BLAZOR WEBASSEMBLY

SUCCINCTLY

BY MICHAEL WASHINGTON

Blazor WebAssembly Succinctly

By

Michael Washington

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author
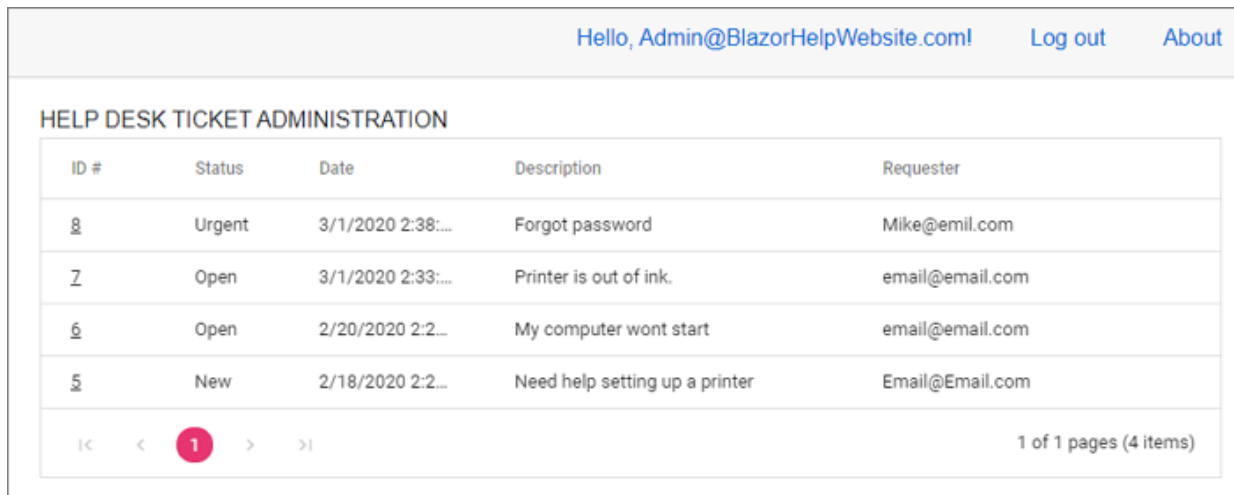
Michael Washington is a Microsoft MVP, an ASP.NET C# programmer, and the founder of BlazorHelpWebsite.com. He is the author of the book *An Introduction to Building Applications with Blazor*. He has extensive knowledge in process improvement, billing systems, and student information systems. He has a son, Zachary, and resides in Los Angeles with his wife, Valerie.

You can follow him on Twitter at @ADefWebserver.

Introduction

Blazor is an exciting technology that allows you to create web-based applications using C# instead of JavaScript. However, you still have the ability to implement custom JavaScript when you desire. Blazor WebAssembly is a component that allows browsers to run binary code for enhanced performance and security.

Blazor is an alternative to other single-page application (SPA) frameworks such as Angular, React, and Vue.js.

Figure 1: Help Desk Administration

In this book, we will cover the core elements of Blazor, and then explore additional features by building a sample application called Syncfusion Help Desk.

This application will demonstrate the following:

- Implementing authentication and authorization.
- Inserting, updating, and deleting data from the database.
- Using forms and validation.
- Implementing email notifications.

The code for this e-book is available on GitHub. The step-by-step instructions use Visual Studio 2019 Community edition, which is available for free at this link. In addition, SQL Server Developer edition is recommended, and it is available for download for free at this link.

You may want to bookmark the official Microsoft Blazor documentation available at https://blazor.net.

Chapter 1 What is Blazor?

Blazor applications are composed of components that are constructed using C#, HTML-based Razor syntax, and CSS.

Blazor has two different runtime modes: server-side Blazor and client-side Blazor, also known as Blazor WebAssembly. Both modes run in all modern web browsers, including web browsers on mobile phones.

