server.

Run the following scaffold command in the **Package Manager Console** to reverse engineer the database to create database context and entity classes from any tables. The **Scaffold** command will create a class for the tables that have a primary key.

```
Scaffold-DbContext
“Server=.\\;Database=Management;Integrated
Security=True”
Microsoft.EntityFrameworkCore.SqlServer -OutputDir
Models
```

- **Connection**: Sets the connection string of the database.
- **Provider**: Sets the provider to use to connect the database.
- **OutputDir**: Sets the directory where the POCO classes are to be generated.

After running the command, the **Employee** class and **Management Context** class will be created as shown in the following screenshot.



In the auto-generated **EmployeeContext** class file, the database credentials you can see are hard coded in the **OnConfiguring** method. It is not a good practice to have SQL Server credentials in C# class, considering the security issues. So, remove the following **OnConfiguring** method and **parameterless constructor** from context file.

```
public partial class ManagementContext : DbContext
{
    public ManagementContext()
    {
    }

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

    public virtual DbSet<Employee> Employee { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
ning To protect potentially sensitive information in your connection string, you should move it out of source code. See http://go.mi
            optionsBuilder.UseSqlServer("Server=localhost\\SQLEXPRESS;Database=Management;User Id=sa;Password=sa@1234");
        }
    }
```

Then add the connection string into the **appsetting.json** file, as shown in the following screenshot.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "EmployeeDatabase": "Data Source=localhost\\SQLEXPRESS;Initial Catalog=Management;User Id=sa; Password=sa@1234;"
  }
}
```

Then, register the database context service (**EmployeeContext**) during application startup. In the following code, the connection string is read from the **appsetting** file and is passed to the context service.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().AddNewtonsoftJson(options => {
        options.SerializerSettings.ReferenceLoopHandling = ReferenceLoopHandling.Ignore;
    });

    services.AddResponseCompression(opts =>
    {
        opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
            new[] { "application/octet-stream" });
    });

    var connection = Configuration.GetConnectionString("EmployeeDatabase");
    services.AddDbContext<BlazorContext>(options => options.UseSqlServer(connection));
    services.AddScoped<IEmployeAccessLayer, EmployeAccessLayer>();
}
```

Whenever a new context is requested, it will be returned from the context pool if it is available, otherwise a new context will be created and returned.

Creating Data Access Layer for the Application

Right-click the **BlazorCrud.Server** and then select **Add >> New Folder** and named as **DataAccess.**

Create a class **EmployeAccessLayer** in **DataAccess** Folder. This class will handle the CRUD related DB operations. Paste the following code into it.

```
using BlazorCrud.Server.Models;

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;



namespace BlazorCrud.Server.DataAccess

{

    public interface IEmployeAccessLayer

    {

        IEnumerable GetAllEmployees();

        void AddEmployee(Employee employee);

        void UpdateEmployee(Employee employee);

        Employee GetEmployeeData(long id);
```

```
        void DeleteEmployee(long id);

    }



    public class EmployeAccessLayer : IEmployeAccessLayer

    {

        private ManagementContext _context;

        public EmployeAccessLayer(ManagementContext context)

        {

            _context = context;

        }



        //To Get all employees details

        public IEnumerable GetAllEmployees()

        {
```

```
            try
            {
                return _context.Employee.ToList();
            }
            catch(Exception ex)
            {
                throw;
            }
        }


        //To Add new employee record
        public void AddEmployee(Employee employee)
        {
            try
            {
```

```
                _context.Employee.Add(employee);

                _context.SaveChanges();

            }

            catch

            {

                throw;

            }

        }



        //To Update the records of a particluar employee

        public void UpdateEmployee(Employee employee)

        {

            try

            {

```

```
                _context.Entry(employee).State = EntityState.Modified;

                _context.SaveChanges();

            }

            catch

            {

                throw;

            }

        }



        //Get the details of a particular employee

        public Employee GetEmployeeData(long id)

        {

            try

            {

```

```
                Employee employee = _context.Employee.Find(id);

                return employee;

            }

            catch

            {

                throw;

            }

        }



        //To Delete the record of a particular employee

        public void DeleteEmployee(long id)

        {

            try

            {

```

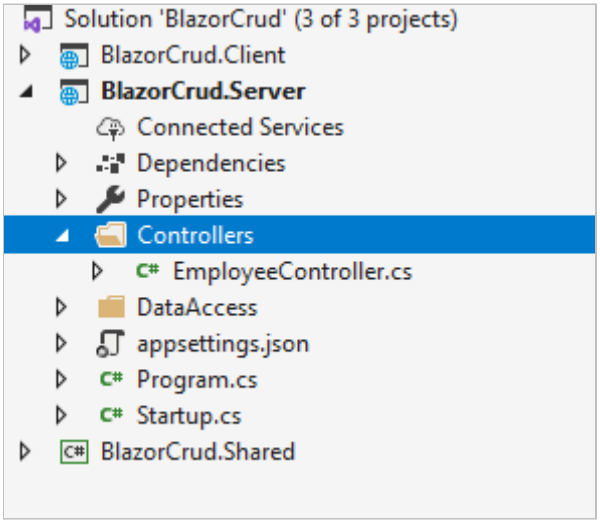
```
                Employee emp = _context.Employee.Find(id);
                _context.Employee.Remove(emp);
                _context.SaveChanges();
            }
            catch
            {
                throw;
            }
        }
    }
}
```

