

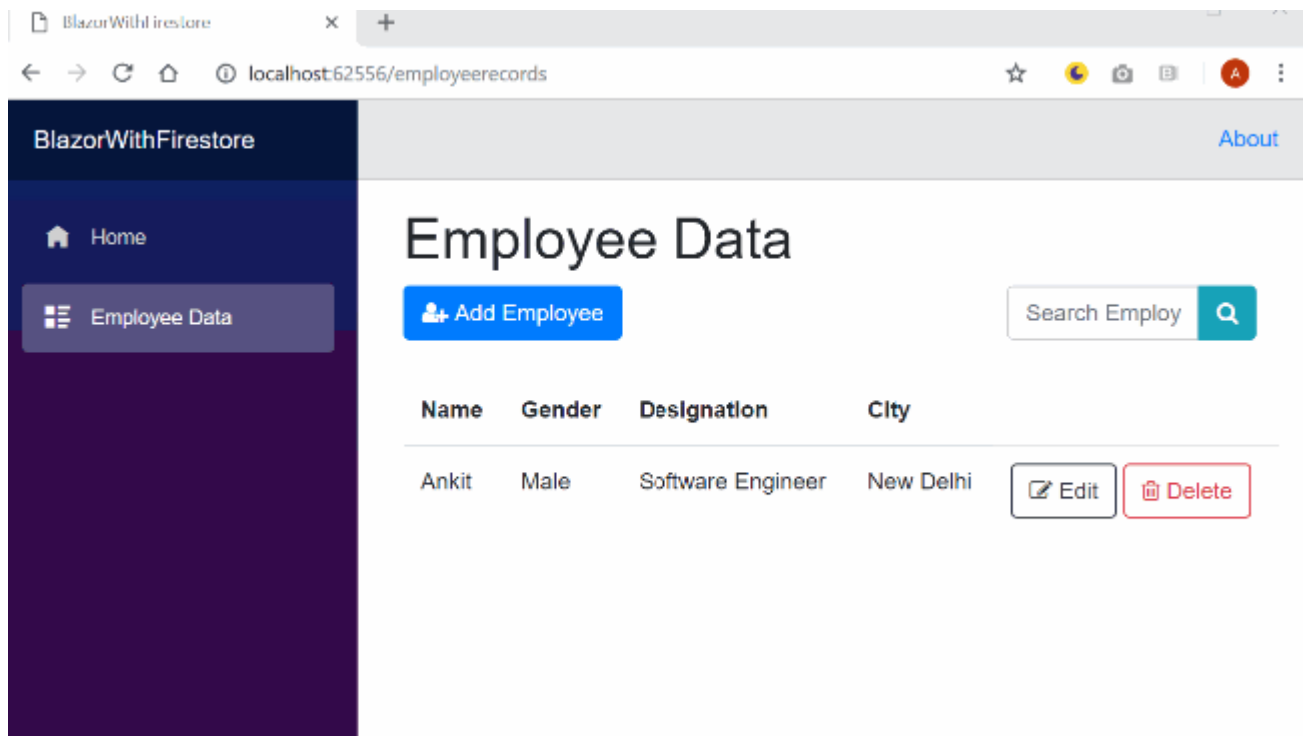
Blazor CRUD Using Google Cloud Firestore

February 25, 2019 [Ankit Sharma](#) [ASP.NET Core](#), [Blazor](#)

Introduction

In this article, we will create a Blazor application using Google Firestore as database provider. We will create a Single Page Application (SPA) and perform CRUD operations on it. We will use Bootstrap 4 to display a modal popup for handling user inputs. The form also has a dropdown list, which will bind to a collection in our database. We will also implement a client-side search functionality to search the employee list by employee name.

Take a look at the final application.



Prerequisites

- Install the .NET Core 2.1 or above SDK from [here](#)
- Install latest version of Visual Studio 2017 from <https://www.visualstudio.com/downloads/>
- Install ASP.NET Core Blazor Language Services extension from [here](#)

Source Code

The source code has been updated to .NET Core 3.2 Preview-1. Get the source code from [GitHub](#).




Configuring Cloud Firestore

The first step is to create a project in google Firebase console. Navigate to <https://console.firebase.google.com> and sign-in with your google account. Click on Add Project link. A pop up window will open as shown in the image below. Provide your project name and click on Create project button at the bottom.

Add a project ✕


Project name

BlazorWithFirestore

 +  + 


Tip: Projects span apps across platforms ?

Project ID ?

blazorwithfirestore 

Locations ?

United States (Analytics)

nam5 (us-central) (Cloud Firestore) 

☒ Use the default settings for sharing Google Analytics for Firebase data

- ✓ Share your Analytics data with all Firebase features
- ✓ Share your Analytics data with Google to improve Google Products and Services
- ✓ Share your Analytics data with Google to enable technical support
- ✓ Share your Analytics data with Google to enable Benchmarking
- ✓ Share your Analytics data with Google Account Specialists

☒ I accept the [controller-controller terms](#). This is required when sharing Analytics data to improve Google Products and Services. [Learn more](#)

