# How to Implement Blazor CRUD using Entity Framework Core? Detailed Demonstration

By Mukesh Murugan | Last Updated On April 15, 2021

BLAZOR

**Building a CRUD Application is like the Hello World for Intermediate Developers.** It helps you understand the most common operations of any particular stack. In this tutorial, let's build a Client-side Blazor CRUD Application that uses Entity Framework Core as it's Data Access Layer. In our previous articles, we discussed Blazor basics and it's folder structures.

Blazor is the new popular kid in town. It's quite important for a .NET Developer to be aware of what this Awesome Tech packs within itself. What more would make you understand it's abilities other than learning how to implement CRUD Functions using Blazor and Entity Framework Core?

This is part 3 of my Blazor Blog Series, where we learn together various concepts and implementations of Microsoft's latest tech, Blazor. Here are the contents of my Blazor Blog Series.

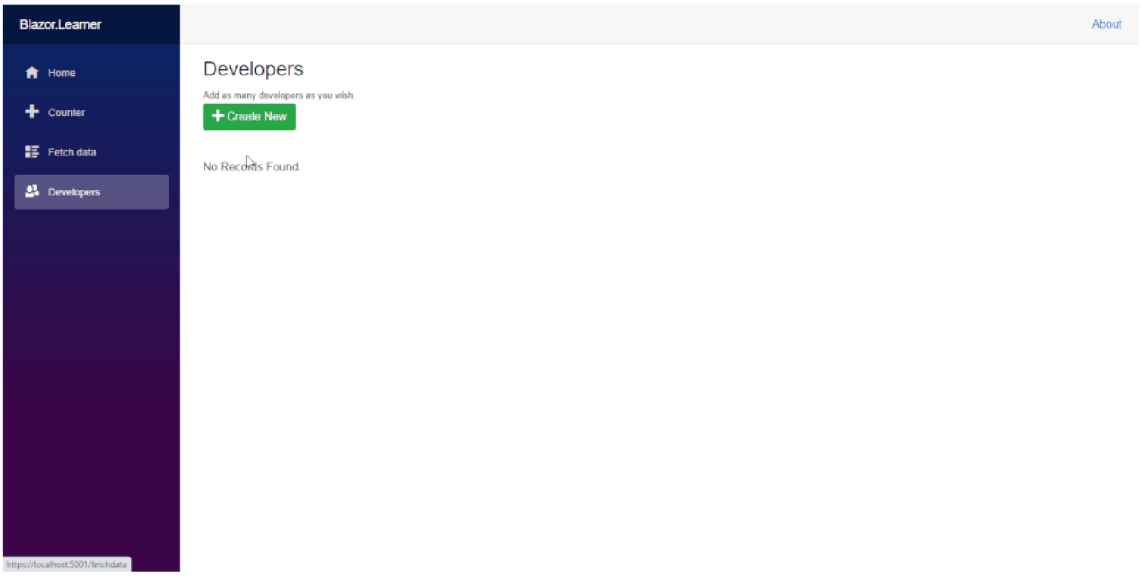
1. Getting Started with Blazor
2. Exploring Blazor Project Structure
3. **Blazor CRUD with Entity Framework Core – (You are here)**
4. Implementing Blazor CRUD using Mudblazor Component Library in .NET 5 – Detailed Guide

> PS, I recommend using Visual Studio 2019 Community as my IDE. Also, To get Blazor onto your machine, please go through Part 1 of the Blazor Blog Series where the pre-requisites are mentioned.

Table of Contents

# What we will be Building

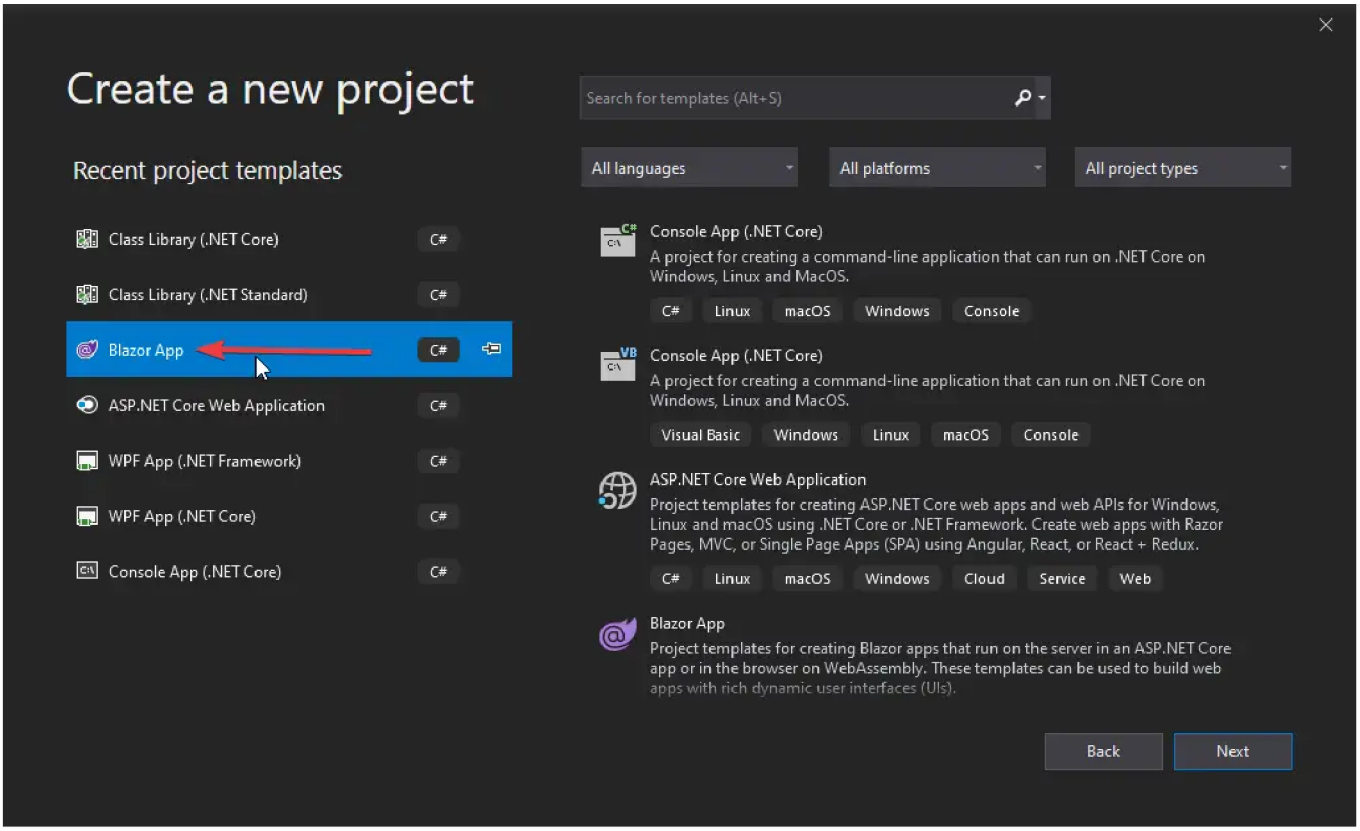
I thought it would be nice if I showed you guys what we are going to build before jumping into the tutorial so that you get a good idea of the scope of this tutorial and we will be covering. We will have a simple Blazor CRUD implementation on the Developer entity using Blazor WebAssembly and Entity Framework Core. I will also try to demonstrate certain best practices while developing Blazor CRUD Applications.

