

Get started with Blazor

TAGS: [ASP.NET CORE](#), [BLAZOR](#)

- [Admin](#)
- September 16, 2019
- [Dotnet / Technology](#)
- [2 Comments](#)

Blazor is a new web UI framework based on C#, Razor, and HTML. This runs in the browser via WebAssembly. It helps build interactive web UI using C# instead of JavaScript. This post demonstrates how to build a SPA using Blazor. Blazor simplifies the task of building fast and beautiful SPAs that run in any browser. It does this by enabling developers to write Dotnet based web apps that run client-side in web browsers using open web standards. Let's get started with Blazor.

In this post, we will discuss the following

- Hosting Models
- Enable Authentication and Authorization
- Dive deep into Default Blazor pages

Prerequisites

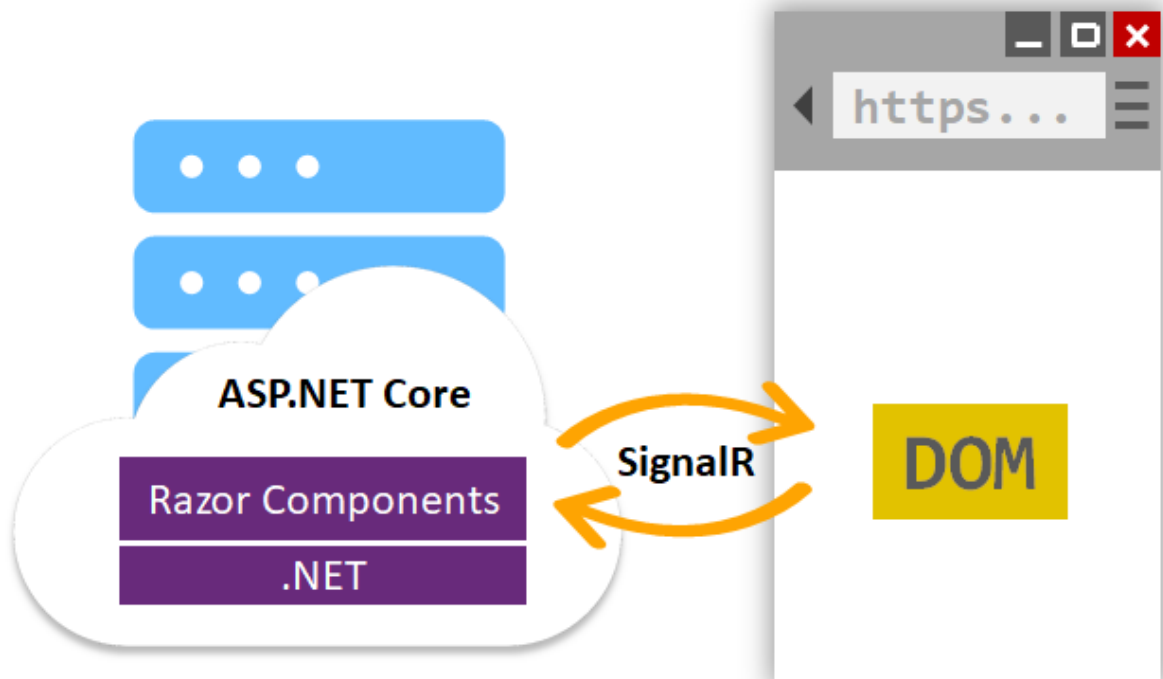
- Visual Studio 2019
- Install .NET Core 3.0
- Install Blazor Templates

Hosting models

You have the option to host the app in the server as razor components or run the app in the browser on WebAssembly.

Server-side

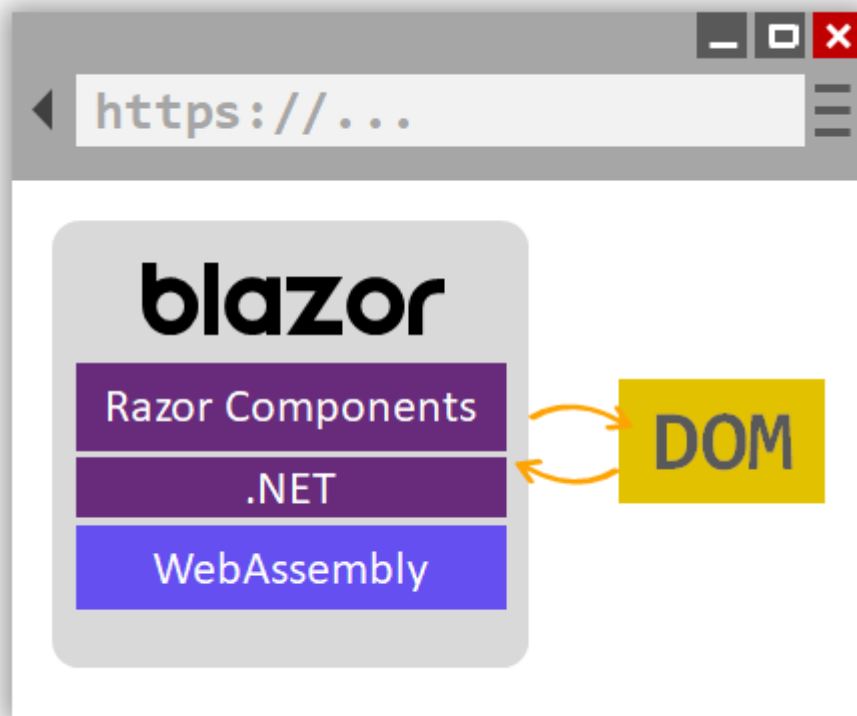
Supports hosting Razor components on the server inside an ASP.NET core app and handles user interactions over a SignalR connection.



Source – Microsoft Docs

Client-side

The Blazor app along with the .NET runtime and other dependencies downloaded to the browser. You can share the model, validation and other business logic between client and server-side. You can also take advantage of several libraries that will run directly in .NET on the browser.