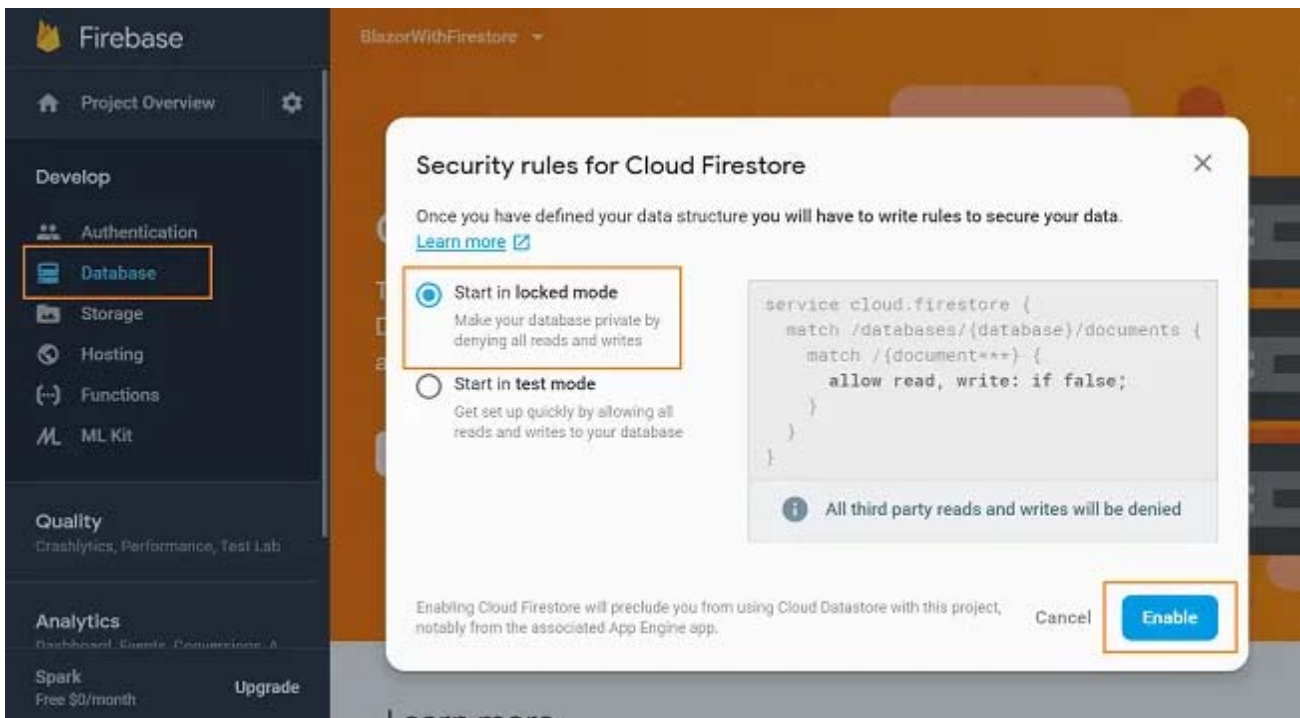
Cancel

Create project

Note the project id here. Firebase project ids are globally unique. You can edit your project id while creating a new project. Once the project is created you cannot change your project id. We will use this project id in next section while initializing our application.

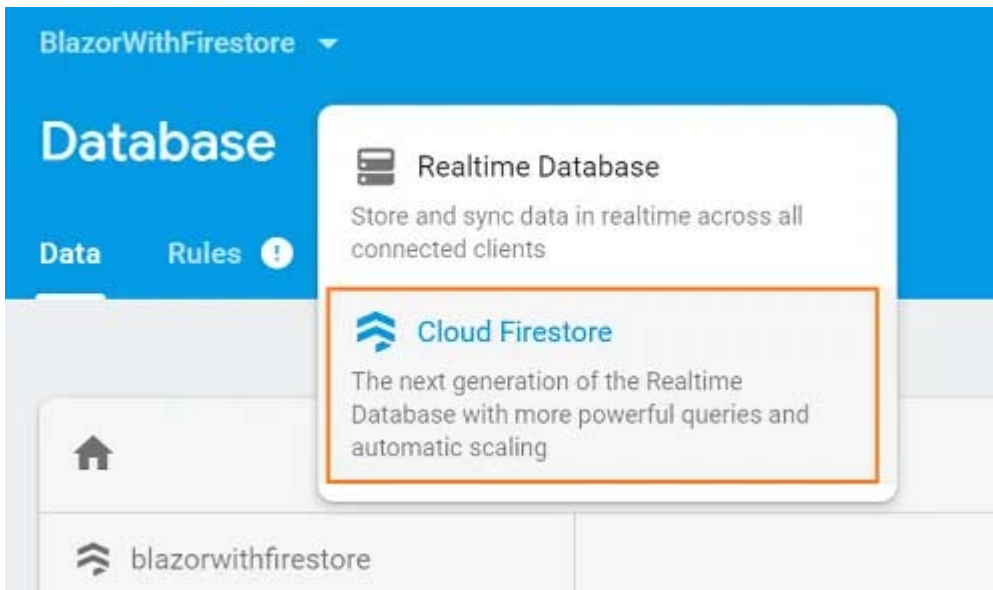
Click on the project you just created. A project overview page will open. Select “Database” from left menu. Then click on “Create database” button. A popup window will open asking you to select the “Security rules for Cloud Firestore”. Select “Start in locked mode” and click on enable.

Refer to the image below:



This will enable the database for your project. Firebase project have two options for database – Realtime Database and Cloud Firestore. For this application, we will use “Cloud Firestore” database. Click on “Database” dropdown at the top of the page and select “Cloud Firestore”.

Refer to the image below:



We will create cities collection to store the city name for employees. We will also bind this collection to a dropdown list in our web application from which the user will select the desired city. Click on “Add collection”. Set the collection ID as “cities”. Click on “Next”. Refer to the image below:

Start a collection

1 Set collection ID — 2 Add first document

Parent path

/

Collection ID

cities

Cancel Next

Put the Field value as “CityName”, Select string from the Type dropdown and fill the value with city name as “Mumbai”. Click on Save. Refer to the image below:

Start a collection

✓ Set collection ID — 2 Add first document

Document parent path

/cities

Document ID

Auto-ID

Field	Type	Value
CityName	string	Mumbai

+ Add field

Cancel Save

This will create the “cities” collection and insert the first document in it. Similarly, create four more documents inside this collection and put the “CityName” value as Chennai, New Delhi, Bengaluru and Hyderabad.

We will use “employees” collection to store employee data, but we will not create it manually. We will create “employees” collection while adding the first employee data from the application.

Configuring Google Application Credentials

To access database from our project, we need to set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to point to a JSON service account key file. This will set an authentication pipeline from our application to cloud Firestore.