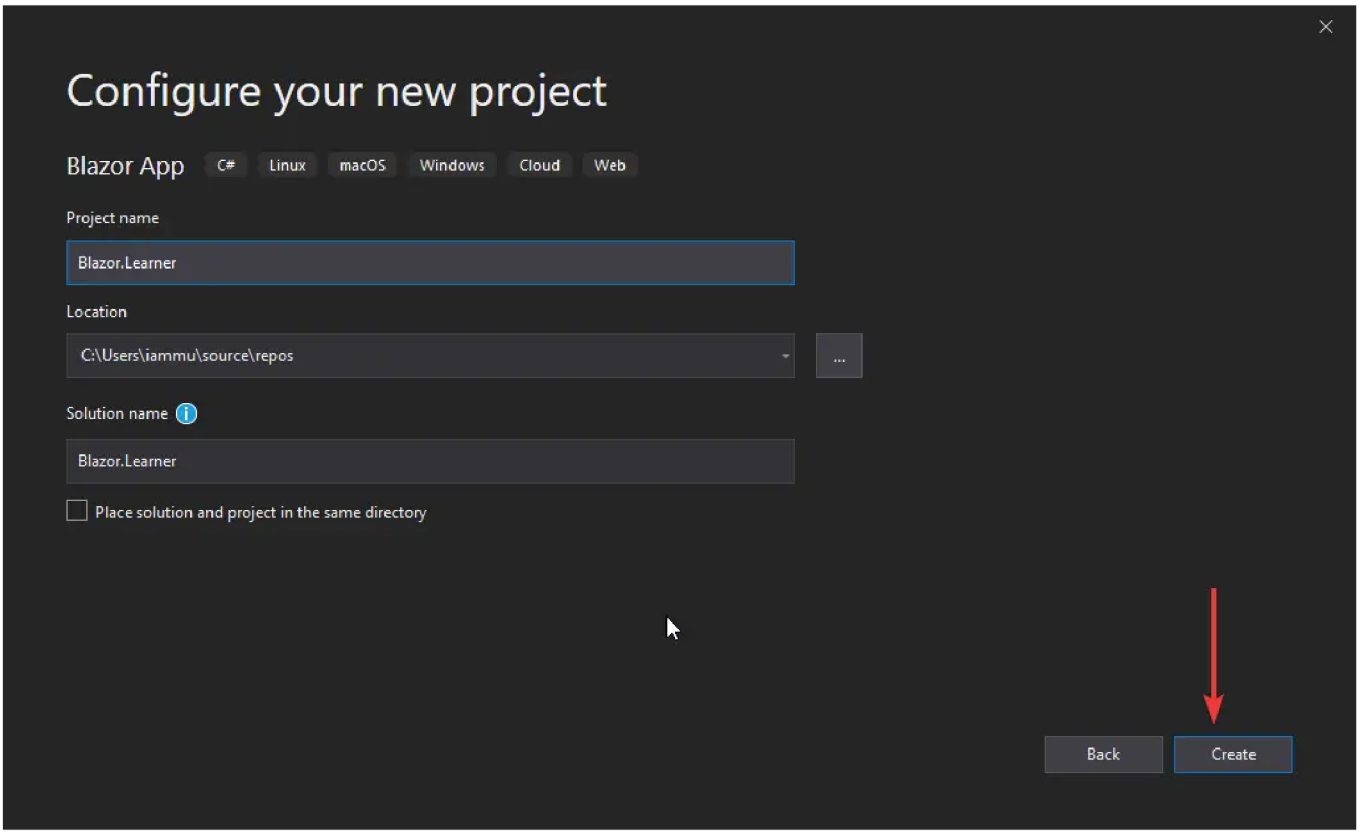


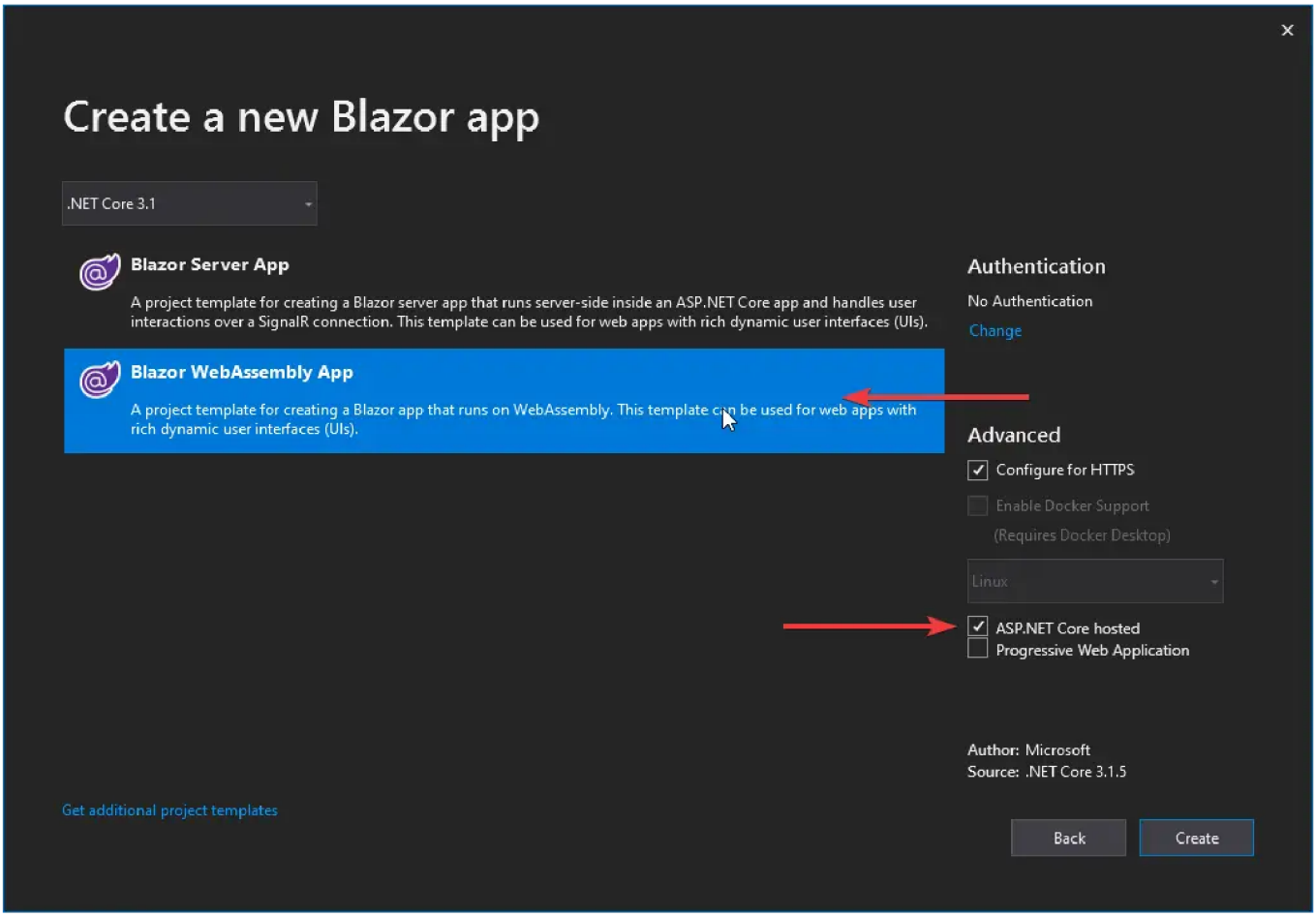

> Note: Let's build a complete Blazor Application step by step. **I will upload the source code of the Application (Blazor.Learner) over at GitHub.** We will be reusing and building on top of this application in our future articles as well. **Make sure you guys follow me at GitHub to stay posted.**

## Creating the Blazor CRUD Project

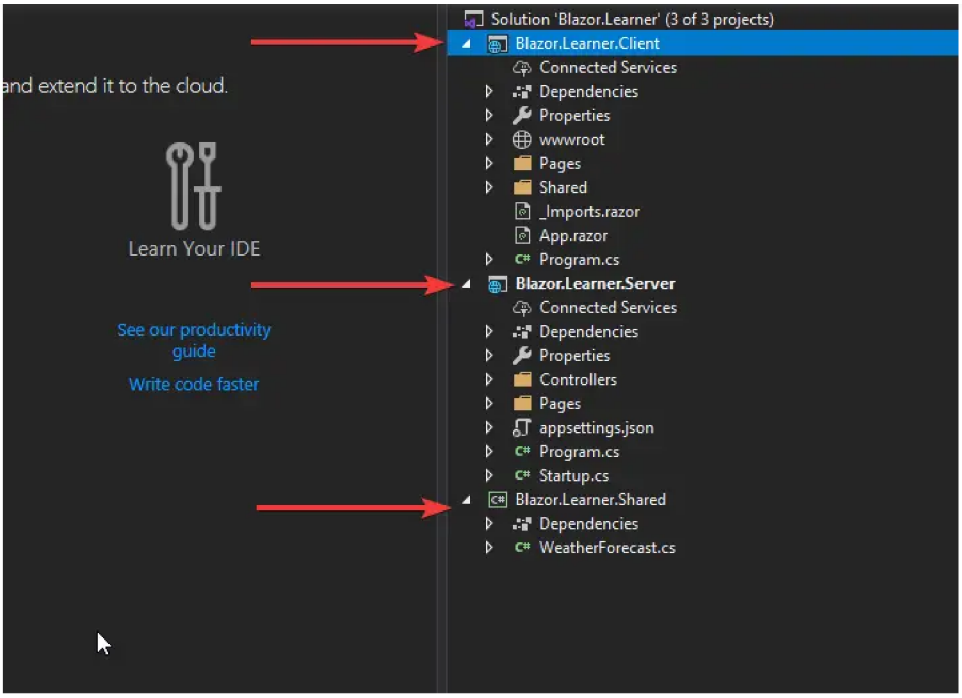
Let's get started with the Blazor WebAssembly CRUD application by creating a new Blazor App. Follow the screenshots below.






**Make sure to check the ASP.NET Core Hosted option.** So, we are building a client-side aka Blazor WebAssembly project. But there can be instances where we need the features of an ASP.NET Core Application to support the Blazor Application, like using external APIs for data, Application-specific database, etc. For such requirements, we use the ASP.NET Core Hosted hosting model.




You can see that Visual Studio automatically generates 3 different projects, i.e, Client, Server, and Shared.

1. Client – Where the Blazor Application lives along with the Razor components.
2. Server – Mostly used as a container that has ASP.NET Core Features (We use it here for EF Core, Api Controllers, and DB).
3. Shared – As the name suggests, all the entity models will be defined here.

**So basically, the Server will have API endpoints that can be accessed by the Client Project. Please note that this is a single application and runs on the same port. Hence the need for CORS access doesn't arise. All these projects will be executed directly on your browser and do not need a dedicated server.**

## Adding the Model

To begin with our Blazor CRUD Application, let's add a Developer Model on to our Shared Project under the Models folder (create one).

```
1.    public class Developer
2.       {
3.           public int Id { get; set; }
4.           public string FirstName { get; set; }
5.           public string LastName { get; set; }
6.           public string Email { get; set; }
7.           public decimal Experience { get; set; }
```

# Entity Framework Core

Now, go to the server project and install the following required packages to enable EF Core.

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.Design
Install-Package Microsoft.EntityFrameworkCore.Tools
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

# Defining Connection String

Navigate to appsettings.json under the Server Project and add the Connection String.

```
"ConnectionStrings": {
    "DefaultConnection": "<Connection String Here>"
  },
```

# Adding Application Context

We will need a database context to work on the data. Create a new class in the Server Project at **Data/ApplicationDBContext.cs**

```
public class ApplicationDBContext : DbContext
    {
        public ApplicationDBContext(DbContextOptions<ApplicationDBContext> options):base(options)
        {
        }
        public DbSet<Developer> Developers { get; set; }
    }
```

We will be adding DbSet of Developers to our context. Using this context class, we will be able to perform operations in our database that we will generate in some time.