8
5
8
6
8
7
8
8
8
9
9
0
9
1
9
2
9
3
9
4
9
5
9
6

```
9
7
```

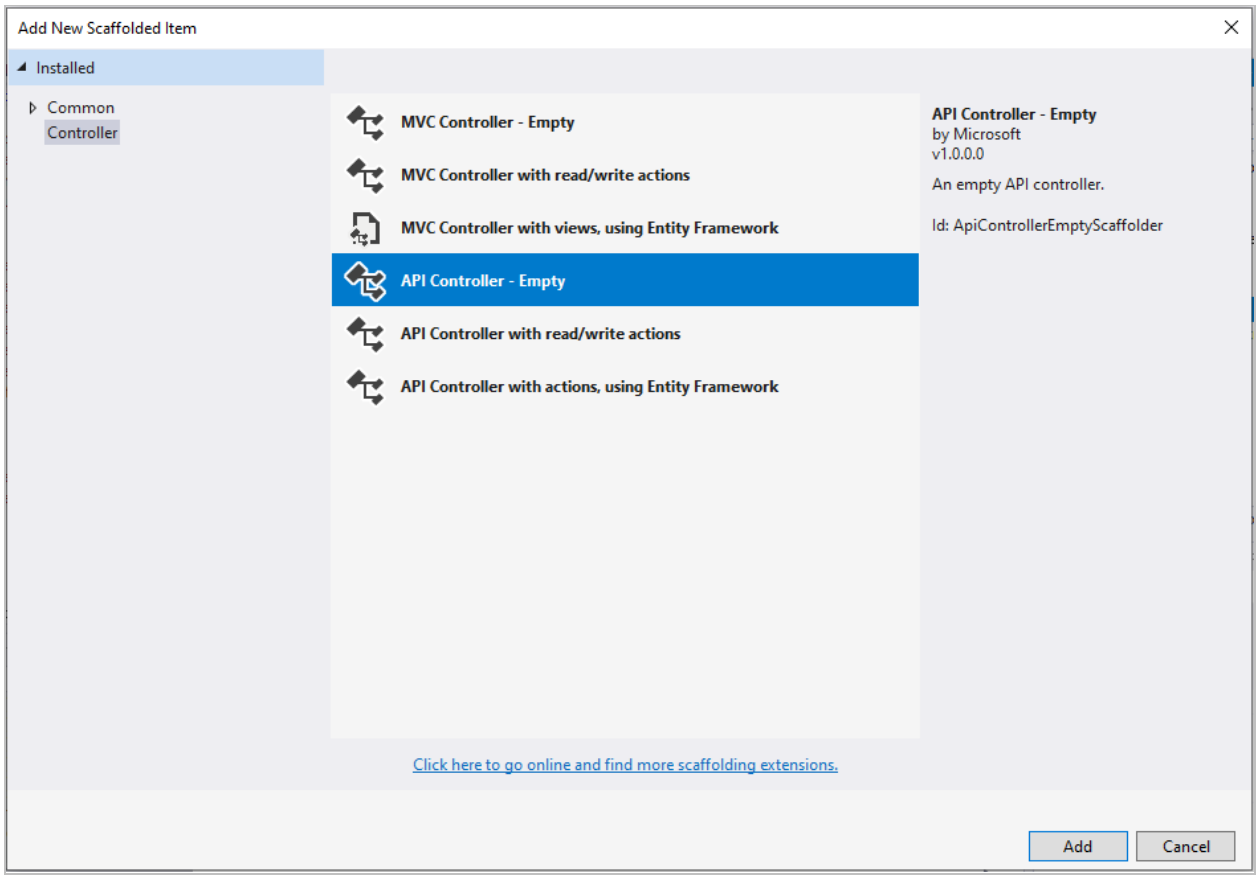
Add the web API controller to the application

1. Right-click the **Server/Controllers** folder and select the controller.



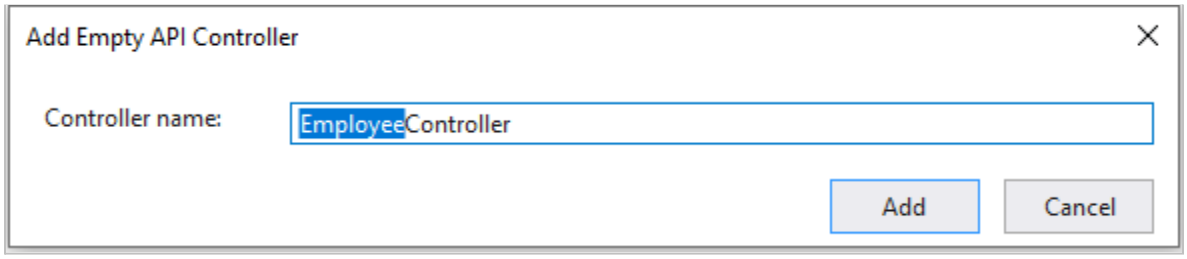
Selecting the File to Add a Controller

2. An **Add New Scaffolded Item** dialog box will open. Select **API Controller – Empty**.



Adding an API Controller

3. An **Add Empty API Controller** dialog box will open. Enter the name **EmployeeController.cs** and click **Add**.



Naming the Controller

Perform CRUD operations

Now, we will modify the controller actions to perform CRUD operations.