Server-side Blazor
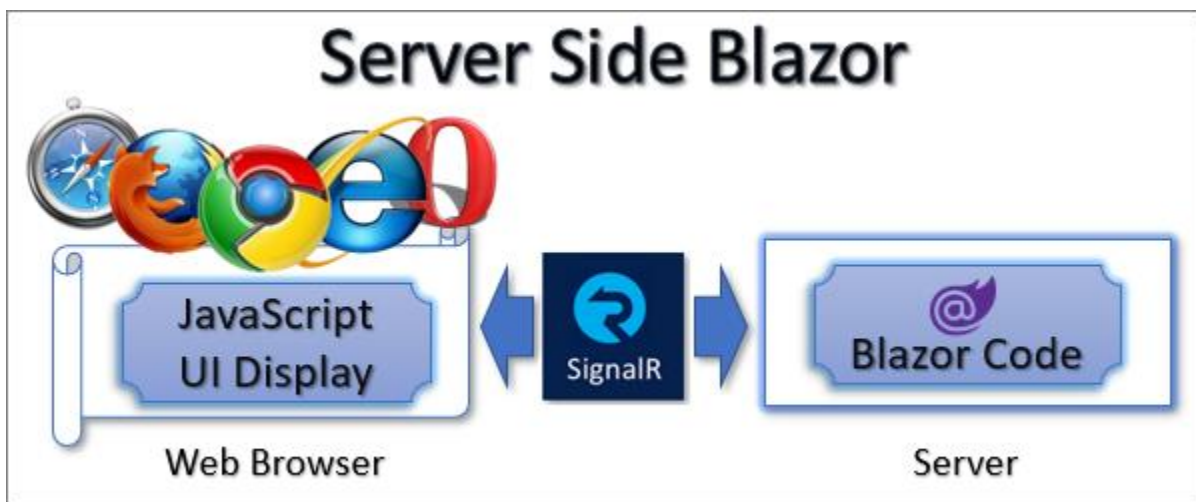
Figure 2: Server-Side Blazor
© BlazorHelpWebsite.com, used with permission

Server-side Blazor renders the Razor components on the server and updates the webpage using a SignalR connection. The Blazor framework sends events from the web browser, such as button clicks and mouse movements, to the server. The Blazor runtime computes changes to the components on the server and sends a *diff-based* webpage back to the browser.

Client-side Blazor (WebAssembly)

Figure 3: Client-Side Blazor (WebAssembly)
© BlazorHelpWebsite.com, used with permission

Client-side Blazor is composed of the same code as server-side Blazor; however, it runs entirely in the web browser using a technology known as WebAssembly.

The primary difference in Blazor applications that are created in server-side Blazor versus client-side Blazor is that the client-side Blazor applications need to make web calls to access server data, whereas the server-side Blazor applications can omit this step as all their code is executed on the server.

One way to think of Blazor is that Blazor is a framework for creating SPA webpages using one of two architectures (client side or server side), using Razor technology written with the C# language.

Because client-side Blazor with WebAssembly executes entirely on a user's browser, it is very fast for many applications.

The Syncfusion Help Desk sample application covered in this book will be built using client-side Blazor.

Core Blazor features

Components and routing

A Blazor application is composed of components. A component is a chunk of code consisting of a user interface and processing logic. A Blazor component is also called a Razor component.

Blazor features routing, where you can provide navigation to your controls using the **@page** directive followed by a unique route in quotes preceded by a slash.

The following is an example of a simple Razor component called **ComponentExample.razor**.

Code Listing 1: ComponentExample.razor

```
@page "/componentexample"

<h3>This is Component Example</h3>

@code {

}
```

The following shows what the component looks like in a running application.

Figure 4: A Simple Component

A Razor component is contained in a .razor file and can be nested inside of other components.

For example, we can create a component named **ComponentOne.razor** using the following code.

Code Listing 2: ComponentOne.razor

```
<h4 style="background-color:goldenrod">
    This is ComponentOne
</h4>

@code {

}
```

We can alter **ComponentExample.razor** to contain **ComponentOne.razor**.