# Configuring

We will need to add EF Core and define it's connection string. Navigate to the Startup.cs found in the server project and add the following line to the ConfigureServices method.

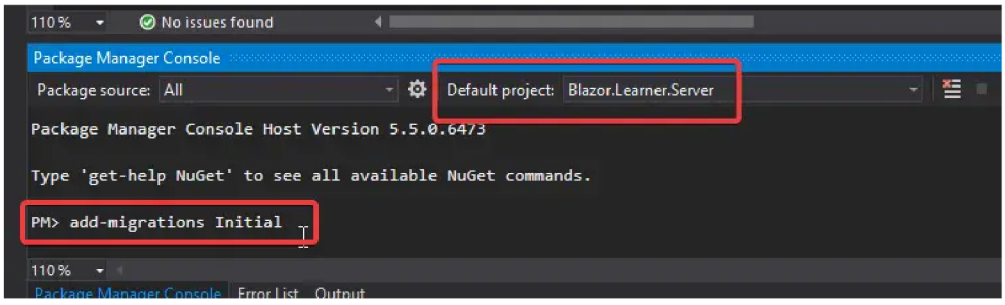
```
services.AddDbContext<ApplicationDBContext>(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnecti
```

## Generating / Migrating the Database

Now that our Entity Framework Core is all set and up on the ASP.NET Core Application, let 's do the migrations and update our database. For this, open up the Package Manager Console and type in the following.

```
add-migration Initial
update-database
```

**Important – Make sure you have chosen the server project as the default**.




## Developer Controller.

Now that our database and EF Core is set up, we will build an API Controller that will serve the data to our Blazor client. Create an Empty API Controller, **Controllers/DeveloperController.cs**

```
    public class DeveloperController : ControllerBase
    {
        private readonly ApplicationDBContext _context;

        public DeveloperController(ApplicationDBContext context)
        {
            this._context = context;
        }
    }
```

Here we have injected a new instance of ApplicationDBContent to the constructor of the controller. Let's continue by adding each of the endpoints for our CRUD Operations.

## Get

An Action method to get all the developers from the context instance.

```
[HttpGet]
        public async Task<IActionResult> Get()
        {
            var devs = await _context.Developers.ToListAsync();
            return Ok(devs);
        }
```

## Get By Id

Fetches the details of one developer that matches the passed id as parameter.

```
 [HttpGet("{id}")]
        public async Task<IActionResult> Get(int id)
        {
            var dev = await _context.Developers.FirstOrDefaultAsync(a=>a.Id ==id);
            return Ok(dev);
        }
```

## Create

Creates a new Developer with the passed developer object data.

```
        public async Task<IActionResult> Post(Developer developer)
        {
            _context.Add(developer);
            await _context.SaveChangesAsync();
            return Ok(developer.Id);
        }
```

## Update

Modifies an existing developer record.

```
[HttpPut]
        public async Task<IActionResult> Put(Developer developer)
        {
            _context.Entry(developer).State = EntityState.Modified;
            await _context.SaveChangesAsync();
            return NoContent();
        }
```

## Delete

Deletes a developer record by Id.

```
[HttpDelete("{id}")]
        public async Task<IActionResult> Delete(int id)
        {
            var dev = new Developer { Id = id };
            _context.Remove(dev);
            await _context.SaveChangesAsync();
            return NoContent();
        }
```

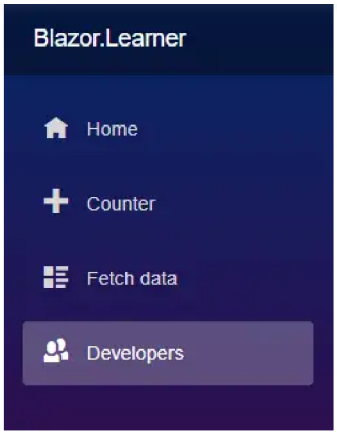
# Getting Started with Blazor CRUD

With all that out of the way, let's continue building our Blazor CRUD Application. Our Agenda is to do CRUD Operations on the Developer Entity. Essentially we have completed our data layer. Let's now build the UI.

# Adding a new Navigation Menu Entry

We will have to add a new entry in the navigation menu sidebar to access the Pages. On the client project, Navigate to **Shared/NavMenu.razor** and add a similar entry to the list.

```
 <li class="nav-item px-3">
            <NavLink class="nav-link" href="developer">
                <span class="oi oi-people" aria-hidden="true"></span> Developers
            </NavLink>
        </li>
```




## Shared Namespaces

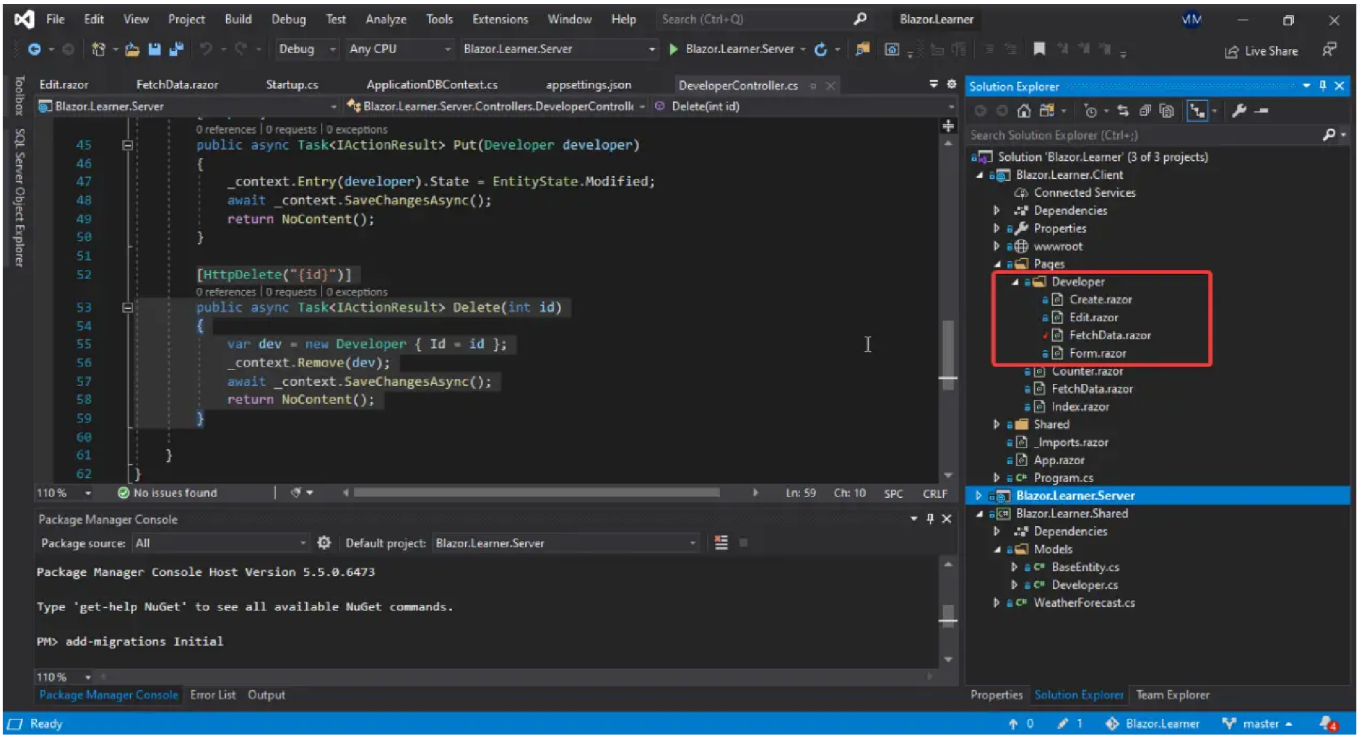
We will be using the Shared/Models/Developer.cs Model class throughout this tutorial. Let's add it's the namespace to the _Imports.razor, so that it can be accessed at all our new components.

```
1.  @using Blazor.Learner.Shared.Models
```

## Proposed Folder Structure

This is considered as one of the better practices. The idea goes like this.

1. Have a folder under pages that will be specific for an entity/feature. In our case, we have a Developer as an entity.
2. Under this folder, we try to include all the concerned Razor components.
3. If you are already an ASP.NET Core Developer, you would know that while implementing standard CRUD, we have many similar things (UI) for Create and Update Forms. For this, we create a common component called Form that will be shared by the Create and Edit components.
4. FetchData is to retrieve all the data and display it onto a Bootstrap table.