Add controller action

Step 1: Inject **IEmployeeAccessLayer** with the employee controller's constructor, as shown in the following screenshot.

```
[ApiController]
public class EmployeeController : ControllerBase
{
    IEmployeAccessLayer _employee;

    public EmployeeController(IEmployeAccessLayer employee)
    {
        _employee = employee;
    }
```

Step 2: Replace the controller with the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using BlazorCrud.Server.DataAccess;
using BlazorCrud.Server.Models;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;


namespace BlazorCrud.Server.Controllers
{
    [ApiController]
    public class EmployeeController :
ControllerBase
    {
```

```
        IEmployeAccessLayer _employee;

        public EmployeeController(IEmployeAccessLayer employee)
        {
            _employee = employee;
        }

        [HttpGet]
        [Route("api/Employee/Index")]
        public IEnumerable<Employee> Index()
        {
            return _employee.GetAllEmployees();
        }
```

```
        [HttpPost]
        [Route("api/Employee/Create")]
        public void Create([FromBody] Employee employee)
        {
            if (ModelState.IsValid)
this._employee.AddEmployee(employee);
        }

        [HttpGet]
        [Route("api/Employee/Details/{id}")]
        public Employee Details(int id)
        {
            return _employee.GetEmployeeData(id);
        }
```

```
37
        [HttpPut]
38
        [Route("api/Employee/Edit")]
39
        public void Edit([FromBody]Employee
40 employee)
        {
41
            if (ModelState.IsValid)
42
this._employee.UpdateEmployee(employee);
43
        }
44
44
        [HttpDelete]
45
        [Route("api/Employee/Delete/{id}")]
46
        public void Delete(int id)
47
        {
48
            _employee.DeleteEmployee(id);
```