Code Listing 3: ComponentExample.razor

```
@page "/componentexample"

<h3>This is Component Example</h3>
```

```
<ComponentOne />

@code {

}
```

The following shows what **ComponentExample.razor** now looks like when running in the application.
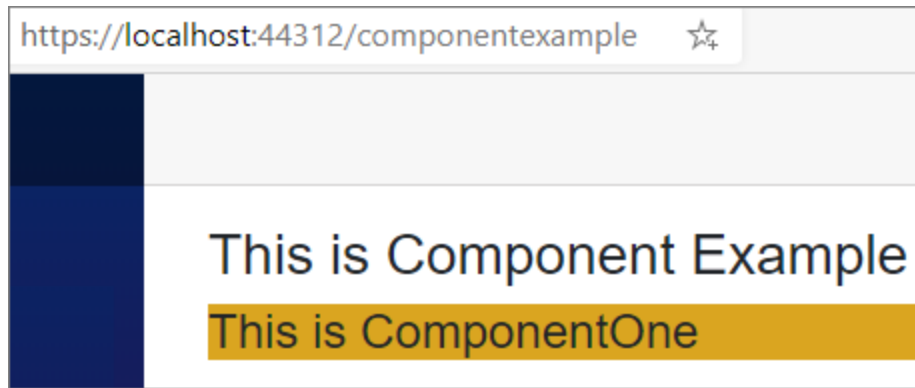
Figure 5: Nested Component

Note: A component's name must start with an uppercase character.

Parameters

Razor components can pass values to other components using parameters. Component parameters are defined using the **[Parameter]** attribute, which must be declared as public.

For example, we can create a Razor component called **ParameterExampleComponent.razor** that contains a parameter called **Title**.

Code Listing 4: ParameterExampleComponent.razor

```
<h4>Parameter Example Component</h4>

<h5 style="color:red">@Title</h5>

@code {
    [Parameter]
    public string Title { get; set; }
}
```

We create another Razor component called **ParameterExample.razor** that consumes the **ParameterExampleComponent.razor** control and passes a value (**Passed from Parent)** to the **Title** parameter in the **ParameterExampleComponent.razor** control.

```razor
@page "/parameterexample"

<h4>Parameter Example</h4>

<ParameterExampleComponent Title="Passed from Parent" />

@code {

}
```

When we run the application, we get the following result.
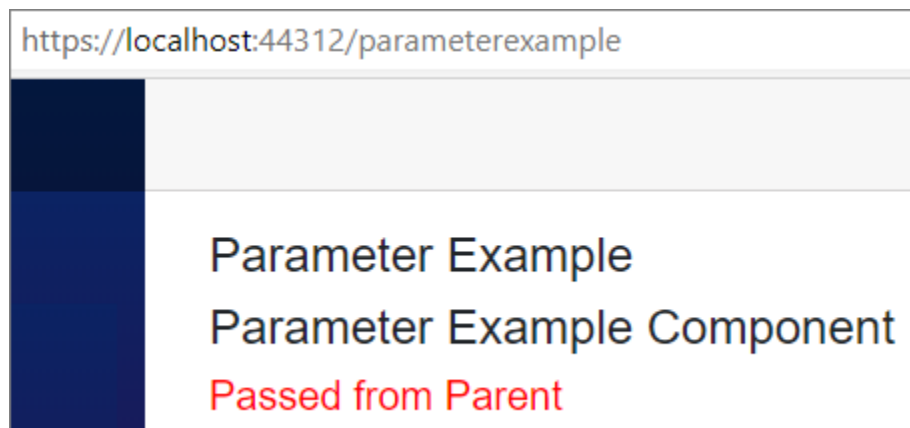
Figure 6: Parameter Example

Data binding

Simple, one-way binding in Blazor is achieved by declaring a parameter and referencing it using the **@** symbol. An example of this is shown in the following code.

Code Listing 6: One-Way Binding

```razor
<b>BoundValue:</b> @BoundValue

@code {
    private string BoundValue { get; set; }

    protected override void OnInitialized()
    {
        BoundValue = "Initial Value";
    }
}
```

This displays the following when rendered.

BoundValue: Initial Value
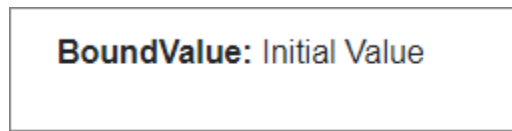
Figure 7: One-Way Binding

Two-way, dynamic data binding in Razor components is implemented using the **@bind** attribute.

The following example demonstrates this.