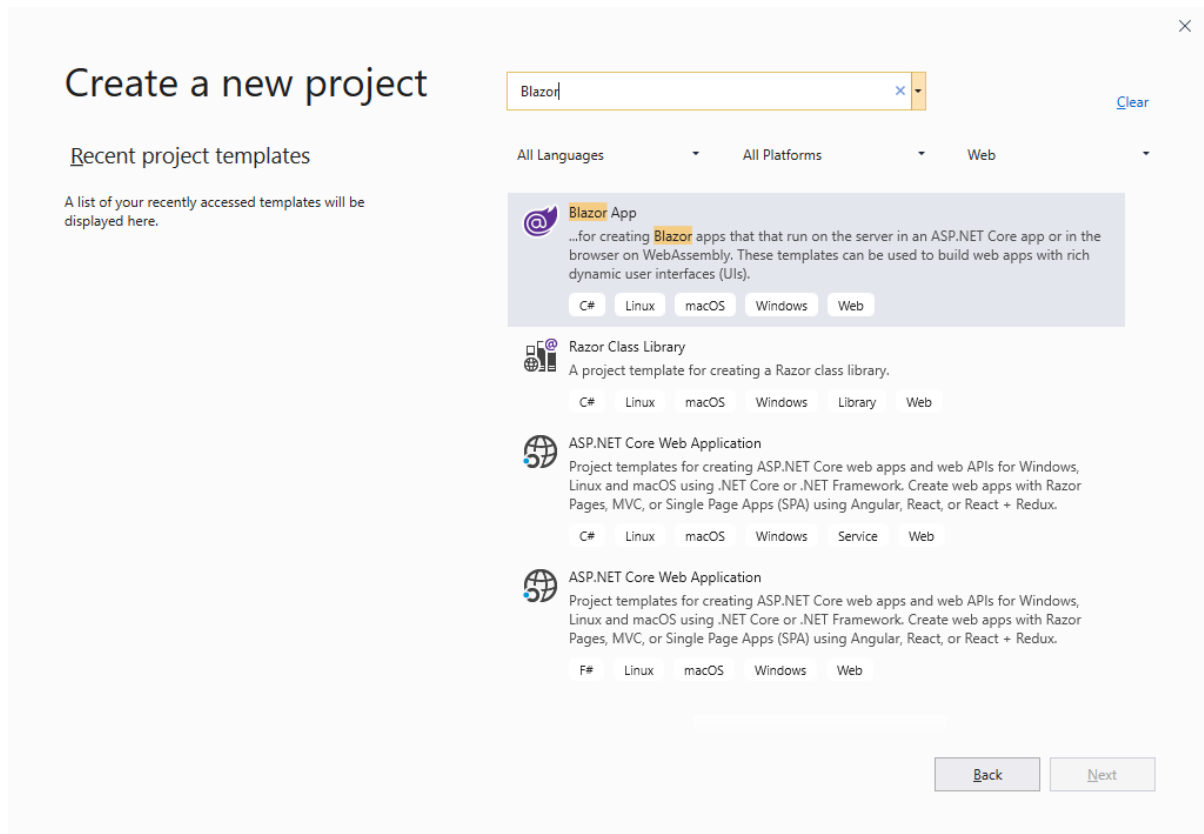


Source – Microsoft

Docs

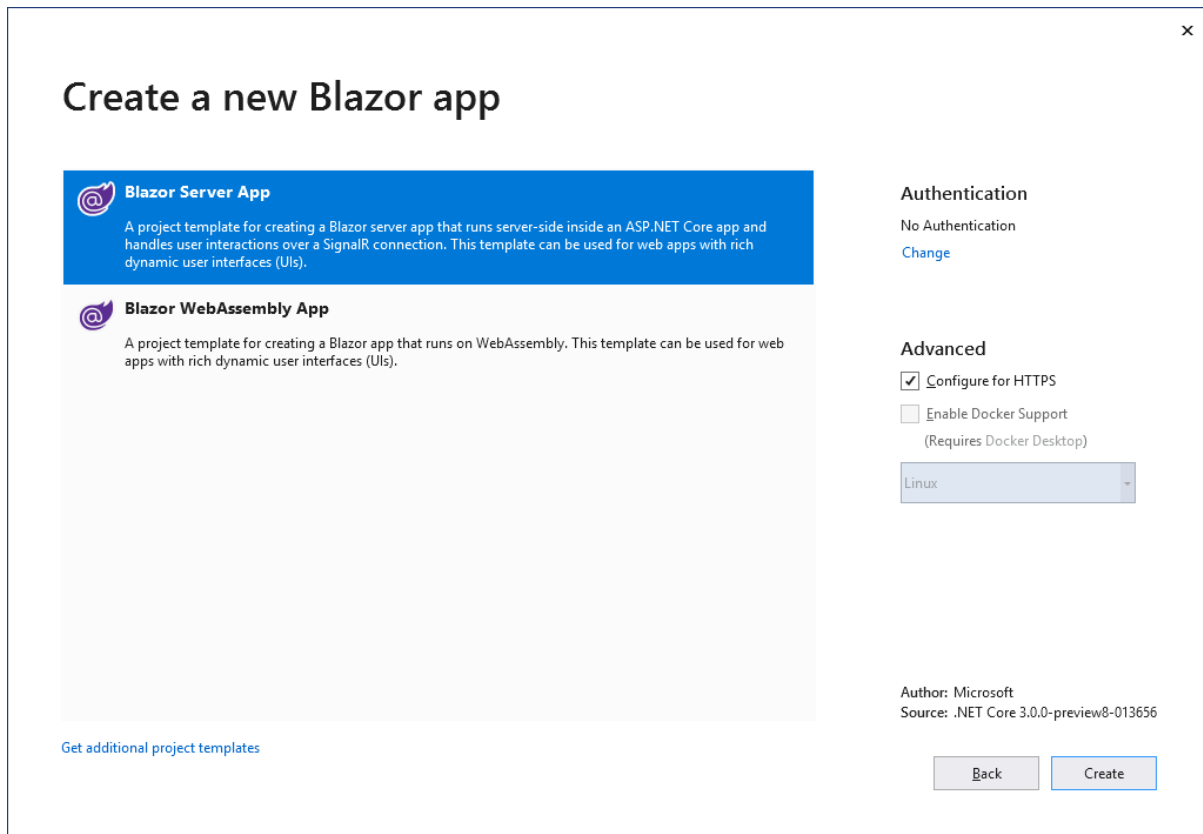
Each of the models has its own benefits and downsides. Most of them are related to dependency, performance, browser compatibility, etc. The decision of choosing one of the approaches for implementation is up to you. We will discuss server-side hosting with an example, the however decision of one over other is not the main purpose of this post.

Get Started

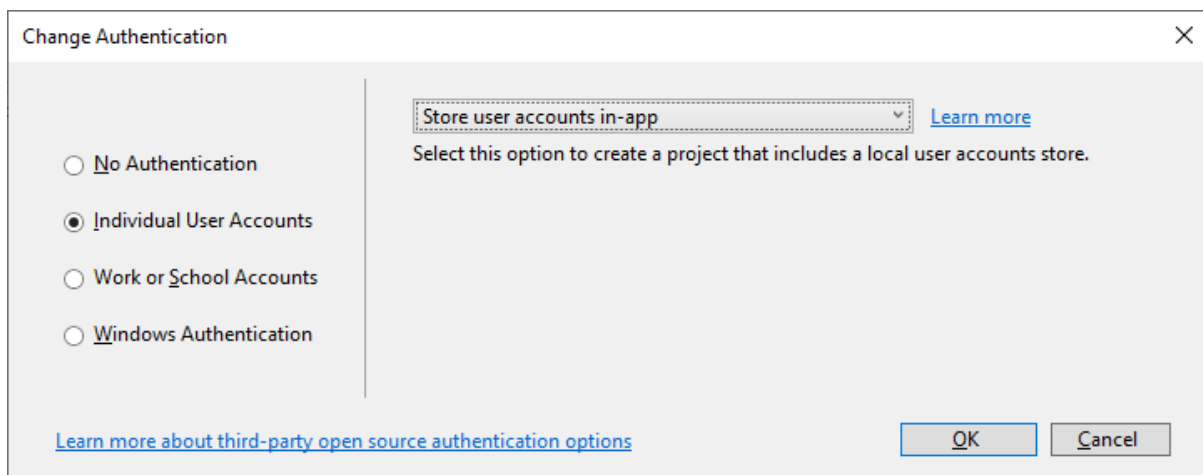


Search for “Blazor” when creating the new project and select “Blazor App”