# FetchData Component

We will start off by building our Index Component that fetches all the developers from the database. Let's call it the FetchData Component. Create a new Blazor Component under Pages, **Pages/Developer/FetchData.razor**

```
@page "/developer"
@inject HttpClient client
@inject IJSRuntime js

<h3>Developers</h3>
<small>Add as many developers as you wish.</small>
<div class="form-group">
    <a class="btn btn-success" href="developer/create"><i class="oi oi-plus"></i> Create New</a>
</div>
<br>

@if (developers == null)
{
    <text>Loading...</text>
}
else if (developers.Length == 0)
{
    <text>No Records Found.</text>
}
```

```
23.          <thead>
24.              <tr>
25.                  <th>Id</th>
26.                  <th>First Name</th>
27.                  <th>Last Name</th>
28.                  <th>Email</th>
29.                  <th>Experience (Years)</th>
30.                  <th></th>
31.              </tr>
32.          </thead>
33.          <tbody>
34.              @foreach (Developer dev in developers)
35.              {
36.                  <tr>
37.                      <td>@dev.Id</td>
38.                      <td>@dev.FirstName</td>
39.                      <td>@dev.LastName</td>
40.                      <td>@dev.Email</td>
41.                      <td>@dev.Experience</td>
42.                      <td>
43.                          <a class="btn btn-success" href="developer/edit/@dev.Id">Edit</a>
44.                          <button class="btn btn-danger" @onclick="@(() => Delete(dev.Id))">Delete</button>
45.                      </td>
46.                  </tr>
47.              }
48.
49.          </tbody>
50.      </table>
51.  }
52.  @code {
53.      Developer[] developers { get; set; }
54.      protected override async Task OnInitializedAsync()
55.      {
56.          developers = await client.GetFromJsonAsync<Developer[]>("api/developer");
57.      }
58.
59.      async Task Delete(int developerId)
60.      {
61.          var dev = developers.First(x => x.Id == developerId);
62.          if (await js.InvokeAsync<bool>("confirm", $"Do you want to delete {dev.FirstName}'s ({dev.Id}) Record?"))
63.          {
64.              await client.DeleteAsync($"api/developer/{developerId}");
65.              await OnInitializedAsync();
66.          }
67.      }
68.  }
```

Line 8 – Button to Create a new Developer

Line 34 – Iteration for a new Row for each developer record.

Line 43 – An Edit Button for every record that navigates to …developer/edit/{id}

Line 44 – Delete Button for the Record. This Button invokes a Delete Method written in the Code area of this component.

Line 53 – Defining a list of Developers.

Line 54-55 – This is the function that gets fired on page load. Here we use the HTTP Client object to retrieve data from the API endpoint, api/developer.

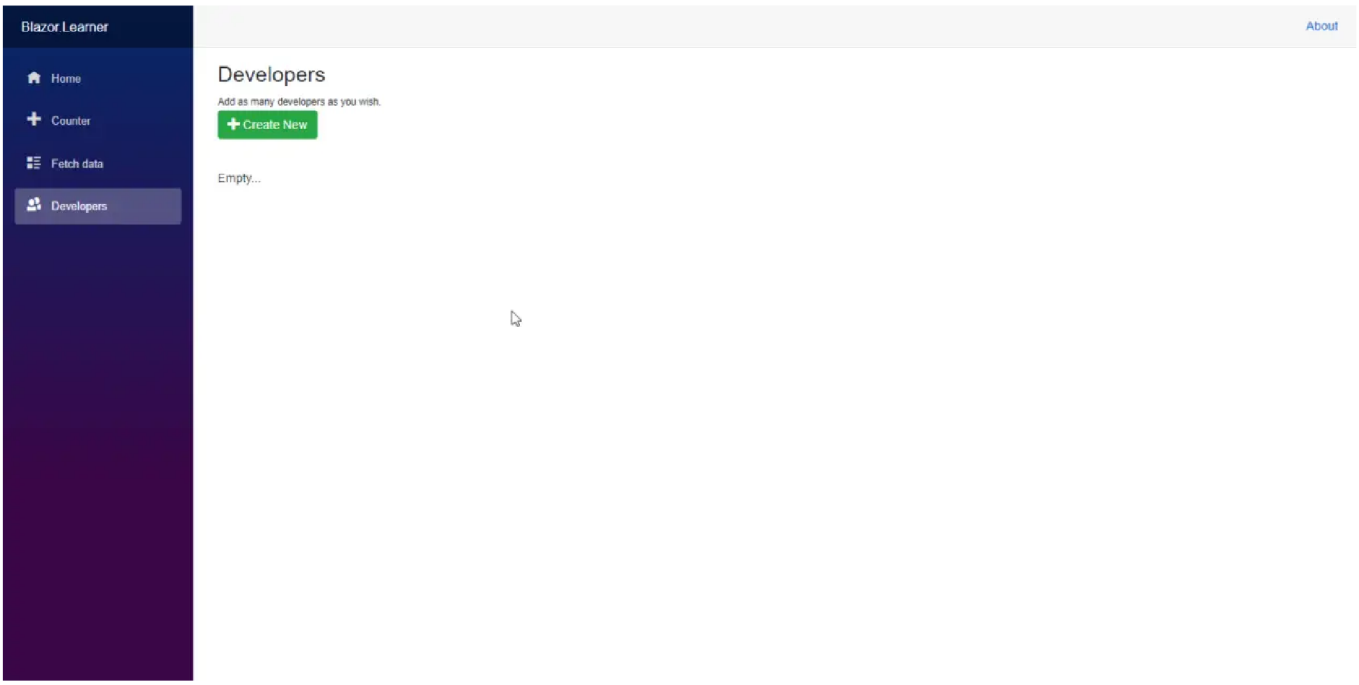
Line 59-67 – A standard delete function that gets invoked on a button click.

Line 61- An Object of IJSRuntime that invoked the confirmation dialog.

## What is IJSRuntime?

Previously, we had learned that it is also possible to execute Javascripts in our Blazor Application. The IJSRuntime interface acts as a gateway to JS. This object gives us methods to invoke JS functions in Blazor applications.

Let's build our application and run it.




Your first CRUD Component ready in no time! 😀 You can see that we don't have any record. Let's add a Create Razor Component to be able to add new entries to the database.

But before that, we will have to build a common Form that will be shared by Create and Edit Componets.

# Form Component

Add a new Razor component at **Pages/Developer/Form.razor**

```
<EditForm Model="@dev" OnValidSubmit="@OnValidSubmit">
    <DataAnnotationsValidator />
    <div class="form-group">
        <label>First Name :</label>
        <div>
            <InputText @bind-Value="@dev.FirstName" />
            <ValidationMessage For="@(() => dev.FirstName)" />
        </div>
    </div>
    <div class="form-group ">
        <div>
            <label>Last Name :</label>
            <div>
                <InputText @bind-Value="@dev.LastName" />
                <ValidationMessage For="@(() => dev.LastName)" />
            </div>
        </div>
    </div>
    <div class="form-group ">
        <div>
            <label>Email :</label>
            <div>
                <InputText @bind-Value="@dev.Email" />
                <ValidationMessage For="@(() => dev.Email)" />
            </div>
        </div>
    </div>
    <div class="form-group ">
        <div>
            <label>Experience :</label>
            <div>
                <InputNumber @bind-Value="@dev.Experience" />
                <ValidationMessage For="@(() => dev.Experience)" />
            </div>
        </div>
    </div>

    <button type="submit" class="btn btn-success">
        @ButtonText
    </button>

</EditForm>


@code {
    [Parameter] public Developer dev { get; set; }
    [Parameter] public string ButtonText { get; set; } = "Save";
    [Parameter] public EventCallback OnValidSubmit { get; set; }
}
```

Line 1 – EditForm tag that takes in a Developer Model and has a function call to submit.

Line 2 – Validation.

Line 3-9 – A Text Box for the First Name of the developer. Notice that we have bound it to the FirstName property of the model.
Line 28-36 – An Input for numbers.

Line 46 – Developer Object.
Line 47 – Button Text Caption Defaults.
Line 48 – Here is the Method to be called on submitting the form.

> Thinking about this, I feel this is quite similar to the concepts of Interface and Concrete classes. Understand it this way. Form Component is the interface that has a blueprint of the properties and methods needed. And the Create/Edit component would be the Concrete class which has the actual implementation of the interface properties/methods. Doesn't make sense? Read this paragraph again after going through the Create Component Section.

## What is the Parameter tag for?

In Blazor, you can add parameters to any components by decorating them with a Parameter tag. What it does is, it becomes available for external components to pass in these parameters. In our case, we have defined the Developer object as the parameters of the Forms components. So, the other components that will use the Form Components, ie, the Create / Edit Components have an option to pass in the Developer object as a parameter to the Form Components. This helps high re-usability of components.

# Create Component

Now, let's start using the previously created Forms component. We will build an interdace for adding a new developer. Create a new Razor Component, **Pages/Developer/Create.razor**

```
1.  @page "/developer/create"
2.  @inject HttpClient http
3.  @inject NavigationManager uriHelper
```

```
7.  <Form ButtonText="Create Developer" dev="@dev"
8.              OnValidSubmit="@CreateDeveloper" />
9.
10. @code {
11.     Developer dev = new Developer();
12.     async Task CreateDeveloper()
13.     {
14.         await http.PostAsJsonAsync("api/developer", dev);
15.         uriHelper.NavigateTo("developer");
16.     }
17. }
```