```
            }

        }

}
```

The actions and their purposes are explained:

Index: Returns all the employees from database and returns to view.

Details: Returns the employee details from the employee table by employee id. If the employee is not found, then it will return a **Not Found** result.

Create: Accepts employee details as input and creates a new employee in the database.

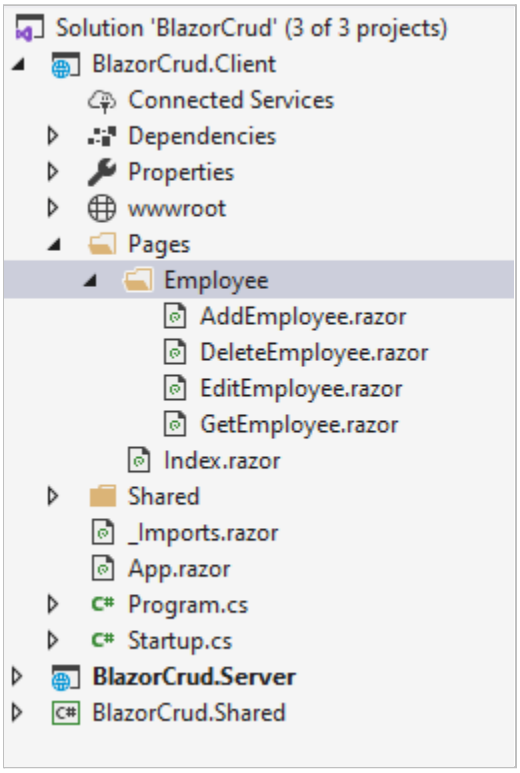
Edit: Accepts the employee ID and details as input and, if the employee is found, then it updates the new details in the database.

Delete: Gets the employee ID as input, requests confirmation, and deletes that employee from database.

Add a Razor view to the application

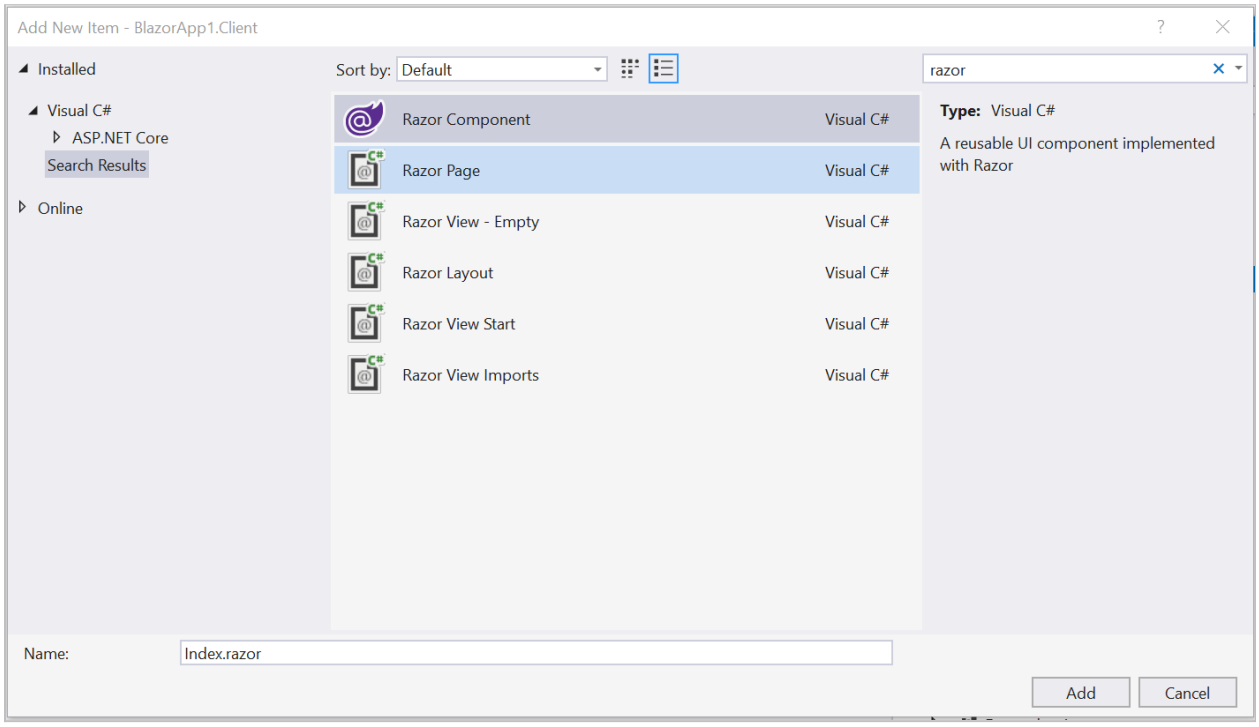
Let’s see how to create Razor pages for CRUD operations.

1. Right-click the **Client/Pages/Employee** folder, and then select **Add**> **New Item**.



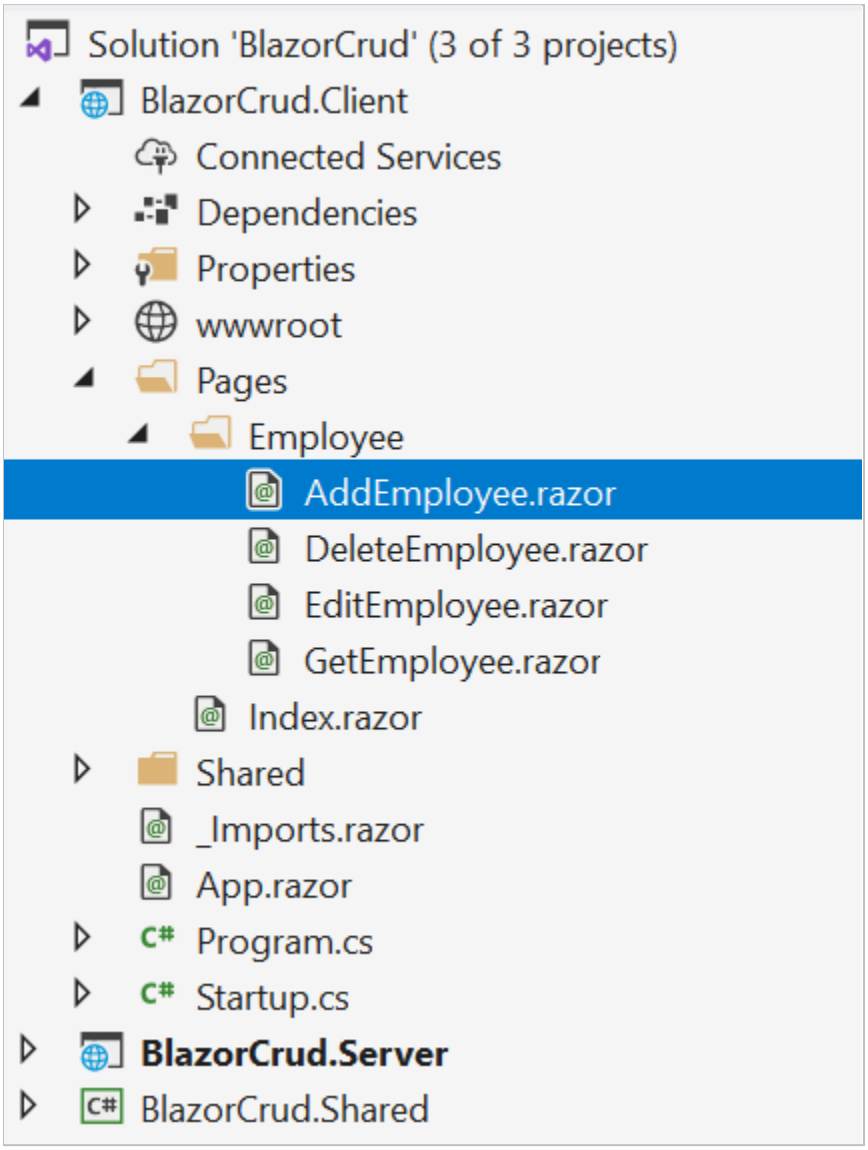
2. An **Add New Item** dialog box will open. Select **Web** from the left panel, and then select **Razor View** from templates listed. Name it

GetEmployee.razor.



3. Follow the above steps and create below other razor files.
    - AddEmployee.razor
    - EditEmployee.razor

- DeleteEmployee.razor



Let's add the code in the Razor files and web API controller to display, create, update, and delete.

Read

An HTTP GET request can be sent to get a resource from the API using the **GetJsonAsync()** method provided by the **HttpClient** class.

Open **GetEmployee.razor** and add the following code in it.