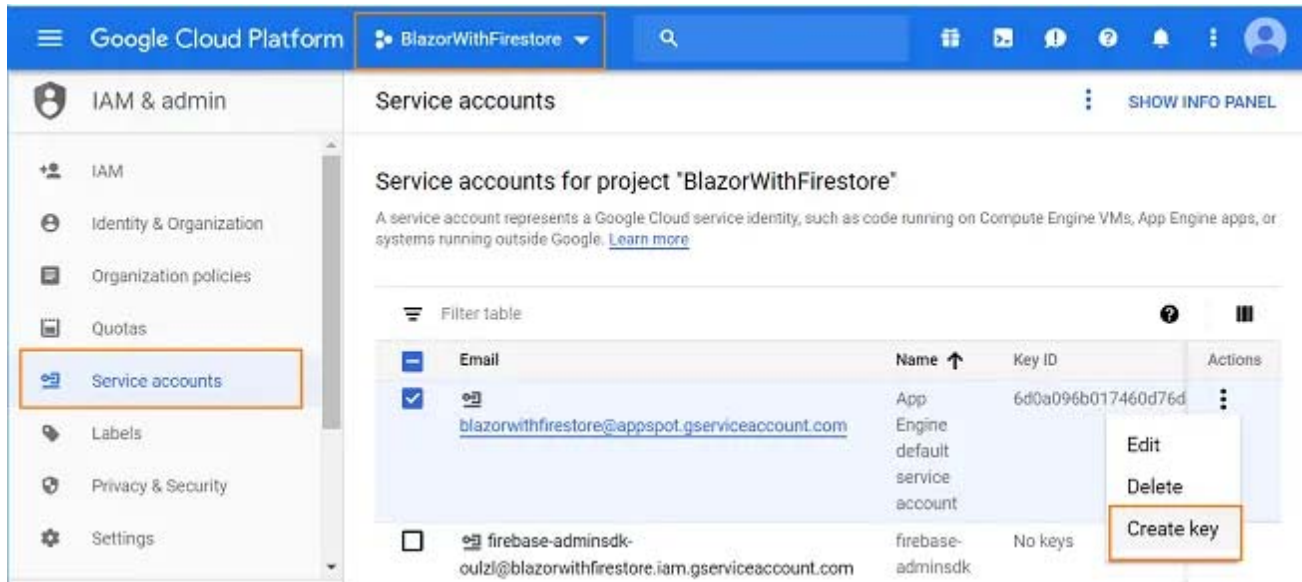
To generate the service account key file follow the steps mentioned below:

Step 1: Navigate to <https://console.cloud.google.com/iam-admin/>. Login with the same google account, you have used to create Firestore DB.

Step 2: Select Project from the drop down in menu bar at the top.

Step 3: Select “Service accounts” from the left menu. Select the service account for which you want to create the key. Click on more button in the “Actions” column in that row, and then click Create key.

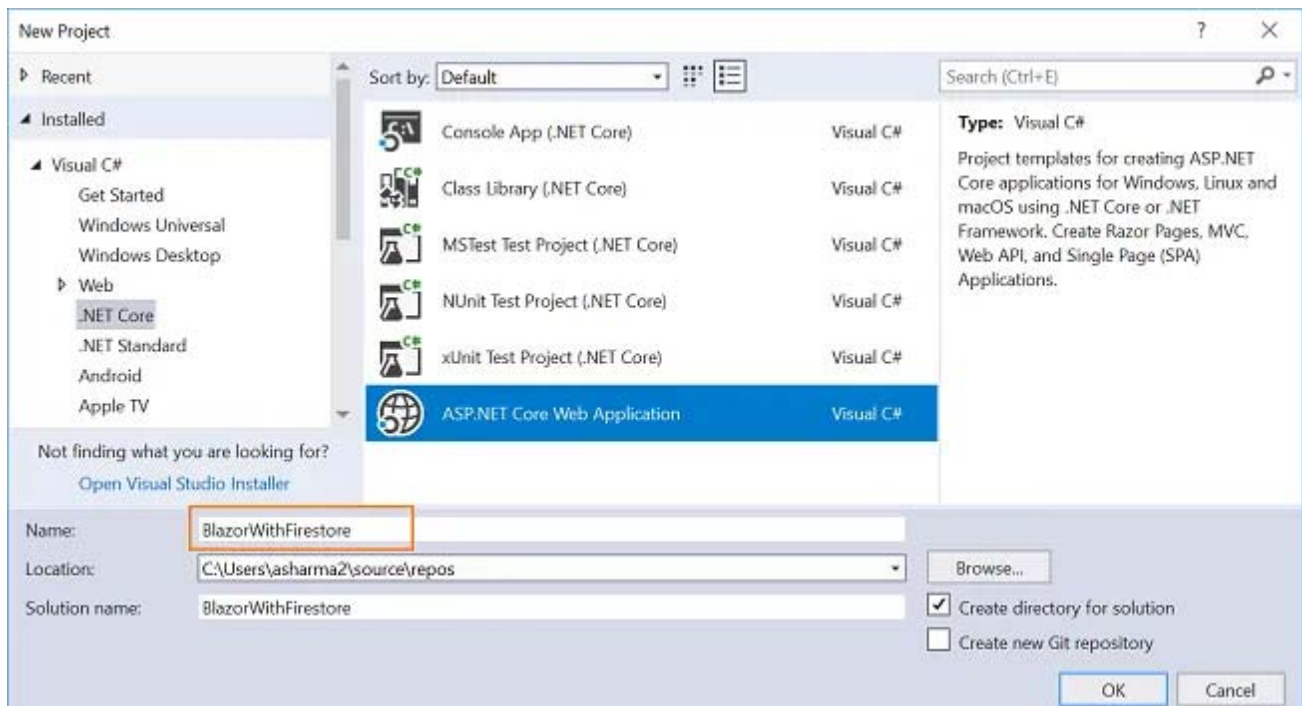
Refer to the image below:



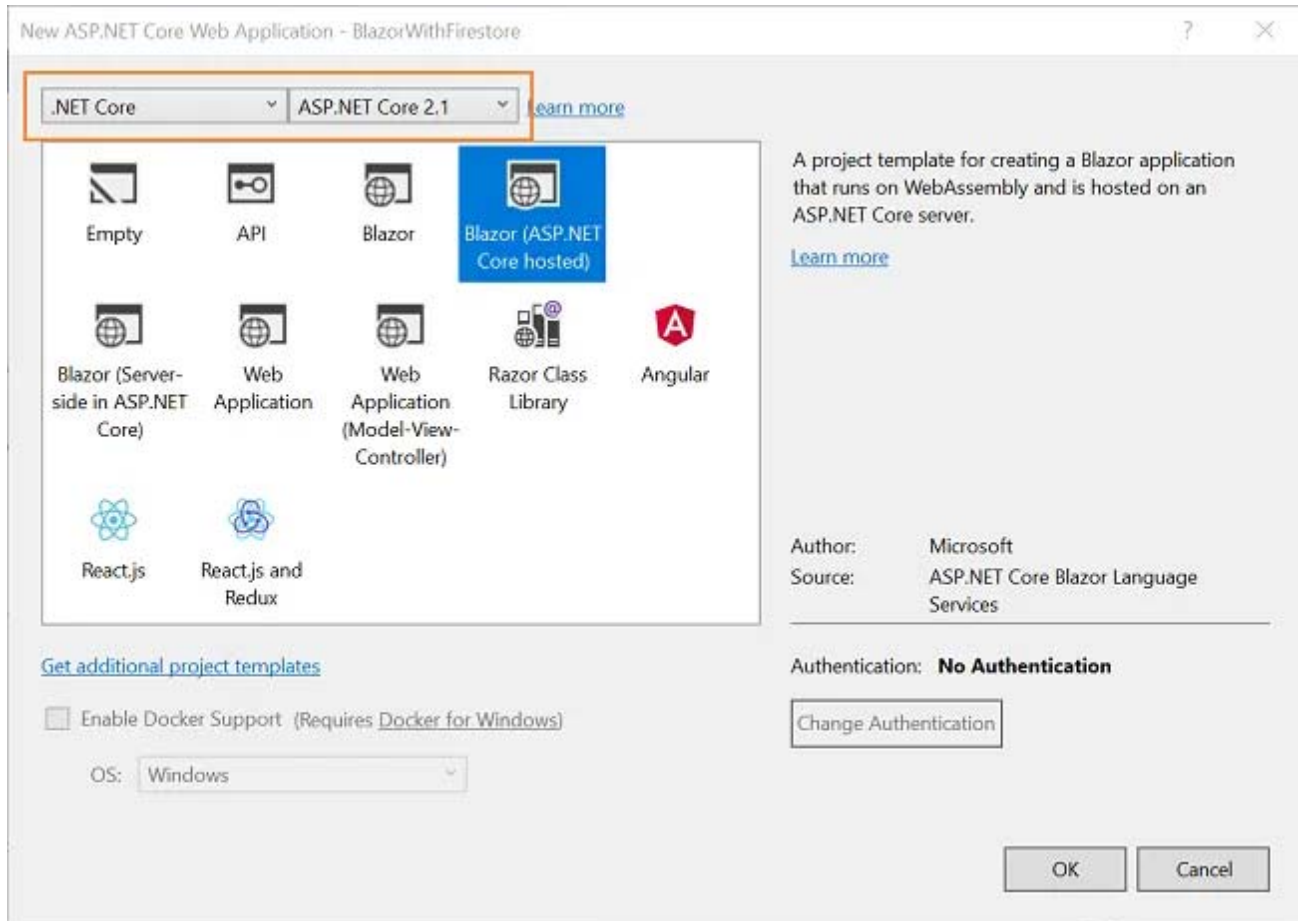
Step 4: A popup modal will open asking you to select the key type. Select “JSON” and click on create button. This will create private key for accessing your Firestore account and downloads a JSON key file to your machine. We will use this file to set `GOOGLE_APPLICATION_CREDENTIALS` environment variable in later part of this article.

Create Blazor Web Application

Open Visual Studio and select File >> New >> Project. After selecting the project, a “New Project” dialog will open. Select .NET Core inside Visual C# menu from the left panel. Then, select “ASP.NET Core Web Application” from available project types. Put the name of the project as `BlazorWithFirestore` and press OK.



After clicking on OK, a new dialog will open asking you to select the project template. You can observe two drop-down menus at the top left of the template window. Select “.NET Core” and “ASP.NET Core 2.1” from these dropdowns. Then, select “Blazor (ASP .NET Core hosted)” template and press OK.



Now, our Blazor solution will be created. You can observe that we have three project files created in this solution.

1. BlazorWithFirestore.Client – It has the client side code and contains the pages that will be rendered on the browser.
2. BlazorWithFirestore.Server – It has the server side codes such as data access layer and web API.
3. BlazorWithFirestore.Shared – It contains the shared code that can be accessed by both client and server. It contains our Model classes.

Adding Package reference for Firestore

We need to add the package reference for Google cloud Firestore, which will allow us to access our DB from the Blazor application. Right click on **BlazorWithFirestore.Shared** project. Select “Edit BlazorWithFirestore.Shared.csproj”. It will open the **BlazorWithFirestore.Shared.csproj** file. Add the following lines inside it.

```
<ItemGroup>
  <PackageReference Include="Google.Cloud.Firestore" Version="1.0.0-beta14" /><br></ItemGroup>
```

Similarly add these lines to **BlazorWithFirestore.Server.csproj** file also.

Creating the Model

We will create our model class in **BlazorWithFirestore.Shared** project. Right click on **BlazorWithFirestore.Shared** and select Add >> New Folder. Name the folder as Models. Again, right click on Models folder and select Add >> Class to add a new class file. Put the name of you class as Employee.cs and click Add.

Open the **Employee.cs** class and put the following code into it.

```
using System;
using Google.Cloud.Firestore;
namespace BlazorWithFirestore.Shared.Models
{
    [FirestoreData]
    public class Employee
```

```

{
    public string EmployeeId { get; set; }
    public DateTime date { get; set; }
    [FirestoreProperty]
    public string EmployeeName { get; set; }
    [FirestoreProperty]
    public string CityName { get; set; }
    [FirestoreProperty]
    public string Designation { get; set; }
    [FirestoreProperty]
    public string Gender { get; set; }
}
}

```

We have decorated the class with `[FirestoreData]` attribute. This will allow us to map this class object to Firestore collection. Only those class properties, which are marked with `[FirestoreProperty]` attribute, are considered when we are saving the document to our collection. We do not need to save EmployeeId to our database as it is generated automatically. While fetching the data, we will bind the auto generated document id to the EmployeeId property. Similarly, we will use date property to bind the created date of collection while fetching the record. We will use this date property to sort the list of employees by created date. Hence, we have not applied `[FirestoreProperty]` attribute to these two properties.

Similarly, create a class file Cities.cs and put the following code into it.

```

using System;
using Google.Cloud.Firestore;
namespace BlazorWithFirestore.Shared.Models
{
    [FirestoreData]
    public class Cities
    {
        public string CityName { get; set; }
    }
}

```

Creating Data Access Layer for the Application

Right-click on `BlazorWithFirestore.Server` project and then select Add >> New Folder and name the folder as `DataAccess`. We will be adding our class to handle database related operations inside this folder only. Right click on DataAccess folder and select Add >> Class. Name your class `EmployeeDataAccessLayer.cs`.