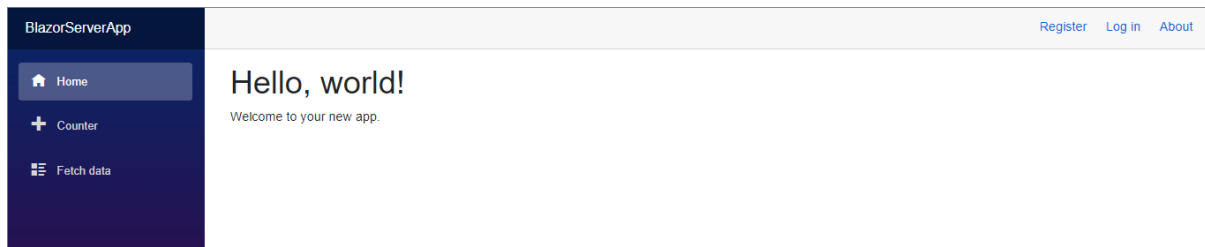
On the next page, select the type of app you want to create. I am going to create a “Blazor Server App” for the purpose of this post.



Before creating the project, click the “Change” link under the “Authentication” section. Select “Store user accounts in-app”. This option lets us use a local user accounts store for authentication. Other options available are No Authentication, Work or School Accounts and Windows Authentication. You can disable the HTTPS from the properties page (Debug tab) of the project later.

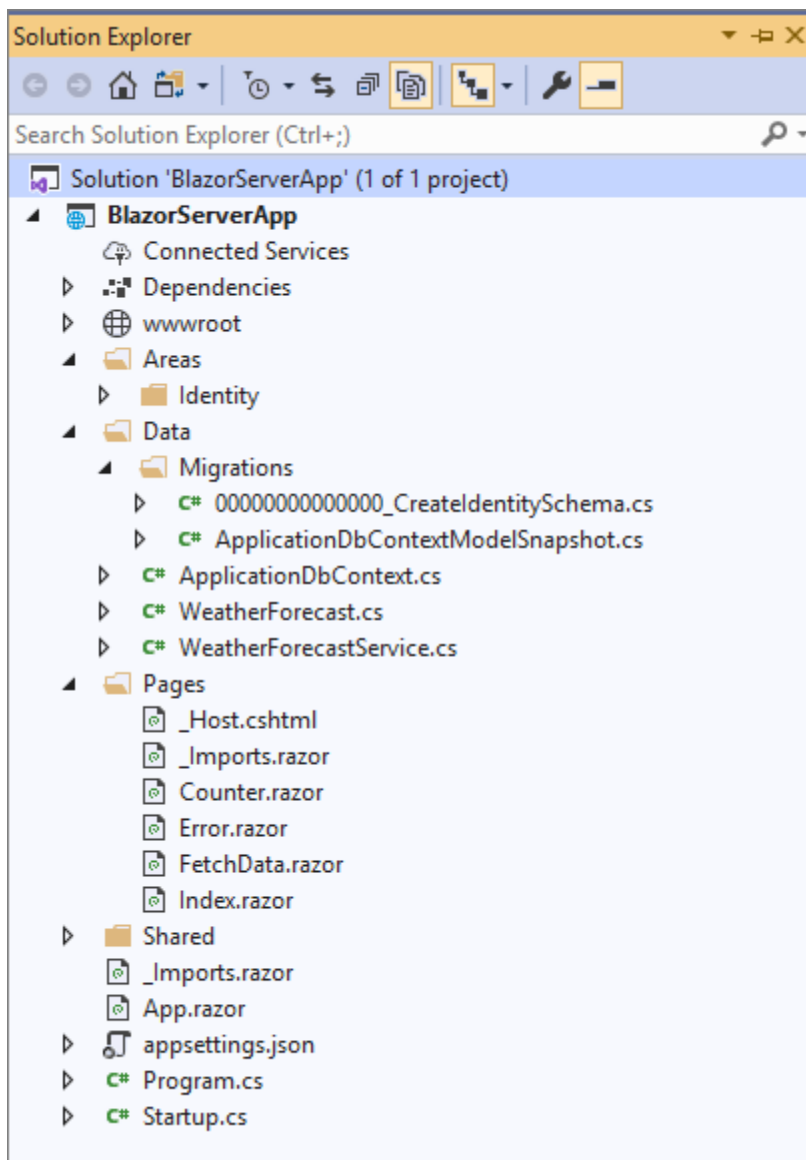


Now we have created the project with authentication enabled. When you run the project, you can see the following screen.



Understanding the project structure

The structure of the project has some components similar to an MVC application like Areas, appsettings.json, Program.cs, and Startup.cs. In addition to that, you can also see files with razor extension and they are specific to “Blazor”. Let’s talk about them in detail.



Identity – This folder was created because we have selected an authentication methodology while creating the project. It has an “AuthenticationServiceProvider” class and HTML files for Login and Logout.

Data – Nuget packages related to Entity Framework are available by default while creating the project. This folder contains a migration file to create tables related to authentication

like Users, Roles, etc. It also contains the “DbContext” file and a model and service to get the weather forecast details for the default demo page.

Pages – If you are a .NET developer, you may already hear about the razor. Blazor apps are based on components. Components are reusable building blocks, it can be an individual control or a block with multiple controls. These component classes are written in razor markup. However, the main difference of razor in the context Blazor is, it is built around UI logic and not on request/response delivery.

Enable Authentication and Authorization

To enable the authentication, do the following steps

- Create the tables to store user and role details
- Add User and Roles
- Implement authorization

Create tables

Run the Entity Framework’s “update-database” command to trigger the table creation. By default, the application creates the database in “localdb”. You can modify the connection string in appsettings.json if required. The default migration creates tables related to authentication like AspNetUsers, AspNetRoles, etc.

User Registration

You can add users to the application in two ways. The first is to use the “Register” UI. This will help add the user to the system.

BlazorServerApp Register Login

Register

Create a new account.

Email

Password

Confirm password

Register

Use another service to register.

There are no external authentication services configured. See [this article](#) for details on setting up this ASP.NET application to support logging in via external services.

© 2019 - BlazorServerApp - [Privacy](#)

The next option is to seed the User and Role data from “Startup.cs”. Role services are not added by default but can be added with `AddRoles<TRole>()`. You have to modify the “ConfigureServices” method to enable Roles.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>()
        .AddRoles<IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();
    services.AddServerSideBlazor();
    services.AddScoped<AuthenticationStateProvider,
RevalidatingAuthenticationStateProvider<IdentityUser>>();
    services.AddSingleton<WeatherForecastService>();
}

```

I use the following method to seed both Roles and Users in the Identity tables. The method is called inside the "Configure" method of "Startup.cs". It uses RoleManager and UserManager to check and add Roles and Users.