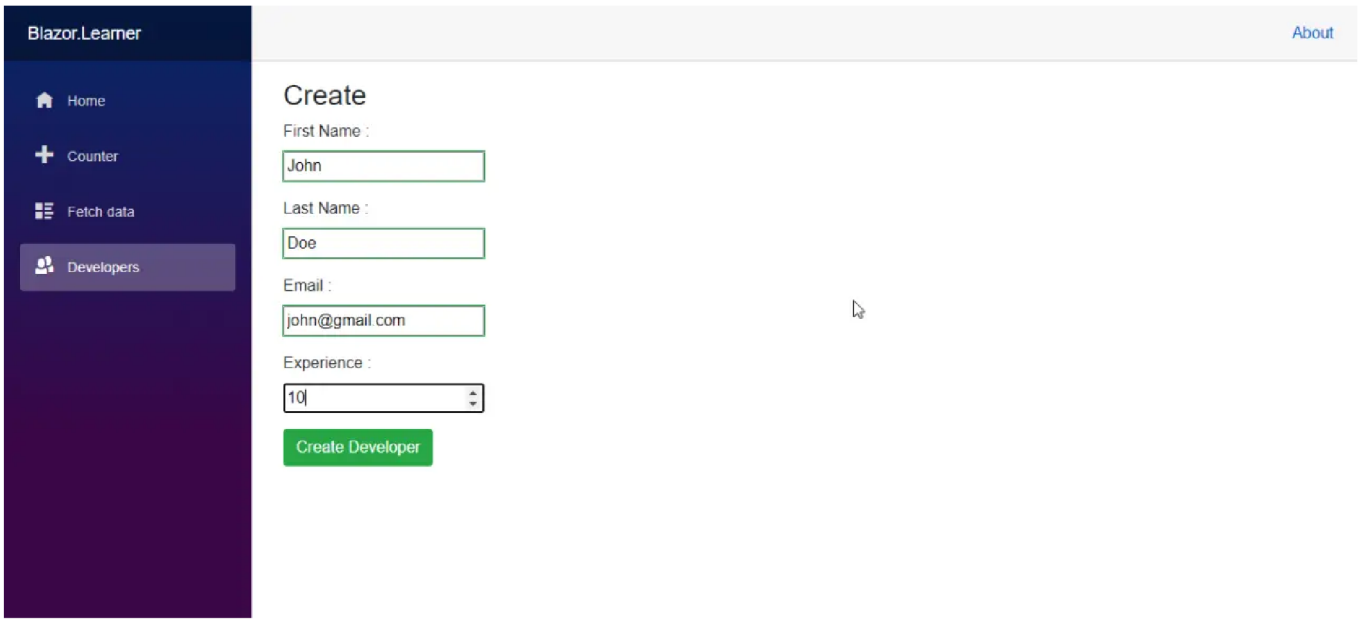
Line 1 – Component Route – ../developer/create

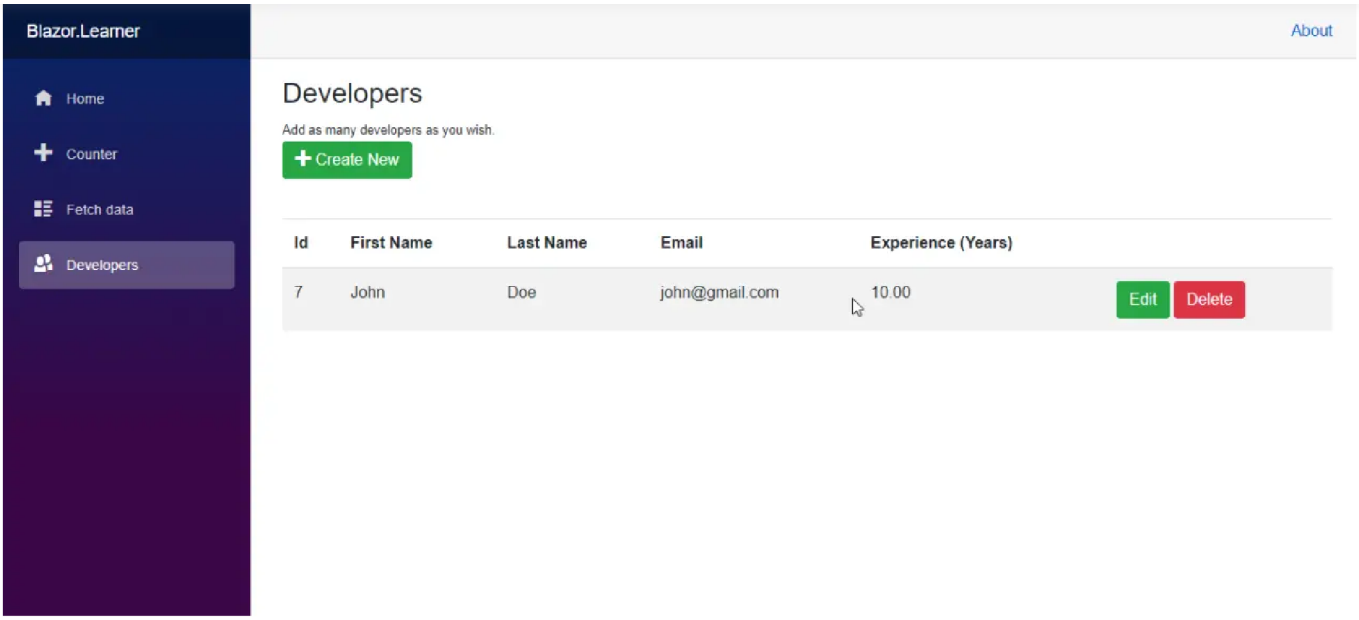
Line 7 – Here we are using the Form Component that we created earlier. To this component tag, we are passing parameters like Button Text, a new Blank Developer Object, and a method that is to be called when a user hits the Create Button of this component,

Line 14 – Post the Data to the DB.

Line 15 – Once Inserted, Navigate back to the FetchData component.

Let's run the application now. Navigate to the Developers Tab and click on the Create Button. Here add in sample data and hit Create Developer.






You will be amazed by the speed of the application. Quite smooth yeah? Also, we have built our Create Component quite well by now. Now let's build our final component. Update.

# Edit Component

Create a new Component under **Pages/Developer/Edit.razor**

```
@page "/developer/edit/{developerId:int}"
@inject HttpClient http
@inject NavigationManager uriHelper
@inject IJSRuntime js

<h3>Edit</h3>

<Form ButtonText="Update" dev="dev"
      OnValidSubmit="@EditDeveloper" />

@code {
    [Parameter] public int developerId { get; set; }
    Developer dev = new Developer();

    protected async override Task OnParametersSetAsync()
    {
        dev = await http.GetFromJsonAsync<Developer>($"api/developer/{developerId}");
    }

    async Task EditDeveloper()
    {
        await http.PutAsJsonAsync("api/developer", dev);
        await js.InvokeVoidAsync("alert", $"Updated Successfully!");
        uriHelper.NavigateTo("developer");

    }
}
```

Line 1 – Component Route – ../developer/edit/{id}

Line 7 – Similar to the precious Create Component, we will use the Forms Component

Line 17 – On getting the parameter from the URL, ie, the developer Id, we retrieve that particular record from our API endpoint. This data will be filled on the form by default.

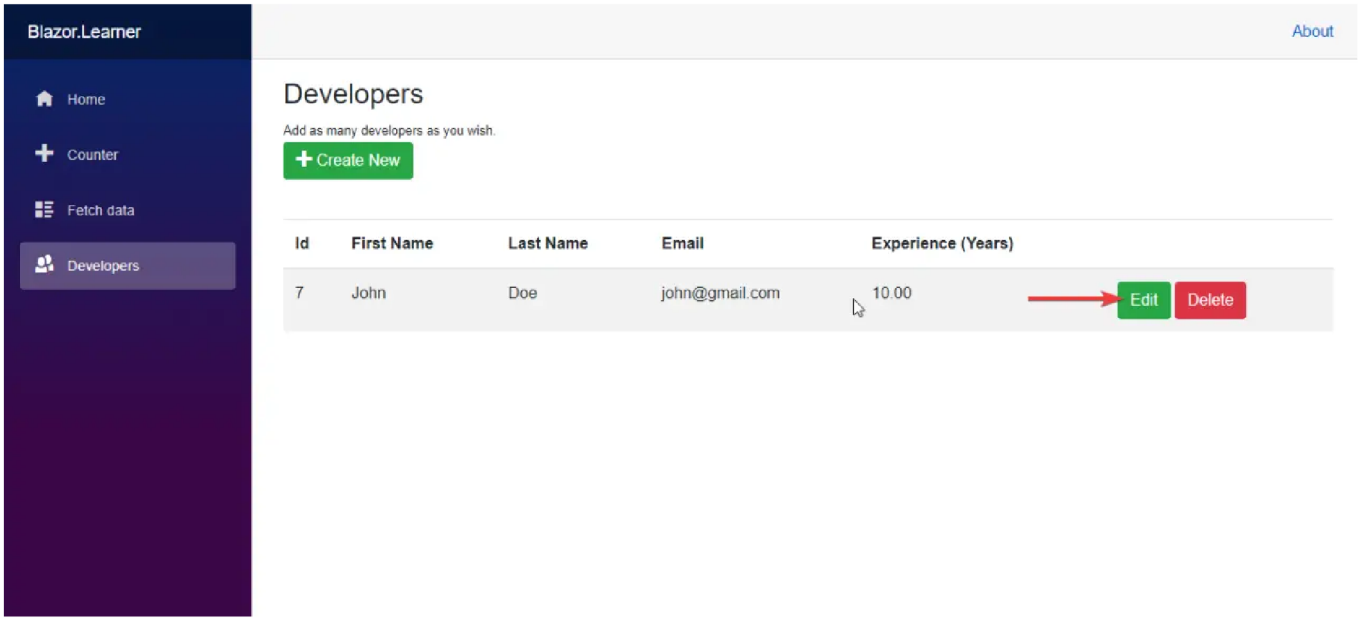
Line 20- 26 – The method that fires up when the user clicks on the Update Button.