Put the following code inside this class.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using BlazorWithFirestore.Shared.Models;
using Google.Cloud.Firestore;
using Newtonsoft.Json;
namespace BlazorWithFirestore.Server.DataAccess
{
    public class EmployeeDataAccessLayer
    {
        string projectId;
        FirestoreDb fireStoreDb;
        public EmployeeDataAccessLayer()
        {
            string filepath = "C:\\FirestoreAPIKey\\blazorwithfirestore-6d0a096b0174.json";
            Environment.SetEnvironmentVariable("GOOGLE_APPLICATION_CREDENTIALS", filepath);
            projectId = "blazorwithfirestore";
            fireStoreDb = FirestoreDb.Create(projectId);
        }
        public async Task<List<Employee>> GetAllEmployees()
        {

```

```

try
{
    Query employeeQuery = firestoreDb.Collection("employees");
    QuerySnapshot employeeQuerySnapshot = await employeeQuery.GetSnapshotAsync();
    List<Employee> lstEmployee = new List<Employee>();
    foreach (DocumentSnapshot documentSnapshot in employeeQuerySnapshot.Documents)
    {
        if (documentSnapshot.Exists)
        {
            Dictionary<string, object> city = documentSnapshot.ToDictionary();
            string json = JsonConvert.SerializeObject(city);
            Employee newuser = JsonConvert.DeserializeObject<Employee>(json);
            newuser.EmployeeId = documentSnapshot.Id;
            newuser.date = documentSnapshot.CreateTime.Value.ToDateTime();
            lstEmployee.Add(newuser);
        }
    }
    List<Employee> sortedEmployeeList = lstEmployee.OrderBy(x => x.date).ToList();
    return sortedEmployeeList;
}
catch
{
    throw;
}
}

public async void AddEmployee(Employee employee)
{
    try
    {
        CollectionReference colRef = firestoreDb.Collection("employees");
        await colRef.AddAsync(employee);
    }
    catch
    {
        throw;
    }
}

public async void UpdateEmployee(Employee employee)
{
    try
    {
        DocumentReference empRef =
firestoreDb.Collection("employees").Document(employee.EmployeeId);
        await empRef.SetAsync(employee, SetOptions.Overwrite);
    }
    catch
    {
        throw;
    }
}

public async Task<Employee> GetEmployeeData(string id)
{
    try
    {
        DocumentReference docRef = firestoreDb.Collection("employees").Document(id);
        DocumentSnapshot snapshot = await docRef.GetSnapshotAsync();
        if (snapshot.Exists)
        {
            Employee emp = snapshot.ConvertTo<Employee>();
            emp.EmployeeId = snapshot.Id;
            return emp;
        }
        else
        {
            return new Employee();
        }
    }
}

```



```

    }
}
catch
{
    throw;
}
}
public async void DeleteEmployee(string id)
{
    try
    {
        DocumentReference empRef = firestoreDb.Collection("employees").Document(id);
        await empRef.DeleteAsync();
    }
    catch
    {
        throw;
    }
}
public async Task<List<Cities>> GetCityData()
{
    try
    {
        Query citiesQuery = firestoreDb.Collection("cities");
        QuerySnapshot citiesQuerySnapshot = await citiesQuery.GetSnapshotAsync();
        List<Cities> lstCity = new List<Cities>();
        foreach (DocumentSnapshot documentSnapshot in citiesQuerySnapshot.Documents)
        {
            if (documentSnapshot.Exists)
            {
                Dictionary<string, object> city = documentSnapshot.ToDictionary();
                string json = JsonConvert.SerializeObject(city);
                Cities newCity = JsonConvert.DeserializeObject<Cities>(json);
                lstCity.Add(newCity);
            }
        }
        return lstCity;
    }
    catch
    {
        throw;
    }
}
}
}

```

In the constructor of this class we are setting the `GOOGLE_APPLICATION_CREDENTIALS` environment variable. You need to set the value of filepath variable to the path where the JSON service account key file is located in your machine. Remember we downloaded this file in the previous section. The projectId variable should be set to the project id of your Firebase project.

We have also defined the methods for performing CRUD operations. The GetAllEmployees method will fetch the list of all employee document from our “employees” collection. It will return the employee list sorted by document creation date.

The `AddEmployee` method will add a new employee document to our “employees” collection. If the collection does not exist, it will create the collection first then insert a new document in it.

The `UpdateEmployee` method will update the field values of an already existing employee document, based on the employee id passed to it. We are binding the document id to employeeId property, hence we can easily manipulate the documents.

The `GetEmployeeData` method will fetch a single employee document from our “employees” collection based on the employee id.

`DeleteEmployee` method will delete the document for a particular employee from the “employees” collection.

`GetCityData` method will return the list of cities from “cities” collection.

Adding the web API Controller to the Application

Right-click on **BlazorWithFirestore.Server/Controllers** folder and select Add >> New Item. An “Add New Item” dialog box will open. Select Web from the left panel, then select “API Controller Class” from templates panel and put the name as **EmployeeController.cs**. Click Add.

This will create our API EmployeeController class. We will call the methods of EmployeeDataAccessLayer class to fetch data and pass on the data to the client side.

Open **EmployeeController.cs** file and put the following code into it.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using BlazorWithFirestore.Server.DataAccess;
using BlazorWithFirestore.Shared.Models;
using Microsoft.AspNetCore.Mvc;
namespace BlazorWithFirestore.Server.Controllers
{
    [Route("api/[controller]")]
    public class EmployeeController : Controller
    {
        EmployeeDataAccessLayer objemployee = new EmployeeDataAccessLayer();

        [HttpGet]
        public Task<List<Employee>> Get()
        {
            return objemployee.GetAllEmployees();
        }

        [HttpGet("{id}")]
        public Task<Employee> Get(string id)
        {
            return objemployee.GetEmployeeData(id);
        }

        [HttpPost]
        public void Post([FromBody] Employee employee)
        {
            objemployee.AddEmployee(employee);
        }

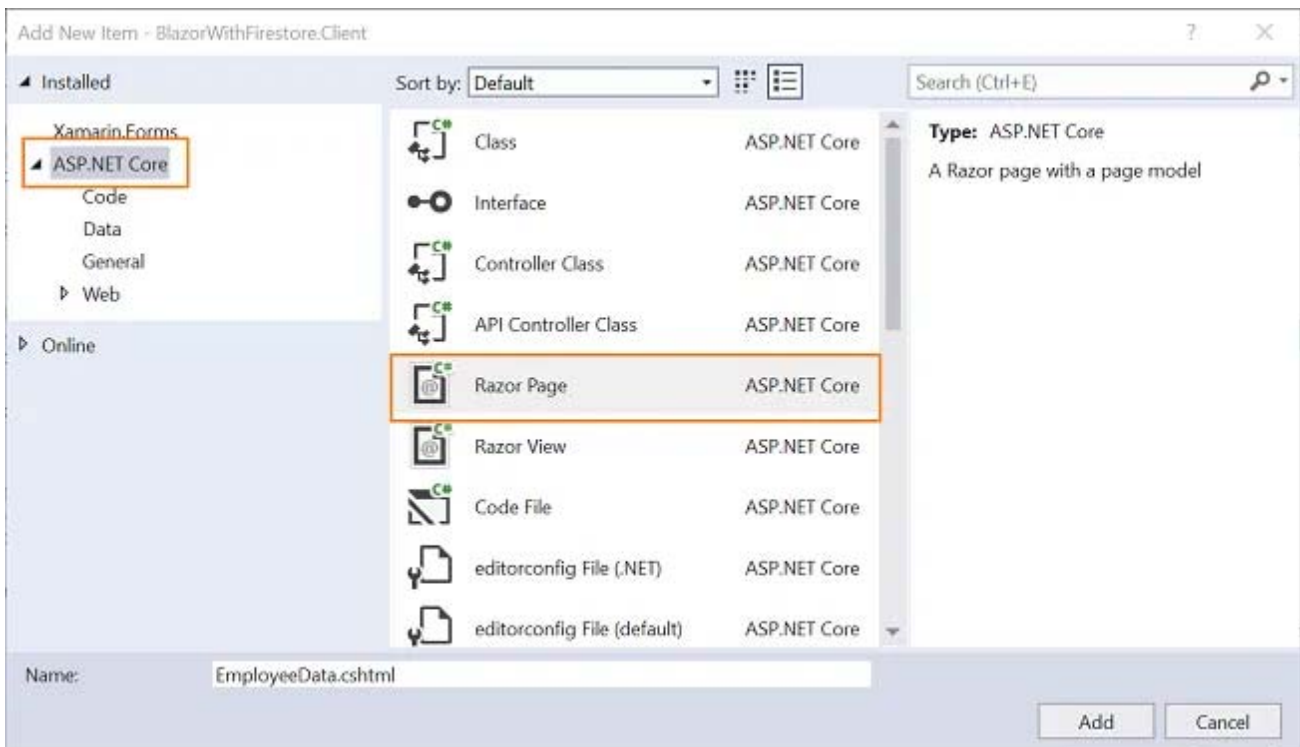
        [HttpPut]
        public void Put([FromBody] Employee employee)
        {
            objemployee.UpdateEmployee(employee);
        }

        [HttpDelete("{id}")]
        public void Delete(string id)
        {
            objemployee.DeleteEmployee(id);
        }

        [HttpGet("GetCities")]
        public Task<List<Cities>> GetCities()
        {
            return objemployee.GetCityData();
        }
    }
}
```