```

private async Task CreateUserAndRoles(IServiceProvider serviceProvider)
{
    //initializing custom roles
    var RoleManager =
serviceProvider.GetRequiredService<RoleManager<IdentityRole>>();
    var UserManager =
serviceProvider.GetRequiredService<UserManager<IdentityUser>>();
    string[] roleNames = { "Admin", "User" };
    IdentityResult roleResult;

    foreach (var roleName in roleNames)
    {
        var roleExist = await RoleManager.RoleExistsAsync(roleName);
        if (!roleExist)
        {
            //create the roles and seed them to the database: Question 1
            roleResult = await RoleManager.CreateAsync(new
IdentityRole(roleName));
        }
    }

    IdentityUser user = await
UserManager.FindByEmailAsync("admin@blogofpi.com");

    if (user == null)
    {
        user = new IdentityUser()
        {
            UserName = "admin@blogofpi.com",
            Email = "admin@blogofpi.com",

```



```

    };
    await UserManager.CreateAsync(user, "Test@579");
}
await UserManager.AddToRoleAsync(user, "Admin");

IdentityUser user1 = await
UserManager.FindByEmailAsync("jane.doe@blogofpi.com");

if (user1 == null)
{
    user1 = new IdentityUser()
    {
        UserName = "jane.doe@blogofpi.com",
        Email = "jane.doe@blogofpi.com",
    };
    await UserManager.CreateAsync(user1, "Test@246");
}
await UserManager.AddToRoleAsync(user1, "User");
}

```

Implement Authorization

Now we have created the tables, add users and roles. Let's enable authorization to pages. Blazor has an "AuthorizeView" component which helps to display content depending on the authorization status. If you have the page content inside "AuthorizeView", only an authorized user can see it.

```

<AuthorizeView>
    <h1>Counter</h1>
    <p>Current count: @currentCount</p>
    <button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
</AuthorizeView>

```

What if you need to show different content for authorized and unauthorized users? You can use "Authorized" and "NotAuthorized" elements inside "AuthorizeView" to serve different content based on the authorization status.

```

<AuthorizeView>
    <Authorized>
        <h1>Counter</h1>
        <p>Current count: @currentCount</p>

        <button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>
    </Authorized>
    <NotAuthorized>
        You are not authorized to view this page!
    </NotAuthorized>
</AuthorizeView>

```

```
</NotAuthorized>
</AuthorizeView>
```

You can authorize users based on Role or Policy. It can be implemented by mention the Policy and Roles as attributes inside the “AuthorizeView” element. You can also use the “Authorize” attribute. The name of the roles and policy mentioned inside AuthorizeView is case sensitive.

```
<AuthorizeView Roles="Admin">
  <Authorized>
    <h1>Counter</h1>
    <p>Current count: @currentCount</p>

    <button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>
  </Authorized>
  <NotAuthorized>
    You are not authorized to view this page!
  </NotAuthorized>
</AuthorizeView>
```

Dive deep into Blazor Pages

Let’s analyze a razor component and try to understand the basic building blocks of it. I am going to use “FetchData.razor” for this.

```
@page "/fetchdata"

@using BlazorServerApp.Data
@inject WeatherForecastService ForecastService

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

@if (forecasts == null)
{
  <p><em>Loading...</em></p>
}
else
{
  <table class="table">
    <thead>
      <tr>
        <th>Date</th>
        <th>Temp. (C)</th>
        <th>Temp. (F)</th>
        <th>Summary</th>
      </tr>
```

```

        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }
}

```

@page – The route attribute in a component is set using “@page” attribute. This specifies that the component is a routing endpoint. A component can have multiple route attributes by having multiple “@page” directives.

@inject – You can inject the services into the component using the “@inject” attribute. In the example, “WeatherForecastService” is injected and later used to retrieve the data. You can use the services from registering them in “Startup.cs” also similar to ASP.NET MVC.

@code – This block contains the code for rendering and event handling. It can be anything like variable declarations to methods. There is something similar to “@code” which is “@functions” and used for the same purpose. From ASP.NET Core 3.0, it is suggested to use “@code” instead of “@function”.

Summary

This post tried to provide an introduction of Blazor and how to create your first application with Blazor. We also discussed the hosting models, implementation of authentication, authorization and the directives used in the default pages.

The full implementation of this post will be available in [Github](#)

CRUD using Blazor and Entity Framework Core

TAGS: [ASP.NET CORE](#), [BLAZOR](#), [DOTNET CORE](#), [ENTITY FRAMEWORK](#)

In this post, I am going to explain CRUD using Blazor and Entity Framework Core. I will use the sample application we have created in the previous post and extend it with the new functionalities we are going to talk about in this post. You will be familiar with the following topics after you read this post.

- Entity Framework Core – Setup in Blazor application
- Using Bootstrap Modals
- Child Components
- Validation
- JavaScript Interop
- Communication Between Components
- Templated Components using Render Fragments (Dynamic Content)

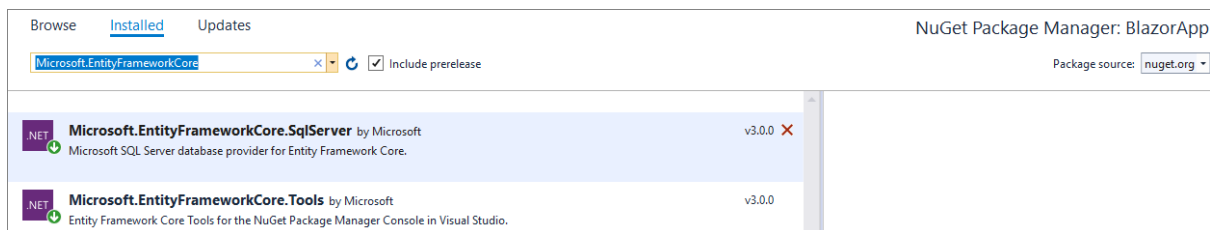
Okay, Let's get started.

Prerequisites

- Visual Studio 2019
- Install .NET Core 3.0
- Install Blazor Templates

Entity Framework Core – Setup

To begin with, make sure you have the following NuGet packages installed in your project.



(Package versions in the image were latest at the time of writing this post)

For the purpose of this post, I create a model called `ToDo.cs` and we will use this model for CRUD operations.