```
@page "/employee"

@inject HttpClient Http


<h1>Employee Data</h1>


<p>

    <a href="/employee/add">Create</a>

</p>


@if (empList == null)

{

    <p><em>Loading...</em></p>

}

else

{
```

```
    <table class='table'>

        <thead>

            <tr>

                <th>ID</th>

                <th>Name</th>

                <th>Designation</th>

                <th>Email</th>

                <th>Location</th>

                <th>Phone</th>

                <th>Action</th>

            </tr>

        </thead>

        <tbody>

            @foreach (var emp in empList)

            {

```

```
                <tr>
                    <td>@emp.EmployeeId</td>
                    <td>@emp.Name</td>
                    <td>@emp.Designation</td>
                    <td>@emp.Email</td>
                    <td>@emp.Location</td>
                    <td>@emp.PhoneNumber</td>
                    <td>
                        <a
href='/employee/edit/@emp.EmployeeId'>Edit</a>  |
                        <a
href='/employee/delete/@emp.EmployeeId'>Delete</a>
                    </td>
                </tr>
            }
        </tbody>
```

```
    </table>

}


@code {

    Employee[] empList;


    protected override async Task OnInitializedAsync()

    {

        empList = await Http.GetJsonAsync<Employee[]>("/api/Employee/Index");

    }

}
```

```
4
9
5
0
5
1
5
2
5
3
5
4
5
5
```

```
1
 [HttpGet]
2
3 [Route("api/Employee/Index")]
4
 public IEnumerable Index()
5
6 {

     return _employee.GetAllEmployees();
```

```
}
```

Create

An HTTP POST request can be sent to add new data in the API using the **SendJsonAsync()** method provided by the **HttpClient** class.

Open **AddEmployee.razor** and add the following code in it.

```
@page "/employee/add"

@inject HttpClient Http

@inject
Microsoft.AspNetCore.Components.NavigationManager
navigation



<h1>Create</h1>



<hr />
<div class="row">
    <div class="col-md-4">
        <form>
            <div class="form-group">
                <label for="Name"
class="control-label">Name</label>
```

```
                <input for="Name" class="form-control" @bind="@emp.Name" />
            </div>
            <div class="form-group">
                <label asp-for="Designation" class="control-label">Designation</label>
                <input for="Designation" class="form-control" @bind="@emp.Designation" />
            </div>
            <div class="form-group">
                <label asp-for="Email" class="control-label">Email</label>
                <input asp-for="Email" class="form-control" @bind="emp.Email" />
            </div>
            <div class="form-group">
```

```
                <label asp-for="Location" class="control-label">Location</label>

                <input asp-for="Location" class="form-control" @bind="@emp.Location" />

            </div>

            <div class="form-group">

                <label asp-for="Phone" class="control-label">Phone</label>

                <input asp-for="Phone" class="form-control" @bind="emp.PhoneNumber" />

            </div>

            <div class="form-group">

                <button type="submit" class="btn btn-default" @onclick="@(async () => await CreateEmployee())">Save</button>

                <button class="btn" @onclick="@cancel">Cancel</button>

            </div>
```

```
37        </form>
    </div>
38</div>
39
40@code {
41
    Employee emp = new Employee();
42
43    protected async Task CreateEmployee()
44    {
45        await Http.SendJsonAsync(HttpMethod.Post, "/api/Employee/Create", emp);
46        navigation.NavigateTo("/employee");
47    }
48
```

```
    void cancel()

    {

        navigation.NavigateTo("/employee");

    }

}
```

```
[HttpPost]

[Route("api/Employee/Create")]

public void Create([FromBody] Employee employee)

{

    if (ModelState.IsValid)

        this.employee.AddEmployee(employee);

}
```

Update

The HTTP PUT method is used to completely replace a resource on the server. We can use the **HttpClient** to send a PUT request to an API using the **SendJsonAsync()** method.

Open **EditEmployee.razor** and add the following code in it.

```
@page "/employee/edit/{empID}"

@inject HttpClient Http

@inject
Microsoft.AspNetCore.Components.NavigationManager
navigation



<h2>Edit</h2>

<h4>Employees</h4>

<hr />

<div class="row">

    <div class="col-md-4">

        <form>

            <div class="form-group">

                <label for="Name"
class="control-label">Name</label>
```

```
                <input for="Name" class="form-control" @bind="@emp.Name" />
            </div>
            <div class="form-group">
                <label asp-for="Designation" class="control-label">Designation</label>
                <input for="Designation" class="form-control" @bind="@emp.Designation" />
            </div>
            <div class="form-group">
                <label asp-for="Email" class="control-label">Email</label>
                <input asp-for="Email" class="form-control" @bind="@emp.Email" />
            </div>
            <div class="form-group">
```

```
                <label asp-for="Location" class="control-label">Location</label>

                <input asp-for="Location" class="form-control" @bind="@emp.Location" />

            </div>

            <div class=" form-group">

                <label asp-for="Phone" class="control-label">Phone</label>

                <input asp-for="Phone" class="form-control" @bind="@emp.PhoneNumber" />

            </div>

            <div class="form-group">

                <input type="submit" value="Save" @onclick="@(async () => await UpdateEmployee())" class="btn btn-default" />