Creating the Blazor component

We will create the component in the **BlazorWithFirestore.Client/Pages** folder. The application template provides the Counter and Fetch Data files by default in this folder. Before adding our own component file, we will delete these two default files to make our solution cleaner. Right-click on **BlazorWithFirestore.Client/Pages** folder and then select Add >> New Item. An “Add New Item” dialog box will open, select “ASP.NET Core” from the left panel, then select “Razor Page” from templates panel and name it **EmployeeData.cshtml**. Click Add. Refer to the image below:



This will add an `EmployeeData.cshtml` page to our `BlazorSPA.Client/Pages` folder. This razor page will have two files – `EmployeeData.cshtml` and `EmployeeData.cshtml.cs`.

Adding references for JS Interop

We will be using a bootstrap modal dialog in our application. We will also include a few Font Awesome icons for styling in the application. To be able to use these two libraries, we need to add the CDN references to allow the JS interop.

```
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js">
</script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"></script>
```

Here, we have included the CDN references, which will allow us to use the bootstrap modal dialog and Font Awesome icons in our applications. Now, we will add codes to our view files.

EmployeeData.cshtml.cs

Open `EmployeeData.cshtml.cs` and put the following code into it.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using BlazorWithFirestore.Shared.Models;
using Microsoft.AspNetCore.Blazor;
using Microsoft.AspNetCore.Blazor.Components;
namespace BlazorWithFirestore.Client.Pages
{
    public class EmployeeDataModel : BlazorComponent
    {
        [Inject]
        protected HttpClient Http { get; set; }
        protected List<Employee> empList = new List<Employee>();
        protected List<Cities> cityList = new List<Cities>();
        protected Employee emp = new Employee();
        protected string modalTitle { get; set; }
    }
}
```