Code Listing 7: Two-Way Binding

```
<input @bind="BoundValue" @bind:event="oninput" />

<p>Display CurrentValue: @BoundValue</p>

@code {
    private string BoundValue { get; set; }
}
```

When we run the code, it displays the value entered into the text input box as text is typed into the input box.

abc123
Display CurrentValue: abc123
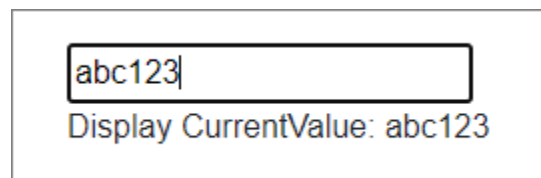
Figure 8: Two-Way Binding

Events

Raising events in Razor components is straightforward. The following example demonstrates using the **@onclick** event handler to execute the method **IncrementCount** when the button is clicked.

Code Listing 8: Simple Event

```
<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">
    Click me
</button>
```

```
@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

When the control is rendered and the button is clicked six times, the UI looks like this.
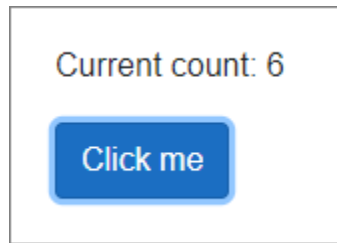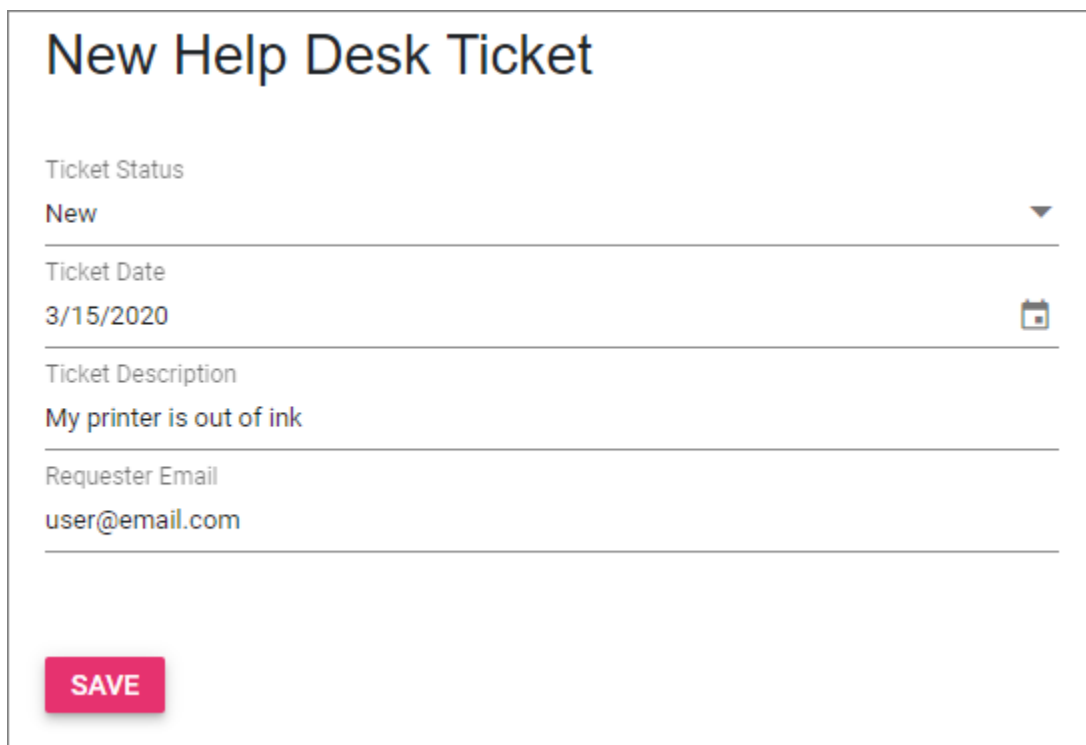
Figure 9: Simple Event

Chapter 2 The Help Desk Application

To demonstrate the features of Blazor, and how controls such as the suite available from Syncfusion can make development faster and easier, we will create a simple help desk application.

Help desk users

Figure 10: New Help Desk Ticket