```
// ToDo.cs

public class ToDo
{
    [Key]
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Required]
    public string Status { get; set; }

    [Required]
    public DateTime DueDate { get; set; }
}
```

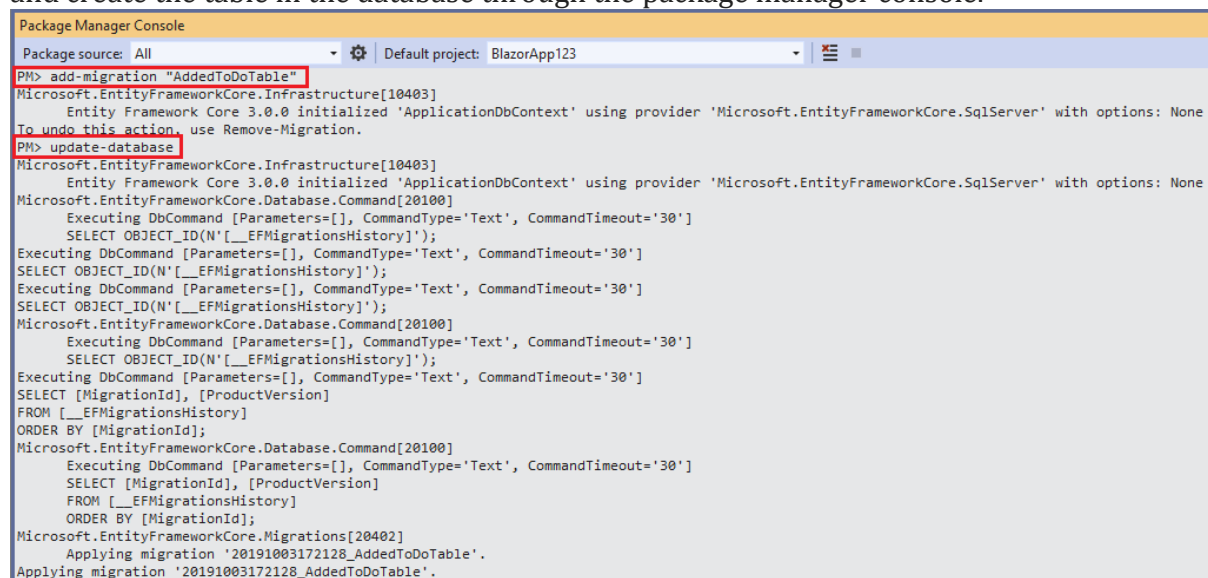
Add the entity in the `ApplicationDbContext.cs`

```
// ApplicationDbContext.cs

public class ApplicationDbContext : IdentityDbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }
    public DbSet<ToDo> ToDoList { get; set; }

    public override int SaveChanges()
    {
        return base.SaveChanges();
    }
}
```

Now we have added the entity and made required changes in **dbcontext**. Add the migration and create the table in the database through the package manager console.



```
Package Manager Console
Package source: All Default project: BlazorApp123
PM> add-migration "AddedToDoTable"
Microsoft.EntityFrameworkCore.Infrastructure[10403]
Entity Framework Core 3.0.0 initialized 'ApplicationDbContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
To undo this action, use Remove-Migration.
PM> update-database
Microsoft.EntityFrameworkCore.Infrastructure[10403]
Entity Framework Core 3.0.0 initialized 'ApplicationDbContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Microsoft.EntityFrameworkCore.Database.Command[20100]
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20100]
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [MigrationId], [ProductVersion]
FROM [__EFMigrationsHistory]
ORDER BY [MigrationId];
Microsoft.EntityFrameworkCore.Database.Command[20100]
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [MigrationId], [ProductVersion]
FROM [__EFMigrationsHistory]
ORDER BY [MigrationId];
Microsoft.EntityFrameworkCore.Migrations[20402]
Applying migration '20191003172128_AddedToDoTable'.
Applying migration '20191003172128_AddedToDoTable'.
```

Data access service

We have created the table in the database. Create a service to access the table and perform CRUD operations. This service implements from an interface and the interface is configured in the startup for dependency injection.

```
// ToDoListService.cs

using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using BlazorApp.Data;

namespace BlazorApp.Services
```

```
{
    public interface IToDoListService
    {
        Task<List<ToDo>> Get();
        Task<ToDo> Get(int id);
        Task<ToDo> Add(ToDo toDo);
        Task<ToDo> Update(ToDo toDo);
        Task<ToDo> Delete(int id);
    }
    public class ToDoListService : IToDoListService
    {
        private readonly ApplicationDbContext _context;

        public ToDoListService(ApplicationDbContext context)
        {
            _context = context;
        }
        public async Task<List<ToDo>> Get()
        {
            return await _context.ToDoList.ToListAsync();
        }

        public async Task<ToDo> Get(int id)
        {
            var toDo = await _context.ToDoList.FindAsync(id);
            return toDo;
        }

        public async Task<ToDo> Add(ToDo toDo)
        {
            _context.ToDoList.Add(toDo);
            await _context.SaveChangesAsync();
            return toDo;
        }

        public async Task<ToDo> Update(ToDo toDo)
        {
            _context.Entry(toDo).State = EntityState.Modified;
            await _context.SaveChangesAsync();
            return toDo;
        }

        public async Task<ToDo> Delete(int id)
        {
            var toDo = await _context.ToDoList.FindAsync(id);
            _context.ToDoList.Remove(toDo);
            await _context.SaveChangesAsync();
            return toDo;
        }
    }
}
```

```
}
```

Okay, the data access part is now completed. Make sure, you have added the following code at the end of `ConfigureServices` method in `startup.cs`. This is to inject `ToDoListService`.

```
// Startup.cs
```

```
services.AddTransient<IToDoListService, ToDoListService>();
```

Details Page

Let's create a page to list the records from the database. I have copied the default `FetchData.razor` and made changes to show the sample records populated in the To-Do List table from the database. `NavMenu.razor` has been changed to show the "To-Do List" link.

```
// ToDoList.razor
```

```
@page "/todolist"
```

```
@using BlazorApp.Data
```

```
@using BlazorApp.Services
```

```
@inject IToDoListService service
```

```
<h1>To Do List</h1>
```

```
<p>This component demonstrates fetching data from Database.</p>
```

```
@if (todoList == null)
```

```
{
```

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

```
}
```

```
else
```

```
{
```

```
    <table class="table">
```

```
        <thead>
```

```
            <tr>
```

```
                <th>Task</th>
```

```
                <th>Status</th>
```

```
                <th>Due Date</th>
```

```
                <th>Edit</th>
```

```
                <th>Delete</th>
```

```
            </tr>
```

```
        </thead>
```

```
        <tbody>
```

```
            @foreach (var todoItem in todoList)
```

```
            {
```

```
                <tr>
```

```
                    <td>@todoItem.Name</td>
```

```
                    <td>@todoItem.Status</td>
```

```
                    <td>@todoItem.DueDate.ToShortDateString()</td>
```

```

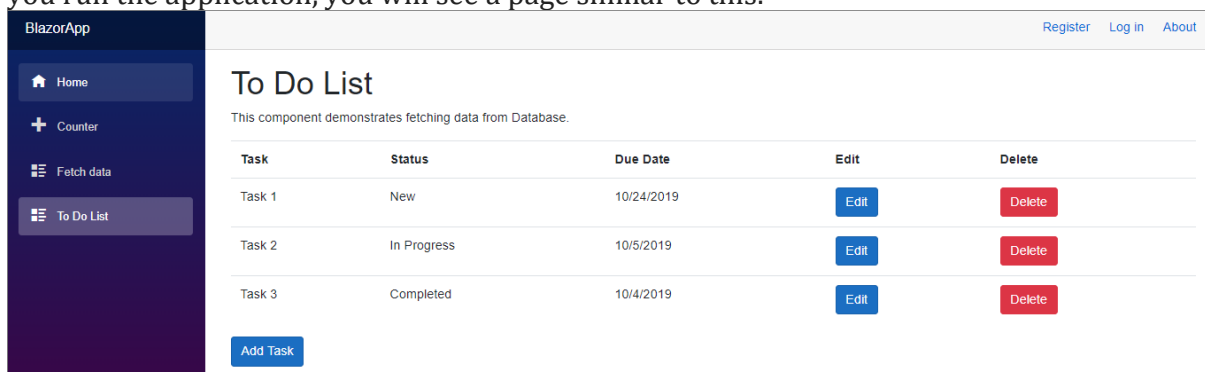
        <td><input type="button" class="btn btn-primary"
value="Edit" /></td>
        <td><input type="button" class="btn btn-danger"
value="Delete" /></td>
    </tr>
    }
</tbody>
</table>
}
<div>
    <input type="button" data-toggle="modal" data-target="#taskModal"
class="btn btn-primary" value="Add Task" />
</div>