                <input type="submit" value="Cancel" @onclick="@cancel" class="btn" />

            </div>
```

```
37        </form>
    </div>
</div>

@code {

    [Parameter]
    public string empId { get; set; }

    Employee emp = new Employee();

    protected override async Task OnInitializedAsync()
    {
```

```
        emp = await Http.GetJsonAsync<Employee>("/api/Employee/Details/" + Convert.ToInt64(empId));
    }

    protected async Task UpdateEmployee()
    {
        await Http.SendJsonAsync(HttpMethod.Put, "api/Employee/Edit", emp);
        navigation.NavigateTo("/employee");

    }

    void cancel()
    {
        navigation.NavigateTo("/employee");
```

```
    }
}
```

```
[HttpPut]
[Route("api/Employee/Edit")]
public void Edit([FromBody]Employee employee)
{
    if (ModelState.IsValid)
        this.employee.UpdateEmployee(employee);
}
```

Delete

An HTTP DELETE request can be sent to delete a resource from the server using the **DeleteAsync()** method provided by the **HttpClient** class.

Open **DeleteEmployee.razor** and add the following code in it.

```
@page "/employee/delete/{empId}"

@inject HttpClient Http

@inject
Microsoft.AspNetCore.Components.NavigationManager
navigation




<h2>Delete</h2>

<h3>Are you sure you want to delete employee with
id : @empId</h3>

<br />



<div class="col-md-4">

    <table class="table">

        <tr>

            <td>Name</td>
```

```
            <td>@emp.Name</td>
        </tr>
        <tr>
            <td>Designation</td>
            <td>@emp.Designation</td>
        </tr>
        <tr>
            <td>Email</td>
            <td>@emp.Email</td>
        </tr>
        <tr>
            <td>Location</td>
            <td>@emp.Location</td>
        </tr>
        <tr>
```

```
            <td>Phone</td>
            <td>@emp.PhoneNumber</td>
        </tr>
    </table>
    <div class="form-group">
        <input type="submit" value="Delete" @onclick="@(async () => await Delete())" class="btn btn-default" />
        <input type="submit" value="Cancel" @onclick="@cancel" class="btn" />
    </div>
</div>


@code {


    [Parameter]
```

```
37     public string empId { get; set; }

38     Employee emp = new Employee();

40     protected override async Task OnInitializedAsync()

41     {

42         emp = await Http.GetJsonAsync<Employee>

43         ("/api/Employee/Details/" + Convert.ToInt64(empId));

44     }

46     protected async Task Delete()

47     {

48
```

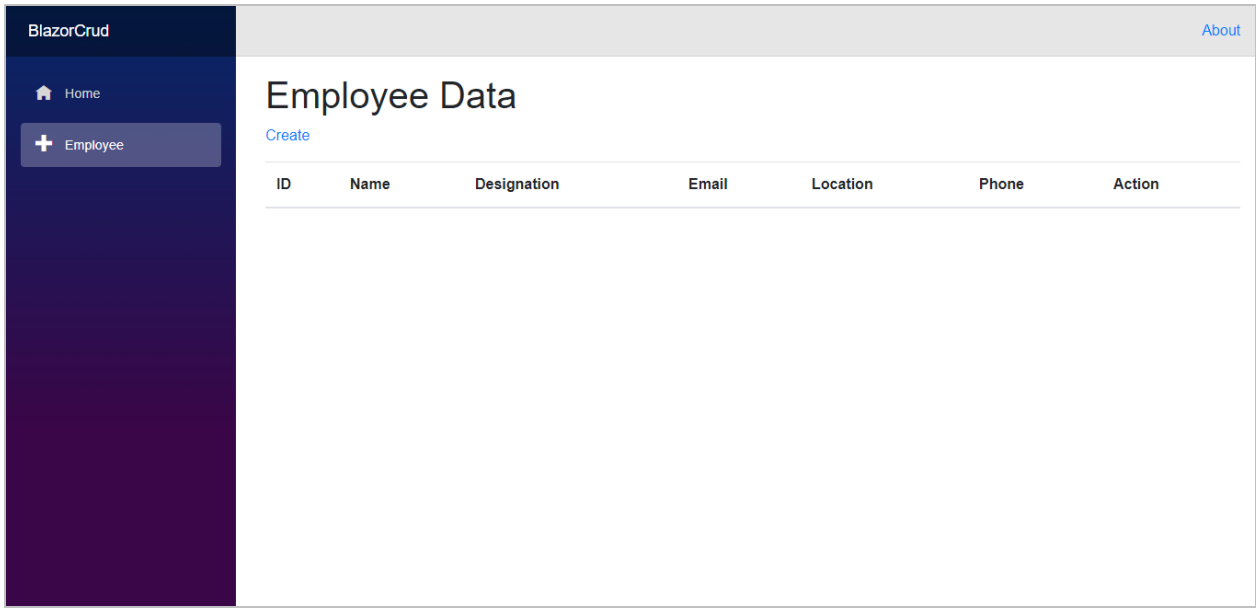
```
4
        await
9 Http.DeleteAsync("api/Employee/Delete/" +
5 Convert.ToInt64(empId));
0
        navigation.NavigateTo("/employee");
5
1     }
5
2
5
    void cancel()
3
5     {
4
        navigation.NavigateTo("/employee");
5
5     }
5
6
5 }
7
5
8
5
9
6
0
```

```
6
1
6
2
6
3
```

```
[HttpDelete]

[Route("api/Employee/Delete/{id}")]

public void Delete(int id)