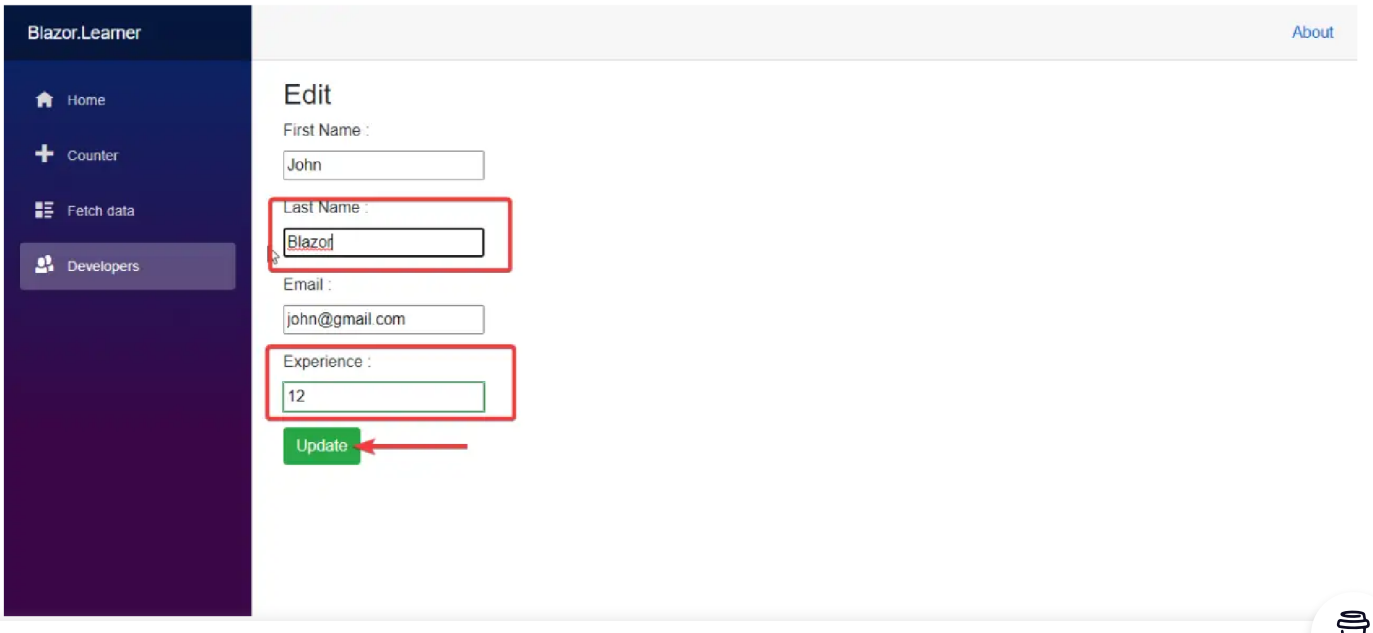
Line 22 – Post the Edited data to the API.

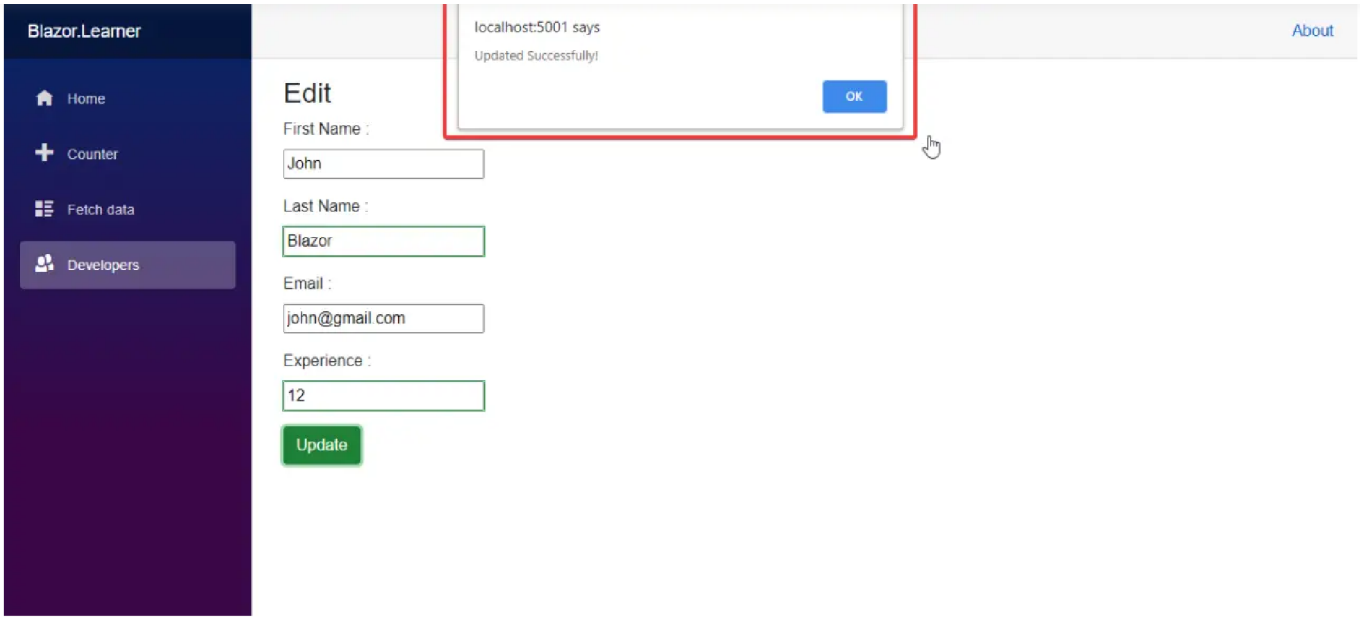
Line 23 – Displays an alert message showing "Updated Successfully!"

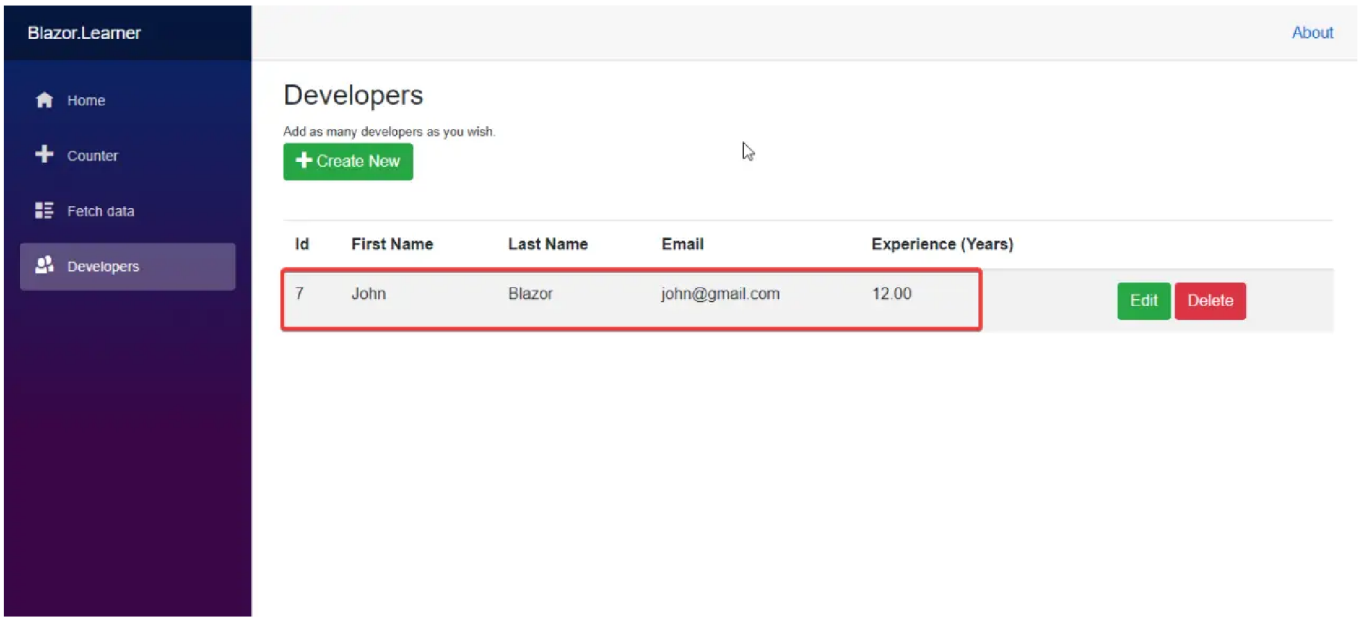
Line 24 – Redirects to the FetchData component.

Let's run the application. Navigate to Developers. and follow the below steps.



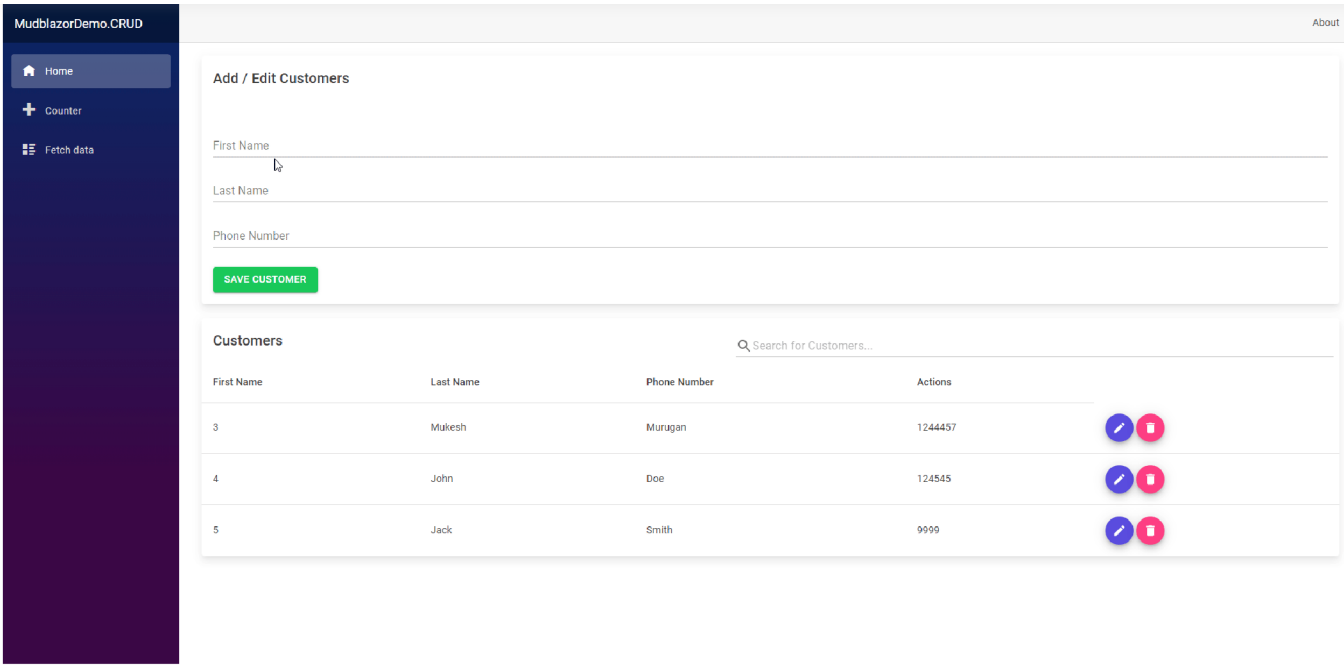






## Bonus Resource

> If you liked this article, I am sure that you would love this too – Implementing Blazor CRUD using Mudblazor Component Library in .NET 5 – Detailed Guide. Here I talk about implementing the same CRUD operations, but with one of the coolest component libraries for Blazor. Here is a small preview of what you can build with MudBlazor.