@code {
    List<ToDo> toDoList;

    protected override async Task OnInitializedAsync()
    {
        toDoList = await service.Get();
    }
}

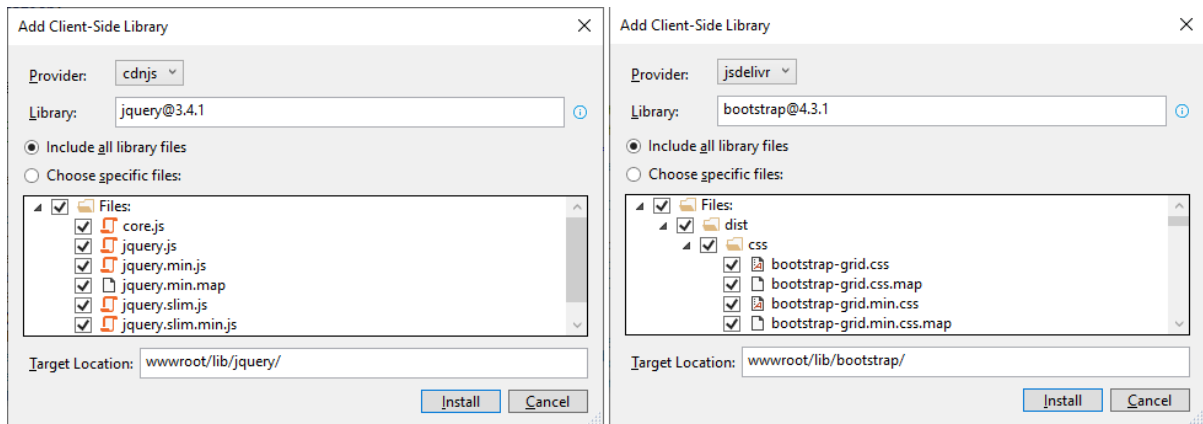
```

This code is very similar to `FetchData.razor` except this is fetching data from the database using Entity Framework Core. The service `IToDoListService` is injected at the top. Now we have to call the service to fetch the data. The right place to make a service call is inside `OnInitializedAsync`. It is one of the Blazor Lifecycle methods. It is executed when the component is completely loaded. You can use this method to load data by calling services because the control rendering will happen after this method. I have also added the code to display buttons for CRUD operations but not wired with any code as of now. When you run the application, you will see a page similar to this.



Add Bootstrap

To use bootstrap modal dialog, I have added jQuery and Bootstrap libraries using “Add Client-side Library”. You can find the option by right-clicking your project then Add -> Client-side Library



Include the following lines inside the body of `_Host.cshtml`

```
// _Host.cshtml

<script src="~/lib/jquery/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.min.js"></script>
```

Child Components

Before we proceed with the CRUD implementations, we need to know about child components. Blazor apps are based on components. Components are reusable building blocks, it can be an individual control or a block with multiple controls. These component classes are written in razor markup.

Components can include other components. You can add a component inside others using the component name in an HTML syntax. We will use this concept to create bootstrap modals as child components for Add/Edit and Confirmation dialogs.

Add Task

The next step is to create a razor component to accept the input from the user for a new To-Do item. I have created a razor component named `TaskDetail.razor`.

```
// TaskDetail.razor

@using BlazorApp.Data
@using BlazorApp.Services
@inject IToDoListService service

<div class="modal" tabindex="-1" role="dialog" id="taskModal">
  <div class="modal-dialog" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title">Task Detail</h5>
        <button type="button" class="close" data-dismiss="modal"
aria-label="Close">
          <span aria-hidden="true">x</span>
        </button>
      </div>
```

```

        <div class="modal-body">
            <EditForm Model="@TaskObject"
OnValidSubmit="@HandleValidSubmit">
                <div class="form-group">
                    <label for="taskName">Task Name</label>
                    <input type="hidden" @bind-value="@TaskObject.Id" />
                    <InputText id="name" class="form-control" @bind-
Value="@TaskObject.Name" />
                </div>
                <div class="form-group">
                    <label for="status">Status</label>
                    <InputSelect id="Summary" class="form-control"
                        @bind-Value="TaskObject.Status">
                        <option value="">Select</option>
                        @foreach (var status in TaskStatusList)
                        {
                            <option value="@status">
                                @status
                            </option>
                        }
                    </InputSelect>
                </div>
                <div class="form-group">
                    <label for="dueDate">Due Date</label>
                    <input type="date" id="addition" name="math" @bind-
value="@TaskObject.DueDate" />
                </div>
                <button type="submit" class="btn btn-
primary">Submit</button>
                <button type="button" class="btn btn-secondary" data-
dismiss="modal">Cancel</button>
            </EditForm>
        </div>
    </div>
</div>

@code {
    [Parameter]
    public ToDo TaskObject { get; set; }

    List<string> TaskStatusList = new List<string>() { "New", "In Progress",
"Completed" };

    private async void HandleValidSubmit()
    {

    }
}

```

In the above code, we have a form defined using the `EditForm` component. You can see the `EditForm` has a model that is passed from the parent component. The properties of the model are bind to the input controls using `bind-value`. `HandleValidSubmit` is triggered when the form successfully submits.

I have also declared the `TaskDetail` Component inside `ToDoList` component and pass an empty `ToDo` object. Find below the `ToDoList` component with changes.

```
// ToDoList.razor

@page "/todolist"

@using BlazorApp.Data
@using BlazorApp.Services
@inject IToDoListService service

<h1>To Do List</h1>

<p>This component demonstrates fetching data from Database.</p>

// Code omitted for brevity

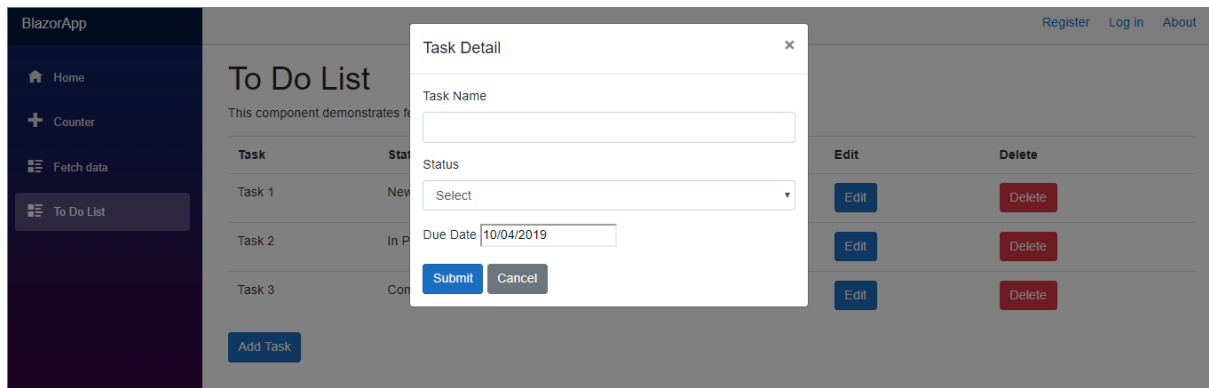
<div>
    <input type="button" data-toggle="modal" data-target="#taskModal"
class="btn btn-primary" value="Add Task" @onclick="(() => InitializeTaskObject())" />
</div>