{

    employee.DeleteEmployee(id);

}
```

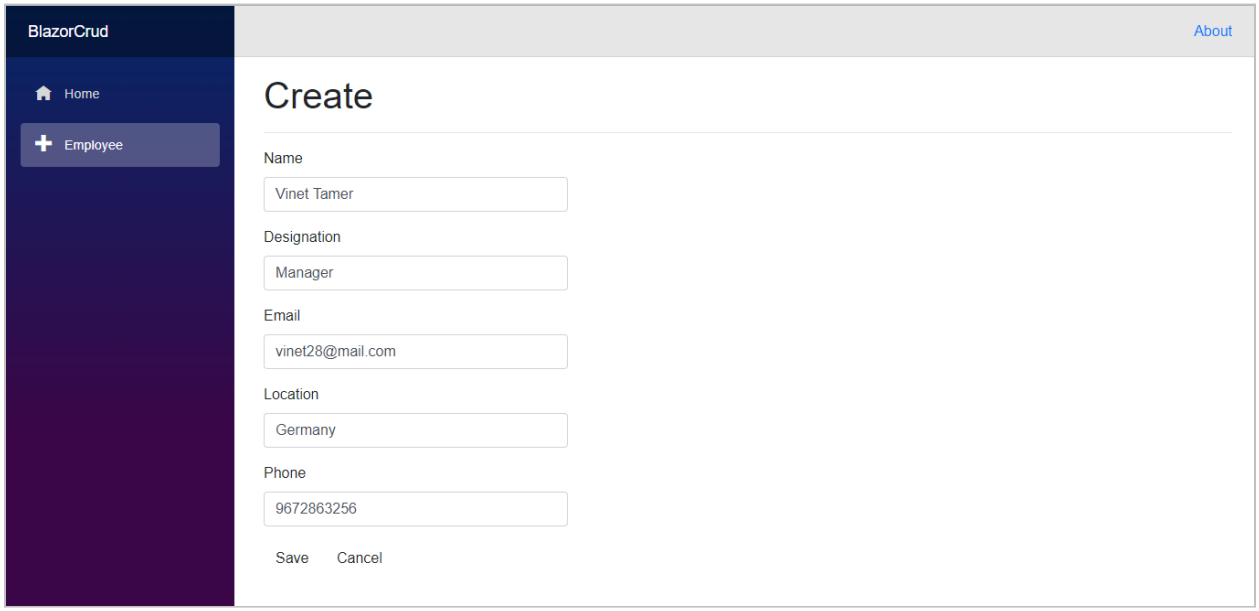
Run the application

1. Click **Run** to view the application. The page will be opened in a new browser window.
2. Click **Employee** in navigation menu. This will direct you to the Employee page. Initially this page will be empty.



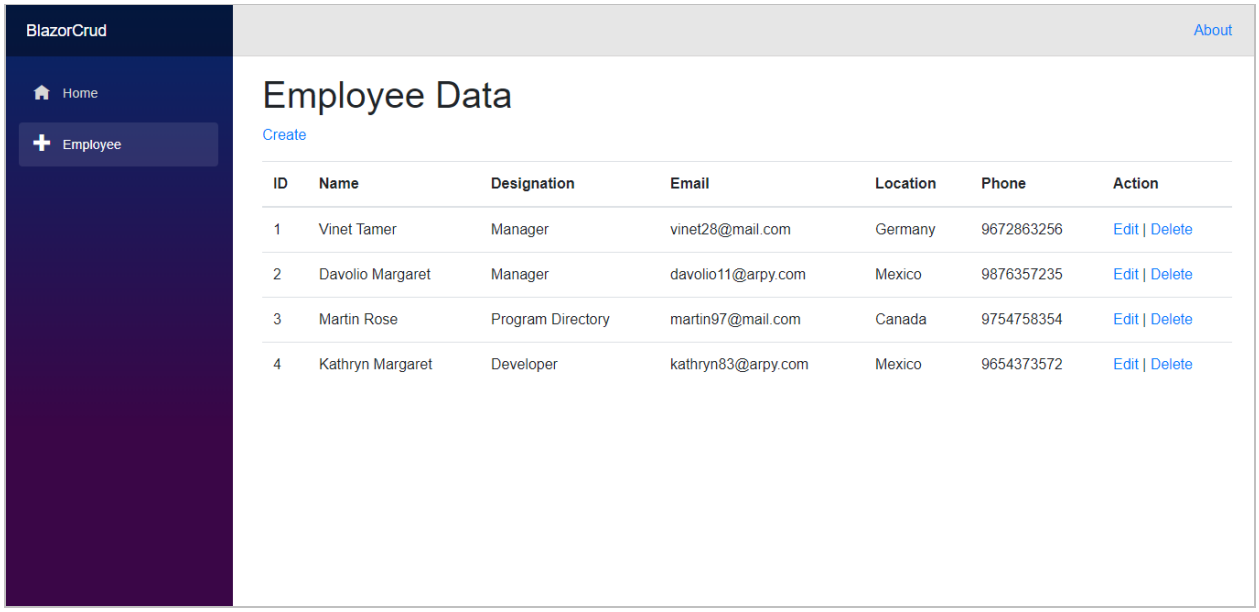
Empty Employee Page

3. Click the **Create** link to create a new employee entry. Enter the employee details and click **Save**.