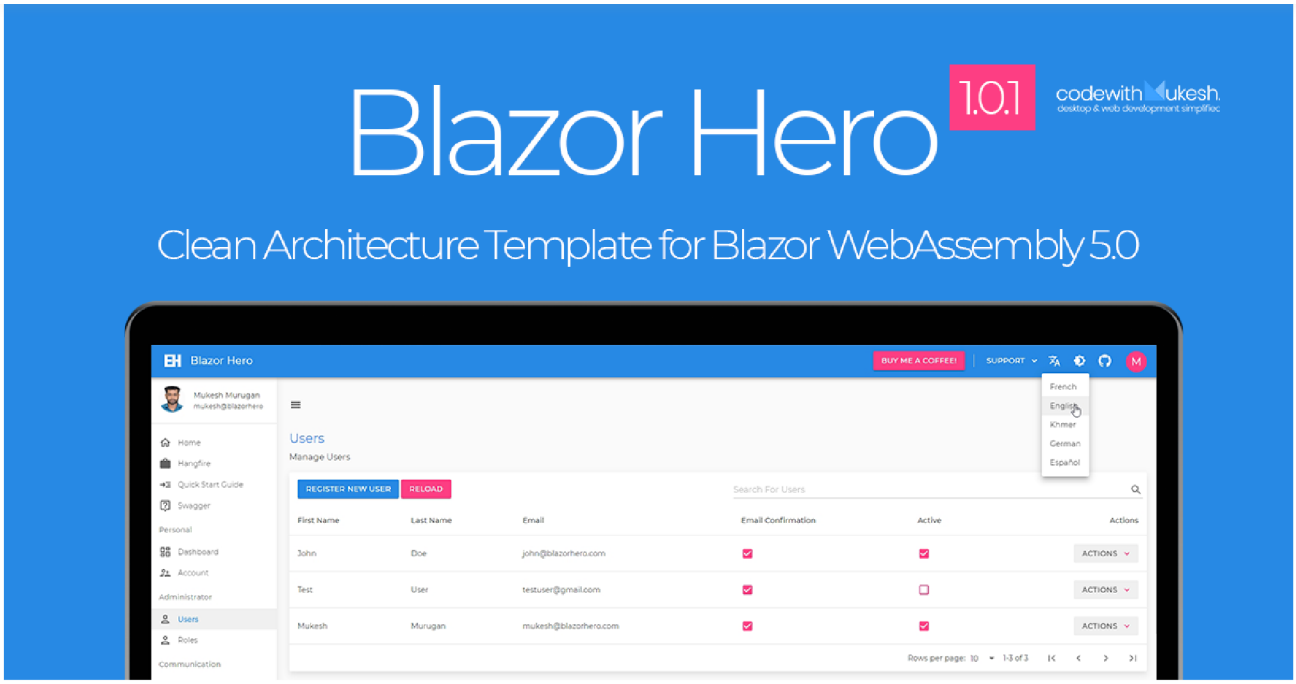



## Summary of Blazor CRUD Application

By now we are able to Create a Complete Blazor CRUD Application from scratch. It was quite easy too, yeah? We have covered basic concepts of Blazor CRUD and it's tags and component reusability. We have also seen a good practice folder structure that is quite applicable for Blazor CRUD Applications in general. I hope you guys have understood and enjoyed this detailed tutorial. You can find the completed source code here. As the next step, we will try to implement JWT Authentication in this same Blazor CRUD Project. We will be doing this in the next of the Blazor Blog Series.

## More Blazor Action!

I managed to put together a complete Blazor WASM 5.0 Project Template which you can install onto your machines and start generating Complete Blazor WebAssembly Project with just one line of CLI Code! Do check it out 😊




# Introducing Blazor Hero!

Blazor Hero – A Clean Architecture Template built for Blazor WebAssembly using MudBlazor Components. It's as easy as running a line of CLI command to start generating awesome Blazor Projects!

Get Started

Here is video as well, that takes you in-depth into Blazor Hero. Do Like and Subscribe to my YouTube channel for more content 🙂

Blazor Hero - Clean Architecture Solution Template for Blazor WebAssembly

Consider supporting me by buying me a coffee.

Thank you for visiting. You can buy me a coffee by clicking the button below. Cheers!


**How do you like my Blazor Blog Series? Is it easy to grasp?** Do you have any other queries/suggestions for me? Feel free to leave them below in the comments section. Happy Coding 😀

← PREVIOUS

Exploring Blazor Project Structure – Blazor For Beginners

NEXT →

How to Implement Pagination in ASP.NET Core WebAPI? – Ultimate Guide

## Similar Posts


### Blazor For Beginners – Getting Started with Blazor

By Mukesh Murugan May 30, 2020


### Implementing Blazor CRUD using Mudblazor Component Library in .NET 5 – Detailed Guide

By Mukesh Murugan February 17, 2021

## Leave a Reply

*Your email address will not be published. Required fields are marked* *

Comment *

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

# 54 Comments

**Thomas Bazan** says:
June 4, 2020 at 8:00 pm

Good tutorial, helped me to understand how to use Blazor a lot.

Had to Install-Package Microsoft.EntityFrameworkCore.SqlServer in order to make use of “UseSqlServer()”-method.

Thx for your effort.

Reply

---

**Mukesh Murugan** says:
June 4, 2020 at 8:12 pm

Oops, Guess I missed it out. I have updated the Article with the additional required package. Thanks for pointing it out! Blazor is quite an impressive stack, right? 😃 Thanks!

Reply

---

**Spencer** says:
June 5, 2020 at 6:13 am

Nice job, I love this. Do you have an example of the same thing using the Blazor Server? I think by making one and comparing the webassembly and server versions it can clearly depict the applicable differences and would make this a complete Blazor tutorial.

Once again, great job.

Reply

---

**Mukesh Murugan** says:
June 5, 2020 at 1:14 pm

Hey, Thanks! I am on the process of building content on Blazor. Yes, I will be doing the Server variant also quite soon.

Reply

---

**Nicolas** says:
June 5, 2020 at 9:12 am

Super great!

Reply

**Mukesh Murugan** says:
June 5, 2020 at 1:15 pm

Thanks a lot. There is Quite more content to come on Blazor 😀 😀

Reply

**Ted** says:
June 8, 2020 at 1:57 am

Very nice walkthrough, Mukesh, thank you for doing this.

Two questions:
1. Is scaffolding an option to build the Create/Edit/Delete CRUD pages?
2. Why javascript? Blazor is specifically designed to use C# instead of javascript, so why not C#?

Thank you again.

Reply

**Mukesh Murugan** says:
June 8, 2020 at 1:47 pm

1. As of now, I believe it is not possible to scaffold Razor/ Blazor Components. There might be ways like maybe a code snippet template on VS?
2. Honestly, I believe Blazor is still a not well-matured tech. That is exactly why Blazor supports JS interop. There is still quite a lot of work pending on Blazor. But I guess I have used C# almost everywhere except for the alert calls in this tutorial.

Thanks for the feedback Ted 😀

Reply

**Michał** says:
June 8, 2020 at 10:11 am

Great job! Thank you

Reply

**Mukesh Murugan** says:
June 8, 2020 at 1:48 pm

Thank you too, Michal! I hope the guide is quite clear.

Reply

**Godwin** says:
June 10, 2020 at 3:09 pm

Good job. I took some time to follow through and it was COOL. I look forward to getting more tutorials from you in more advanced projects. Your tutorial is detailed. Thank you

Reply

**Mukesh Murugan** says:
June 10, 2020 at 6:00 pm

Thanks Godwin! I am preparing the next article “Blazor WebAssembly Authentication”. Will be posted in a day 😀

Reply

**stefanov** says:
June 15, 2020 at 1:53 pm

Very clear article! Can’t wait to read more!

Reply

**Mukesh Murugan** says:
June 15, 2020 at 1:58 pm

Hey. Thanks for the feedback! More coming soon 😀

Reply

**Robby Robson** says:
June 22, 2020 at 8:17 pm

A few comments:

the expected “No Records Found” result from Fetch Data.

2. The tutorial was often silent on the using statements that were necessary to get a clean build. It is clear the tutorial was not aimed at less experienced developers like me. It would be helpful to do a little optional hand holding for folks like me just in case they were stymied by the errors.

3. Found a strange action by VS 2019 v16.6.2. In creating the “Data” folder in the shared project, I inadvertently left the default name of the new folder (NewFolder) on the folder and continued with the tutorial.