Users of the application will see a form that allows them to create a new help desk ticket.
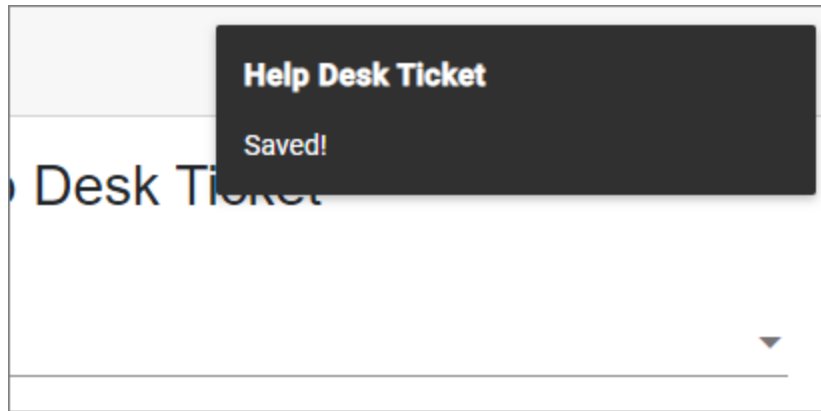
Figure 11: Syncfusion Toast

With the Syncfusion **Toast** control, a pop-up is displayed to let the user know that their action was successful.
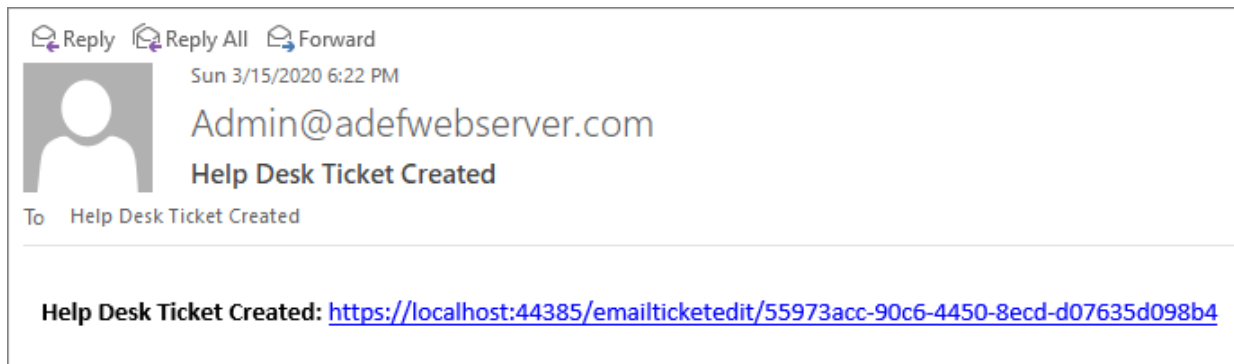
Figure 12: Email Notification

An email is sent to the administrator with a link that will navigate directly to the help desk ticket.

Figure 13: Administrator Adding Details

When the administrator is logged in and they click on the link in the email, they will have the ability to edit all fields and enter help desk ticket detail records at the bottom of the form by entering the desired text and clicking **Add**.

Figure 14: Administrator Saving Details

After adding the desired help desk ticket details, the administrator clicks **Save** to save the record and send a notification email to the user who created the help desk ticket.

Figure 15: User Adding Details

The user who created the help desk ticket receives an email with a link that navigates them to the help desk ticket and allows them to also enter details.

However, the fields at the top of the help desk ticket are grayed-out and disabled for them. They can only add new details.