<TaskDetail TaskObject=taskObject></TaskDetail>

@code {
    List<ToDo> toDoList;
    ToDo taskObject = new ToDo();

    protected override async Task OnInitializedAsync()
    {
        toDoList = await service.Get();
    }
    private void InitializeTaskObject()
    {
        taskObject = new ToDo();
        taskObject.DueDate = DateTime.Now;
    }
}
```

When you click on the “Add New” button, you will see a modal dialog similar to this.



What are the next steps?

- Validate the user input
- Save the data to the database
- Refresh the data in the page to show the new record

Validation

We will use data annotations in the model to validate the user input. I have changed the model to have custom validation messages

```
// ToDo.cs

public class ToDo
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Task name is required")]
    [StringLength(15, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Status is required")]
    public string Status { get; set; }

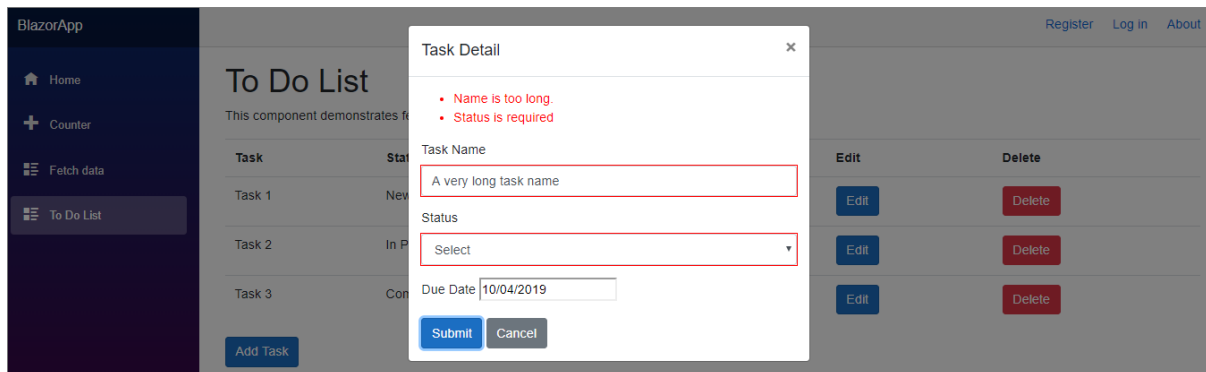
    [Required(ErrorMessage = "Due Date is required")]
    public DateTime DueDate { get; set; }
}
```

We also need to add `DataAnnotationsValidator` component which attaches validation support using data annotations. To display the validation messages we use `ValidationSummary` component. Both the components are added to `TaskDetail.razor` inside the `EditForm` component.

```
// ToDoList.razor

<DataAnnotationsValidator />
<ValidationSummary />
```

With these changes, if I run the application and try to submit an invalid form, I will get an error screen similar to the one below.

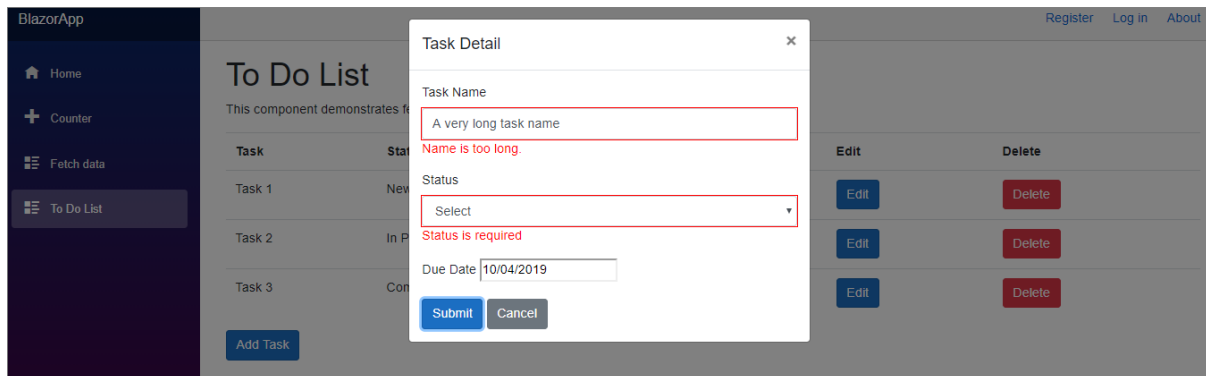


You can see the validation errors in the Validation Summary section. If you want to show the validation message next to each control instead of a summary, remove the `ValidationSummary` component and have the following pattern of code next to each input element.

```
// TaskDetail.razor

<ValidationMessage For="@(() => TaskObject.Name)" />
```

With this, the validation errors will be displayed next to the respective control similar to this.



Save Data

Now we received the data from the user and validated it. Let's save the data to the database. As I mentioned earlier, `HandleValidSubmit` is triggered when the form successfully submits after it passes the validation. We have to add the "save" logic in the method.

```
// TaskDetail.razor

private async void HandleValidSubmit()
{
    await service.Add(TaskObject);
}
```

JavaScript Interop

The data is saved to the database but the modal dialog is still open. We need to call JavaScript code from .NET code to close the dialog. To call JavaScript function from C#, use the `IJSRuntime` abstraction. The `InvokeAsync<T>` method takes an identifier for the JavaScript function that you wish to invoke along with any number of JSON-serializable arguments.

First, we have to create a JavaScript method to close the bootstrap dialog by getting the id of the dialog. The second is to inject the `IJSRuntime` and lastly use the injected object to issue JavaScript Interop calls. The following JavaScript code is added to the `_Host.cshtml`.