Later I saw my error and renamed the NewFolder to “Data”, which should have originally been the name. However, VS 2019 V16.6.2 does not respect that new name. It seems that the name “NewFolder” is somehow “sticky” in VS 2019 and I had to revert the name back from “Data” to “NewFolder” for everything to build properly (and get the ApplicationDbContext.cs to be recognized).

Reply

---

**Mukesh Murugan** says:

June 23, 2020 at 12:18 am

Hi, Great to hear.

I will try to leave a note in my future tutorials to help in adding references and so. I Hope that you liked the implementation.

Thanks for the feedback.

Reply

---

**Priscilla Cliffton** says:

July 21, 2020 at 3:48 pm

Thank you very much Mukesh for the simplicity and clarity of your code.

Pls let me know your email address.

Reply

---

**Mukesh Murugan** says:

July 21, 2020 at 7:32 pm

Hi, Thanks for the feedback.

You can reach me here – https://codewithmukesh.com/contact/ or hello@codewithmukesh.com

Thanks and Regards

Reply

---

**carlospizan** says:

Hola Mukesh Murugan , me parecio excelente manual, seria bueno un ejemplo maestro detalle

Gracias

Reply

**Mukesh Murugan** says:
July 21, 2020 at 12:00 am

Thanks and regards ✌

Reply

**Adam** says:
July 22, 2020 at 11:05 am

Hi thank you for this tutorial. Great work am new to blazor and have grasped some concept here.

Reply

**Mukesh Murugan** says:
July 22, 2020 at 1:38 pm

Hi, Thanks. Glad that I could help.

Regards

Reply

**Manfred Smith** says:
August 11, 2020 at 11:15 am

Excellent, straightforward & simple Blazor course. Many thanks

Reply

**Mukesh Murugan** says:
August 12, 2020 at 1:58 pm

Thanks for the feedback! 😀

Reply

**Tarik** says:
August 20, 2020 at 3:48 pm

Excellent walk through.
it simple and clear.
If i may suggest could you please doing like this walk through with A blazor server with the following :
1- Paging functionality – in its simple way – because this is an essential with huge data.
2- Using TailwindCss A utility-first CSS framework, it is great and powerful tools.
Thank you very much for sharing this article.
Best Regards.

Reply

**Mukesh Murugan** says:
August 20, 2020 at 4:46 pm

Hi,
I will be posting more on Blazor soon.
Thanks for the feedback.
Regards

Reply

**Jim** says:
September 2, 2020 at 11:16 am

I like what you guys tend to be up too. Such clever work and exposure!
Keep up the wonderful works guys I've included you guys to our
blogroll. Hello, I check your blog like every week. Your
story-telling style is awesome, keep doing what you're doing!
I could not resist commenting. Exceptionally well written! http://linux.com

Reply

**Ahmed Nazmy** says:
September 19, 2020 at 12:30 pm

Application shows message 'An unhandled error has occurred', when inspecting through google chrome I see the details below:

System.Net.Http.HttpRequestException: Response status code does not indicate success: 500 (Internal Server Error).
at System.Net.Http.HttpResponseMessage.EnsureSuccessStatusCode () in :0
at System.Net.Http.Json.HttpClientJsonExtensions.GetFromJsonAsyncCore[T] (System.Threading.Tasks.Task`1[TResult] taskResponse, System.Text.Json.JsonSerializerOptions options, System.Threading.CancellationToken cancellationToken) in :0
at Blazor.Learner.Client.Pages.Developer.FetchData.OnInitializedAsync () in :0
at Microsoft.AspNetCore.Components.ComponentBase.RunInitAndSetParametersAsync () in :0
at Microsoft.AspNetCore.Components.RenderTree.Renderer.GetErrorHandledTask (System.Threading.Tasks.Task taskToHandle) in :0

Reply

---

**Ruan** says:
September 23, 2020 at 12:14 pm

Have the SAME issue where you able to resolve??

Reply

---

**Chris** says:
February 7, 2021 at 6:10 am

I am having the same problem…is there a resolution? Mukesh….please help!!!! Same problem!!!

Reply

---

**Oyyou** says:
November 12, 2021 at 5:30 pm

2 things I had to do to get this working

1. Reference the .Client app in the .Server app. This gives the server an index.html to find
2. Change the AddDbContext
From: …options.UseSqlServer(Configuration.GetConnectionString(“DefaultConnection”));
To: …options.UseSqlServer(“”));

Reply

---

**Paul Hagerty** says:
September 23, 2020 at 9:55 pm

good stuff, great work

Reply

**Muhammad Saad** says:
October 15, 2020 at 10:51 am

Very nice article! Line wise description is good

My Question: On successful insertion, I dont see any record in SQL Server Database…..

Please guide

Reply

---

**Ryan Gray** says:
November 8, 2020 at 3:41 pm

Excellent tutorial! Very easy to follow along and well structured. I look forward to reading more of your stuff! A few I would love to see which all pertain to the list pages would be:

1. Implementing search criteria (aka filtering) the data.
2. Paging
3. Sorting

Thanks!

Reply

---

**josep temprano** says:
November 18, 2020 at 11:47 am

Very Nice introduction to Blazor!

It’s possible a future article about integrate Blazor in the Clean Architecture.

Thanks.

Josep

Reply

---

**Shannon D Slaton** says:
November 28, 2020 at 3:18 pm

Excellent article. Perfectly clear instruction and it helped me understand the change from MVC to Blazor. Thank you.

Reply

---

**Mukesh Murugan** says:

Thanks for the feedback.

Reply

**Bjorn Hellesylt** says:
December 7, 2020 at 3:08 am

Thanks a lot for a great article. But once the application is built, how to deploy such an application to Windows Server IIS 10? I have not found any good articles about this topic. Any tip on how to do that would be appreciated.

Reply

**Judi Smith** says:
January 14, 2021 at 6:06 pm

Very nice tutorial. Thank you.

I believe there was one small mistake in the Edit component.

The “dev” in dev=”dev” should be @dev.

Reply

**Orama** says:
January 14, 2021 at 10:18 pm

Great! Fantastic! Thank you for the time and information shared!

The Validation messages are there but I thunk the rules are missing upon defining the model class!

Happy Coding!
Orama

Reply

**S** says:
January 26, 2021 at 4:34 am

Already followed your step. but this happens

PM> add-migrations Initial
add-migrations : The term ‘add-migrations’ is not recognized as the name of a cmdlet, function, script file, or operable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try again.

+ add-migrations Initial

+ ~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound: (add-migrations:String) [], CommandNotFoundException

+ FullyQualifiedErrorId : CommandNotFoundException

Reply

**Mukesh Murugan** says:

January 26, 2021 at 4:52 am

Hi, it's add-migration, not migrations.

Hope that will fix the issue. Also make sure you have installed the lastest .NET CLI tools.

Regards

Reply

**Mario G Vernaza P** says:

January 26, 2021 at 3:38 pm

Excelent demo. I have reproduced the step using .Net Core 5 and after fixing some issues with the names of components and variables/parameters it worked in an amaizing way.

Thanks

Reply

**Daniel** says:

January 26, 2021 at 4:01 pm

Can I apply this to the server side? I am not sure if you are still continuing to write a post about server/crud operation.

Thank you!

Reply

**Enzo Calcagno** says:

February 24, 2021 at 10:51 pm

As a beginner with Blazor, this is a great tutorial. Planning on more tutorials? Razor Components?

Reply

March 13, 2021 at 8:32 pm

Super helpful! Thank you!

Reply

**Mukesh Murugan** says:
March 13, 2021 at 9:16 pm

You are welcome!

Reply

**sylvain** says:
April 13, 2021 at 4:13 pm

Hello, thank you for your tutorial.
I have questions:
_ in the _imports file I had put @using BlazorApp1.Server.Models instead of @using BlazorApp1.Shared.Models, do you know why?
_ I would like to know if I can put other information in the @developer/edit page. For example id in a card box and add text box for comments for example. I think I have to modify the database to add this new data (in the text boxes) but this data should not be displayed in the table
_ Is it also possible to add lists for each row in the table?

Thanks in advance

Reply

**Bob Needham** says:
May 8, 2021 at 7:31 am

Thanks for the tutorial, it was just what i was looking for to get me started on blazor and the best i found for this stage of my learning :). One thing though that may help, (and it was something i discovered from reading another article), and that's to add the controller you can use the 'Add > New Item > API controller with read/write actions using entity framework', which makes the controller methods a little more up to date, just watch out for the spelling though (entity calls it DevelopersController (plural) vs your 'DeveloperController' which fooled me for a while when it wouldnt work but the code still compiles lol. that said both yours and the auto created one both seem to work just fine. keep up the good work. Bob

Reply

**Majid** says:
July 23, 2021 at 7:24 pm

Dear mukesh

Many thanks for very simple trainee for Blazor.

does blazor hero is free?

Reply

**Mukesh Murugan** says:

July 24, 2021 at 3:16 pm

Yes,blazorHero is completely free 🙂

Reply

**Willie van Schalkwyk** says:

August 16, 2021 at 3:46 pm

Hi

How can I add a order to the GetAllAsync.

Also is there a way to call a stored procedure?

Kind regards

Willie

Reply

**Willie van Schalkwyk** says:

August 17, 2021 at 5:37 am

Don't wary, used OrderBy in the result set.

Reply

**Aaron** says:

June 7, 2022 at 3:37 am

I am getting an error at line56 of the FetchData.

line56: developers = await client.GetFromJsonAsync("api/developer");

The error states: Unhandled exception rendering component: The provided ContentType is not supported; the supported types are 'application/json' and the structured syntax suffix 'application/+json'.

I was reading that this has to do with reading html/text instead of Json.

I am wondering in this example how would I fix this or use an alternate method instead of client.GetFromJsonAsync()

Reply