```

protected string searchString { get; set; }
protected override async Task OnInitAsync()
{
    await GetCityList();
    await GetEmployeeList();
}
protected async Task GetCityList()
{
    cityList = await Http.GetJsonAsync<List<Cities>>("api/Employee/GetCities");
}
protected async Task GetEmployeeList()
{
    empList = await Http.GetJsonAsync<List<Employee>>("api/Employee");
}
protected void AddEmployee()
{
    emp = new Employee();
    modalTitle = "Add Employee";
}
protected async Task EditEmployee(string empID)
{
    emp = await Http.GetJsonAsync<Employee>("/api/Employee/" + empID);
    modalTitle = "Edit Employee";
}
protected async Task SaveEmployee()
{
    if (emp.EmployeeId != null)
    {
        await Http.SendJsonAsync(HttpMethod.Put, "api/Employee/", emp);
    }
    else
    {
        await Http.SendJsonAsync(HttpMethod.Post, "/api/Employee/", emp);
    }
    await GetEmployeeList();
}
protected async Task DeleteConfirm(string empID)
{
    emp = await Http.GetJsonAsync<Employee>("/api/Employee/" + empID);
}
protected async Task DeleteEmployee(string empID)
{
    Console.WriteLine(empID);
    await Http.DeleteAsync("api/Employee/" + empID);
    await GetEmployeeList();
}
protected async Task SearchEmployee()
{
    await GetEmployeeList();
    if (searchString != "")
    {
        empList = empList.Where(
            x => x.EmployeeName.IndexOf(searchString,
                StringComparison.OrdinalIgnoreCase) != -1).ToList();
    }
}
}

```

Here, we have defined the EmployeeDataModel class, which is inheriting from BlazorComponent. This allows the EmployeeDataModel class to act as a Blazor component.

We are also injecting the HttpClient service to enable the web API calls to our EmployeeController API.

We will use the two variable – empList and cityList to hold the data of our Employee and Cities collections respectively. The modalTitle property, which is of type string, is used to hold the title that will be displayed in the modal dialog. The value provided in the search box is stored in the searchString property which is also of type string.

The `GetCityList` method will make a call to our web API GetCities method to fetch the list of city data from the cities collection. The `GetEmployeeList` method will send a GET request to our web API to fetch the list of Employee Data from the Employee table.

We are invoking these two methods inside the `OnInitAsync` method, to ensure that the Employee Data and the cities data will be available as the page loads.

The `AddEmployee` method will initialize an empty instance of the Employee object and set the modalTitle property, which will display the title message on the Add modal popup.

The `EditEmployee` method will accept the employee ID as the parameter. It will send a GET request to our web API to fetch the record of the employee corresponding to the employee ID supplied to it.

We will use the `SaveEmployee` method to save the record of the employee for both the Add request and Edit request. To differentiate between the Add and the Edit requests, we will use the EmployeeId property of the Employee object. If an Edit request is made, then the EmployeeId property contains a string value, and we will send a PUT request to our web API, which will update the record of the employee. Otherwise, if we make an Add request, then the EmployeeId property is not initialized, and hence it will be null. In this case, we need to send a POST request to our web API, which will create a new employee record.

The `DeleteConfirm` method will accept the employee ID as the parameter. It will fetch the Employee Data corresponding to the employee ID supplied to it.

The `DeleteEmployee` method will send a delete request to our API and pass the employee ID as the parameter. It will then call the GetEmployeeList method to refresh the view with the updated list of Employee Data.

The `SearchEmployee` method is used to implement the search by the employee name functionality. We will return all the records of the employee, which will match the search criteria either fully or partially. To make the search more effective, we will ignore the text case of the search string. This means the search result will be same whether the search text is in uppercase or in lowercase.

EmployeeData.cshtml

Open `EmployeeData.cshtml` page and put the following code into it.

```
@page "/employeeerecords"
@inherits EmployeeDataModel
<h1>Employee Data</h1>
<div class="container">
    <div class="row">
        <div class="col-xs-3">
            <button class="btn btn-primary" data-toggle="modal" data-target="#AddEditEmpModal"
onclick="@AddEmployee">
                <i class="fa fa-user-plus"></i>
                Add Employee
            </button>
        </div>
        <div class="input-group col-md-4 offset-md-5">
            <input type="text" class="form-control" placeholder="Search Employee"
bind="@searchString" />
            <div class="input-group-append">
                <button class="btn btn-info" onclick="@SearchEmployee">
                    <i class="fa fa-search"></i>
                </button>
            </div>
        </div>
    </div>
</div>
<br />
@if (empList == null)
{
    <p><em>Loading...</em></p>
}
else
```

```

{
    <table class='table'>
        <thead>
            <tr>
                <th>Name</th>
                <th>Gender</th>
                <th>Designation</th>
                <th>City</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var emp in empList)
            {
                <tr>
                    <td>@emp.EmployeeName</td>
                    <td>@emp.Gender</td>
                    <td>@emp.Designation</td>
                    <td>@emp.CityName</td>
                    <td>
                        <button class="btn btn-outline-dark" data-toggle="modal" data-
target="#AddEditEmpModal"
                        onclick="@ (async () => await EditEmployee (@emp.EmployeeId)) ">
                            <i class="fa fa-pencil-square-o"></i>
                            Edit
                        </button>
                        <button class="btn btn-outline-danger" data-toggle="modal" data-
target="#deleteEmpModal"
                        onclick="@ (async () => await DeleteConfirm (@emp.EmployeeId)) ">
                            <i class="fa fa-trash-o"></i>
                            Delete
                        </button>
                    </td>
                </tr>
            }
        </tbody>
    </table>
}

<div class="modal fade" id="AddEditEmpModal">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <h3 class="modal-title">@modalTitle</h3>
                <button type="button" class="close" data-dismiss="modal">
                    <span aria-hidden="true">X</span>
                </button>
            </div>
            <div class="modal-body">
                <form>
                    <div class="form-group">
                        <label class="control-label">Name</label>
                        <input class="form-control" bind="@emp.EmployeeName" />
                    </div>
                    <div class="form-group">
                        <label class="control-label">Gender</label>
                        <select class="form-control" bind="@emp.Gender">
                            <option value="">-- Select Gender --</option>
                            <option value="Male">Male</option>
                            <option value="Female">Female</option>
                        </select>
                    </div>
                    <div class="form-group">
                        <label class="control-label">Designation</label>
                        <input class="form-control" bind="@emp.Designation" />
                    </div>
                    <div class="form-group">

```