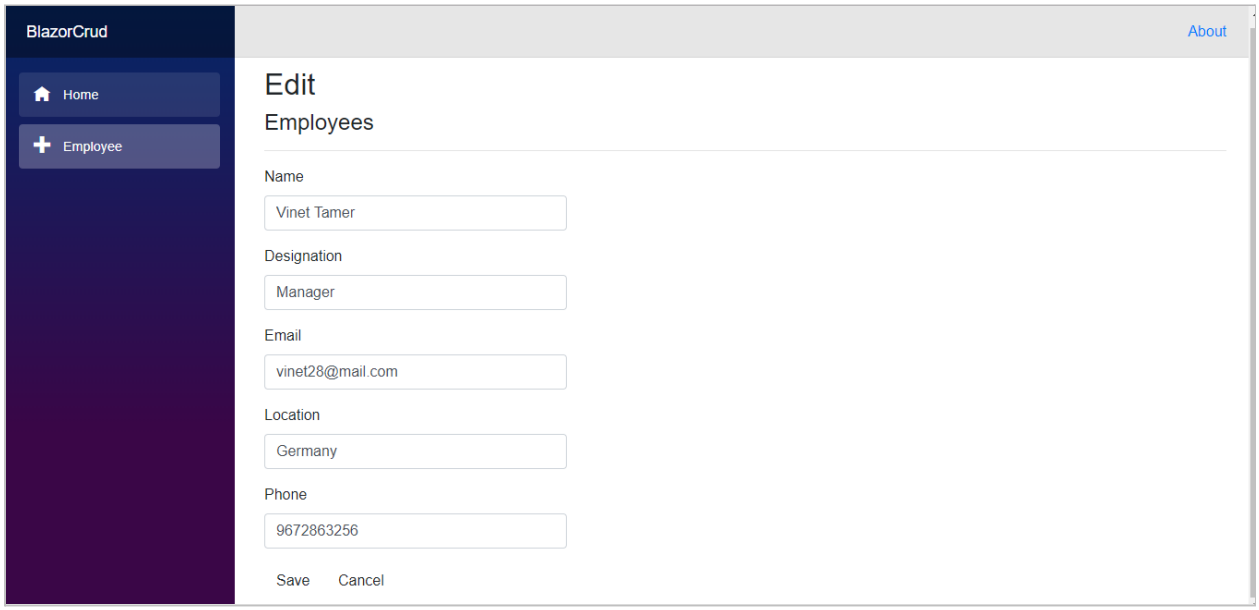
Employee Page with a New Employee Entry

4. Now, the created employee details are displayed as shown in the following screenshot.



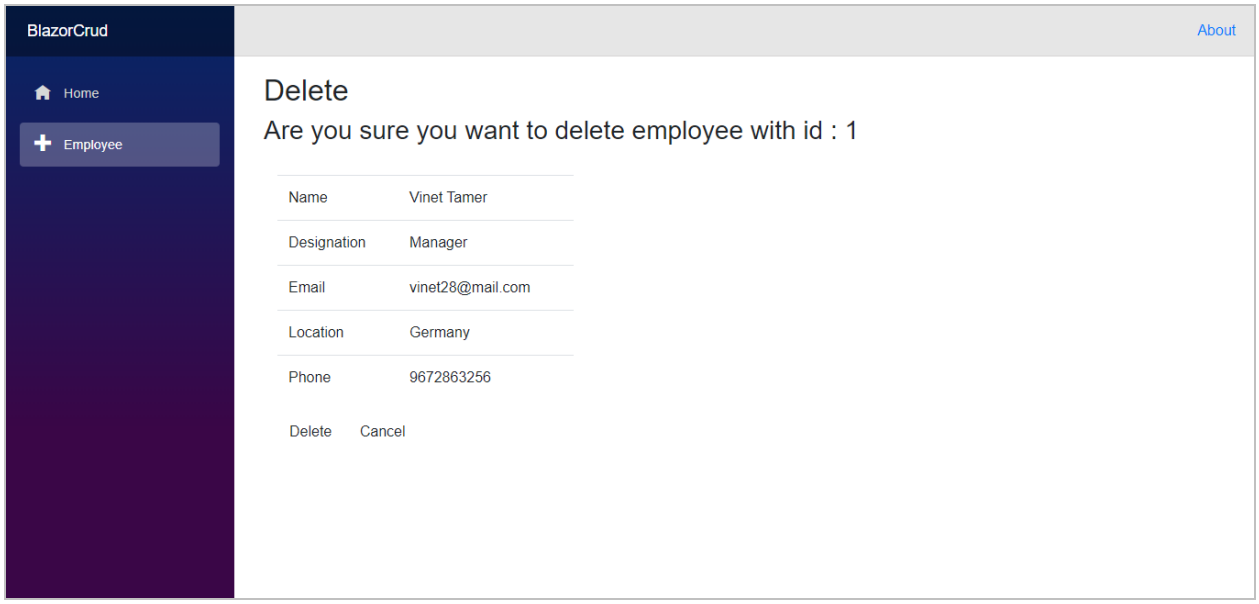
Application with Employees Added

5. Click **Edit** to update the employee details.



Updating Employee Information

6. Click **Delete** to delete a product. A confirmation alert will appear before it is deleted from the database.



Deleting an Employee Record