Help desk administrators
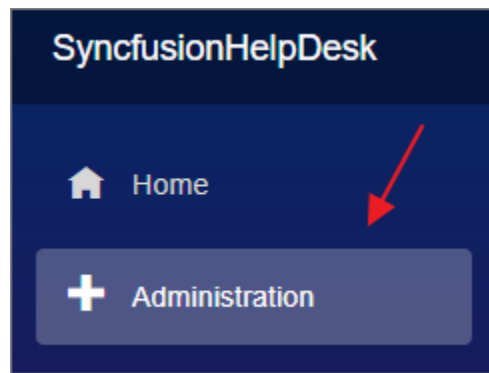
Figure 16: Administration Menu Link

A user logged in as the administrator will see an Administration link that will take them to the section of the application that will allow them to administer all help desk tickets.
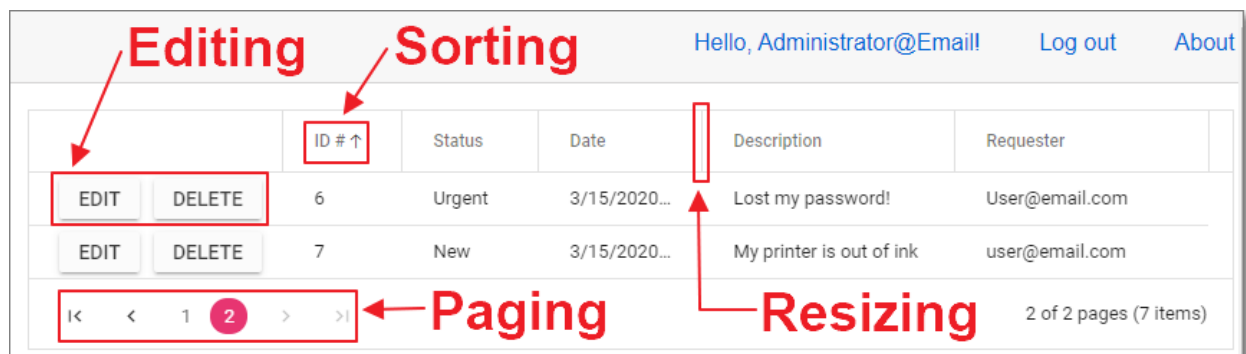
Figure 17: Syncfusion Data Grid

This will display the Syncfusion Data Grid. The administrator can edit records, sort, page, and resize the data grid.

Figure 18: Syncfusion Dialog

Clicking the Edit button next to a record in the data grid will open it up in the Syncfusion `Dialog` control.

This dialog will allow the administrator to edit all fields of the help desk ticket, as well as add help desk ticket detail records at the bottom of the form.

When the administrator saves the record, the user who created the help desk ticket receives an email with a link that navigates them to the help desk ticket.
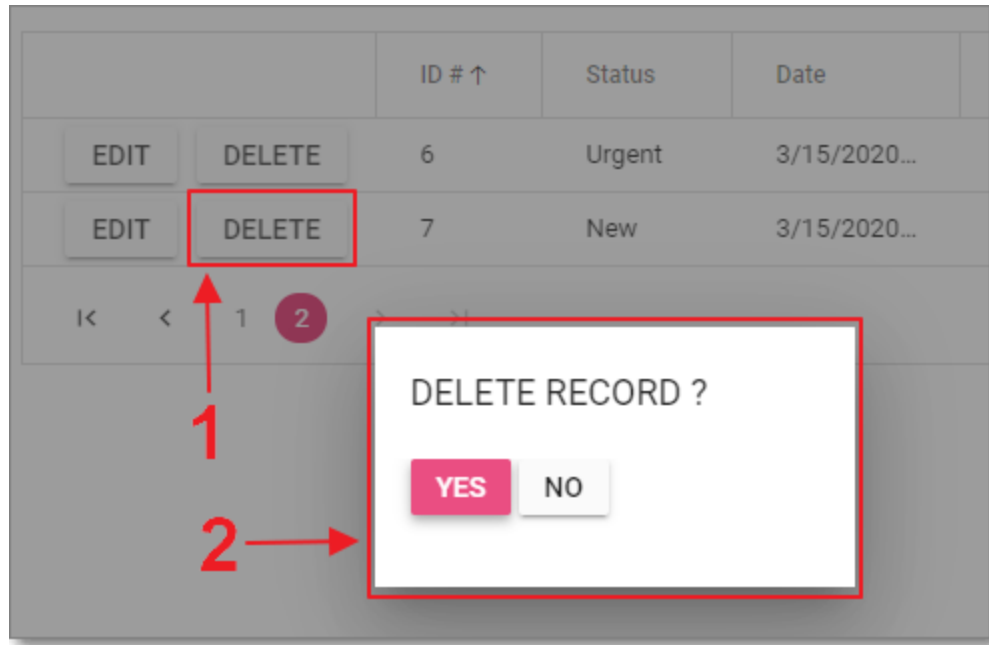
Figure 19: Delete Confirmation

Clicking the **Delete** button next to a record in the Data Grid will open the delete confirmation pop-up in the Syncfusion Dialog.

Clicking **Yes** will delete the record and clicking **No** will cancel the action.

Chapter 3 Create the Help Desk Application