```

<label class="control-label">City</label>
<select class="form-control" bind="@emp.CityName">
  <option value="-- Select City --">-- Select City --</option>
  @foreach (var city in cityList)
  {
    <option value="@city.CityName">@city.CityName</option>
  }
</select>
</div>
</form>
</div>
<div class="modal-footer">
  <button class="btn btn-block btn-success"
    onclick="@ (async () => await SaveEmployee())" data-dismiss="modal">
    Save
  </button>
</div>
</div>
</div>
<div class="modal fade" id="deleteEmpModal">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h3 class="modal-title">Confirm Delete !!!</h3>
        <button type="button" class="close" data-dismiss="modal">
          <span aria-hidden="true">X</span>
        </button>
      </div>
      <div class="modal-body">
        <table class="table">
          <tr>
            <td>Name</td>
            <td>@emp.EmployeeName</td>
          </tr>
          <tr>
            <td>Gender</td>
            <td>@emp.Gender</td>
          </tr>
          <tr>
            <td>Designation</td>
            <td>@emp.Designation</td>
          </tr>
          <tr>
            <td>City</td>
            <td>@emp.CityName</td>
          </tr>
        </table>
      </div>
      <div class="modal-footer">
        <button class="btn btn-danger" data-dismiss="modal"
          onclick="@ (async () => await DeleteEmployee(@emp.EmployeeId))">
          Delete
        </button>
        <button data-dismiss="modal" class="btn">Cancel</button>
      </div>
    </div>
  </div>
</div>
</div>

```

The route for our component is defined at the top as “/employeeerecords”. To use the methods defined in the EmployeeDataModel class, we will inherit it using the `@inherits` directive.

We have defined an Add Employee button. Upon clicking, this button will invoke the `AddEmployee` method and open a modal dialog, which allows the user to fill out the new Employee Data in a form.

We have also defined our search box and a corresponding search button. The search box will bind the value to searchString property. On clicking the search button, `SearchEmployee` method will be invoked, which will return the filtered list of data as per the search text. If the empList property is not null, we will bind the Employee Data to a table to display it on the web page. Each employee record has the following two action buttons corresponding to it:

- Edit: This button will perform two tasks. It will invoke the EditEmployee method and open the edit employee modal dialog for editing the employee record.
- Delete: This button will also perform two tasks. It will invoke the DeleteConfirm method and open a delete confirm modal dialog, asking the user to confirm the deletion of the employee's record.

We have defined a form inside the bootstrap modal to accept user inputs for the employee records. The input fields of this form will bind to the properties of the employee class. The City field is a drop-down list, which will bind to the cities collection of the database with the help of the cityList variable. When we click on the save button, the `SaveEmployee` method will be invoked and the modal dialog will be closed.

When user click on the Delete button corresponding to an employee record, another bootstrap modal dialog will be displayed. This modal will show the Employee Data in a table and ask the user to confirm the deletion. Clicking on the Delete button inside this modal dialog will invoke the DeleteEmployee method and close the modal. Clicking on the Cancel button will close the modal without performing any action on the data.

Adding the navigation link to our component

Before executing the application, we will add the navigation link to our component in the navigation menu.

Open the `BlazorWithFirestore.Client/Shared/NavMenu.cshtml` page and add the following navigation link:

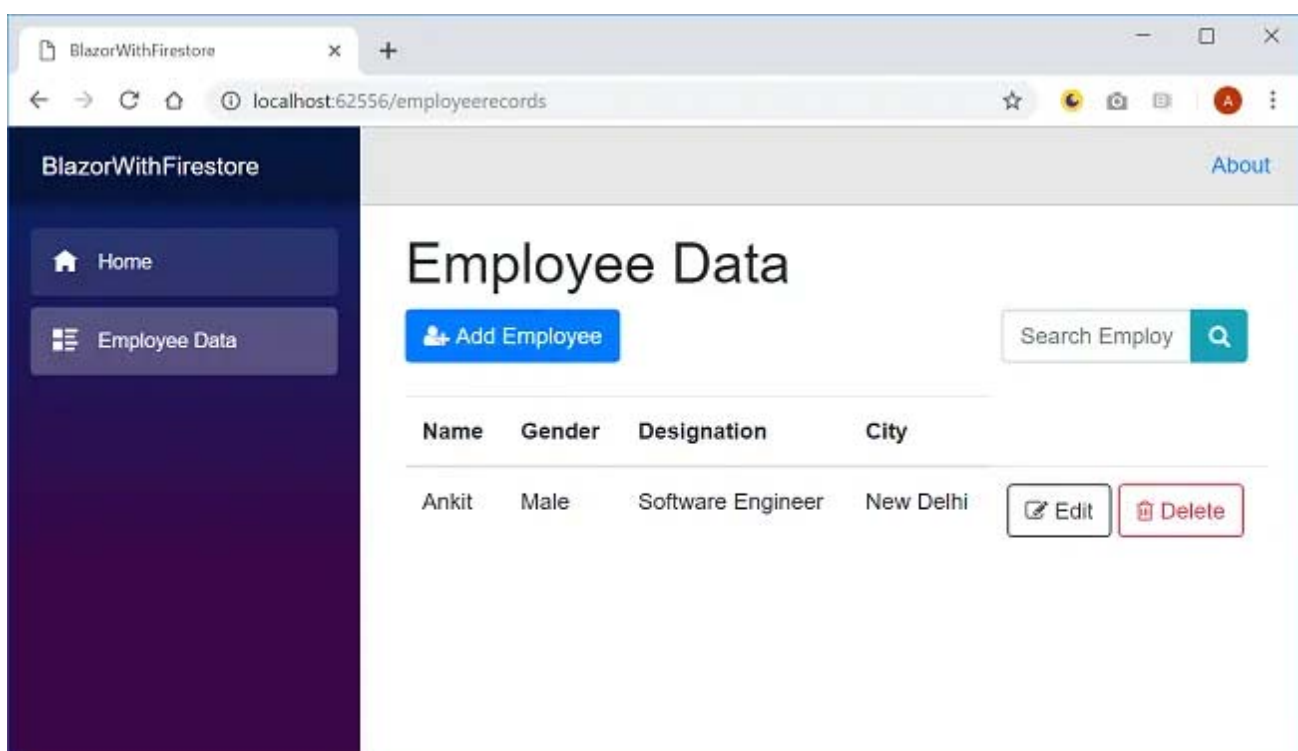
```
<li class="nav-item px-3">
  <NavLink class="nav-link" href="employeerecords">
    <span class="oi oi-list-rich" aria-hidden="true"></span> Employee Data
  </NavLink>
</li>
```

Hence, we have successfully created a Single Page Application (SPA) using Blazor with the help of cloud Firestore as database provider.

Execution Demo

Press F5 to launch the application.

A web page will open as shown in the image below. The navigation menu on the left is showing navigation link for Employee data page.



You can perform the CRUD operations on this application as shown in the GIF image at the start of this article.

Conclusion

We have created a Single Page Application (SPA) using Blazor with the help of Google cloud Firestore as database provider. We have created a sample employee record management system and perform CRUD operations on it. Firestore is a NoSQL database, which allows us to store data in form of collections and documents. We have also used a bootstrap modal popup to handle user inputs. We have also implemented a search box to search the employee list by employee name.

Please get the source code from [GitHub](#) and play around to get a better understanding.

Get my book [Blazor Quick Start Guide](#) to learn more about Blazor.

Preparing for interviews !!! Read my article on [C# Coding Questions For Technical Interviews](#)

See Also

- [BlazorGrid – A Reusable Grid Component For Blazor](#)
- [Publishing A Blazor Component To Nuget Gallery](#)
- [Deploying a Blazor Application on IIS](#)
- [Deploying A Blazor Application On Azure](#)
- [Hosting A Blazor Application on Firebase](#)
- [CRUD Using Blazor with MongoDB](#)
- [Single Page Application Using Server-Side Blazor](#)