```
// _Host.cshtml

<script>
    function ShowModal(modalId) {
        $('#'+modalId).modal('show');
    }
    function CloseModal(modalId) {
        $('#'+modalId).modal('hide');
    }
</script>
```

After the changes, TaskDetail.razor looks like this

```
// TaskDetail.razor

@using BlazorApp.Data
@using BlazorApp.Services
@inject IToDoListService service
@inject IJSRuntime jsRuntime

// Code omitted for brevity

@code {
    [Parameter]
    public Todo TaskObject { get; set; }

    List<string> TaskStatusList = new List<string>() { "New", "In Progress",
    "Completed" };

    private async Task CloseTaskModal()
    {
        await jsRuntime.InvokeAsync<object>("CloseModal", "taskModal");
    }

    private async void HandleValidSubmit()
    {
        await service.Add(TaskObject);
        await CloseTaskModal();
    }
}
```

```
}
```

Communication Between Components

Now the data is saved and the modal is closed, but I cannot see the newly added item in the grid. I can see the data only if I refresh the browser. I have to tell the parent component to refresh itself to display the new set of data. Components can offer callbacks that parent components can use to react on events raised by their child components. Let's see how to implement this. I have declared an action `DataChanged` and invoke it in the `HandleValidSubmit` method of `TaskDetail.razor`

```
// TaskDetail.razor

@code {
    [Parameter]
    public Todo TaskObject { get; set; }

    [Parameter]
    public Action DataChanged { get; set; }

    List<string> TaskStatusList = new List<string>() { "New", "In Progress",
    "Completed" };

    private async Task CloseTaskModal()
    {
        await jsRuntime.InvokeAsync<object>("CloseModal", "taskModal");
    }

    private async void HandleValidSubmit()
    {
        await service.Add(TaskObject);
        await CloseTaskModal();
        DataChanged?.Invoke();
    }
}
```

The parent component `ToDoList.razor` can handle the `DataChanged` event like this

```
// ToDoList.razor

// Code omitted for brevity
<TaskDetail TaskObject=taskObject DataChanged="@DataChanged">

</TaskDetail>

@code {
    List<Todo> toDoList;
    Todo taskObject = new Todo();

    protected override async Task OnInitializedAsync()
```

```

    {
        toDoList = await service.Get();
    }
    private void InitializeTaskObject()
    {
        taskObject = new ToDo();
        taskObject.DueDate = DateTime.Now;
    }

    private async void DataChanged()
    {
        toDoList = await service.Get();
        StateHasChanged();
    }
}

```

With this, we can add a new record after validation, close the dialog upon save and refresh the parent component.

Edit task

Let's see how to reuse the child component and the `ToDoListService` to edit an existing record. Okay, what are all the changes we need to make?

First, we need to wire up the "Edit" button to open the child component `TaskDetail.razor`. It also has to pass the selected task detail to the child component. I have modified the edit button to open the modal and call a method to set the selected object.

```

// ToDoList.razor

// Code omitted for brevity
<td><input type="button" class="btn btn-primary" @onclick="(() =>
PrepareForEdit(toDoItem))" data-toggle="modal" data-target="#taskModal"
value="Edit" /></td>

// ...

@code {
    List<ToDo> toDoList;
    ToDo taskObject = new ToDo();
    //...

    private void PrepareForEdit(ToDo task)
    {
        taskObject = task;
    }
}

```

The validations we have implemented to add new task will work automatically as we are using the same child component for Edit as well. Once the user modified the data and click save, `HandleValidSubmit` will be triggered. We have to distinguish between a new record

and an existing record to make the appropriate service call. I have used the `Id` property to differentiate the records.

```
// TaskDetail.razor

private async void HandleValidSubmit()
{
    if (TaskObject.Id == 0)
    {
        await service.Add(TaskObject);
    }
    else
    {
        await service.Update(TaskObject);
    }
    await CloseTaskModal();
    DataChanged?.Invoke();
}
```

Render Fragments – Dynamic Content

As of now, there is no difference between Add and Edit dialogs. The captions and the controls in both the dialogues are the same as we are reusing the dialog. The next requirement is to show some visual difference between the dialogs but the reusability of the child component should continue. A render fragment represents a segment of UI to render. We can pass dynamic content within component tag elements and can be rendered in child components using `RenderFragment` property. Okay, how to achieve that? First, declare the `RenderFragement` property in the child component and then substitute the title of the modal dialog with the property.

```
// TaskDetail.razor

//...
<h5 class="modal-title">@CustomHeader</h5>

//...
@code {
    //..

    [Parameter]
    public RenderFragment CustomHeader { get; set; }

    //..
}
```

Now in the parent component, we pass the dynamic content within the child component tags. The dynamic content value is set in the add and edit methods. I just pass the inner text as a fragment here, you can pass an HTML element or a nested HTML element.

```
// ToDoList.razor
```

```
//..
<TaskDetail TaskObject=taskObject DataChanged="@DataChanged">
    <CustomHeader>@customHeader</CustomHeader>
</TaskDetail>

@code {
    List<ToDo> toDoList;
    ToDo taskObject = new ToDo();
    string customHeader = string.Empty;

    //..

    private void InitializeTaskObject()
    {
        taskObject = new ToDo();
        taskObject.DueDate = DateTime.Now;
        customHeader = "Add New Task";
    }

    private void PrepareForEdit(ToDo task)
    {
        customHeader = "Edit Task";
        taskObject = task;
    }

    // ..
}
```

See how the caption of the dialog changed between add and edit dialogs.

Delete Task

We have implemented add and edit functionalities with validation. Let's start work on delete now. Though we cannot reuse the child component we already created, we can still use the JavaScript Interop. First, we need to wire up the delete button with some event. That event will set the selected task object and pass the information to the child component. Upon confirmation from the child component for delete, we need to delete the record and refresh the task list.

```
//ToDoList.razor

// Code omitted for brevity
<td><input type="button" class="btn btn-danger" @onclick="(() =>
PrepareForDelete(todoItem))" data-toggle="modal" data-
target="#confirmDeleteModal" value="Delete" /></td>

//..
<ConfirmDialog OnClick="@Delete" />
<TaskDetail TaskObject=taskObject DataChanged="@DataChanged">
    <CustomHeader>@customHeader</CustomHeader>
</TaskDetail>

@code {
    List<ToDo> todoList;
    ToDo taskObject = new ToDo();
    string customHeader = string.Empty;

    //..

    private void PrepareForDelete(ToDo task)
    {
        taskObject = task;
    }

    private async Task Delete()
    {
        var task = await service.Delete(taskObject.Id);
        await jsRuntime.InvokeAsync<object>("CloseModal", "confirmDeleteModal");
        todoList = await service.Get();
        taskObject = new ToDo();
    }
}
```

Next, we need to create a child component (`ConfirmDialog.razor`) to get the confirmation from the user on the Delete button click. This is again a bootstrap modal dialog.

```
<div class="modal" tabindex="-1" role="dialog" id="confirmDeleteModal">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title">Confirmation</h5>
                <button type="button" class="close" data-dismiss="modal"
aria-label="Close">
                    <span aria-hidden="true">x</span>
                </button>
            </div>
            <div class="modal-body">
                <p>Do you want to delete the record?</p>
            </div>
        </div>
    </div>
</div>
```

```

        <div class="modal-footer">
            <button type="button" class="btn btn-primary"
@onclick="OnClick">Yes</button>
            <button type="button" class="btn btn-secondary" data-
dismiss="modal">Cancel</button>
        </div>
    </div>
</div>
</div>

@code {
    [Parameter]
    public int Id { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClick { get; set; }
}

```

Summary

This post tried to explain about using Entity Framework Core with Blazor and implement CRUD functionalities. We also discussed the usage of Bootstrap, Validations using Data Annotations and Creation of Dynamic content using RenderFragment.

The full implementation of this post will be available in [Github](#)