In this chapter, we will cover the steps to create the Help Desk application.

Figure 20: The Help Desk Application in Action

The source code for the completed application is available on GitHub.

Install SQL Server
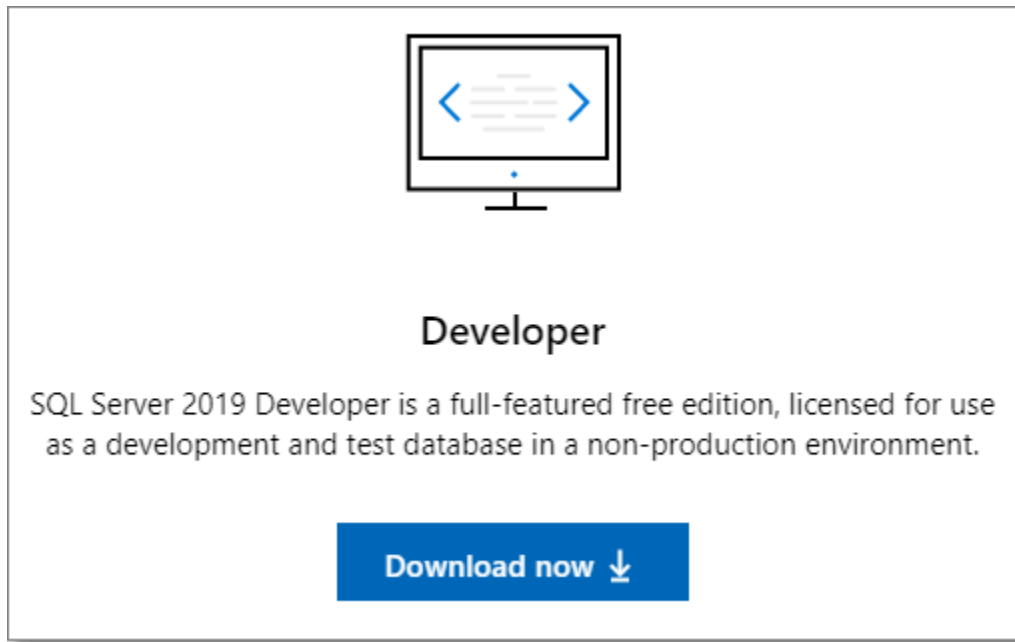
Figure 21: SQL Server

The application requires a database to store the data. Download and install the free SQL Server 2019 Developer edition here, or use the full SQL Server, if you have access to it.

Install .NET Core and Visual Studio

To create the application, these steps are required (if you do not already have the following software installed):

- Install the .NET Core 3.1 SDK.
- Install Visual Studio 2019 version 16.6 (or later) with the ASP.NET and web development workload here.

Note: The requirements for creating applications using Blazor are constantly evolving. See the latest requirements.

Note: If you install Visual Studio 2019 and select the .NET Core workload during installation, the .NET Core SDK and runtime will be installed for you. Learn more.

Create the project
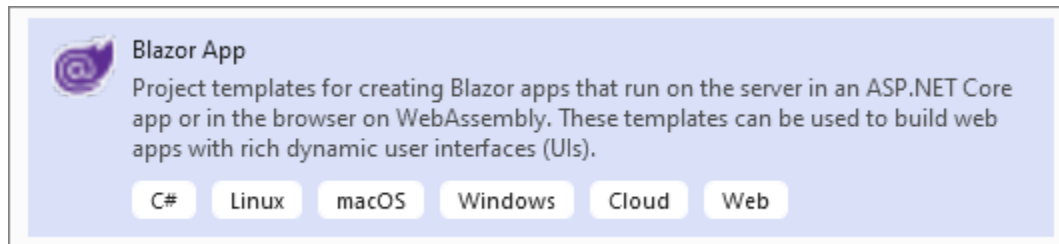
Figure 22: Blazor App

Open Visual Studio and select **Create a New Project** > **Blazor App** > **Next**. Enter **SyncfusionHelpDeskClient** for Project name and click **Create**.
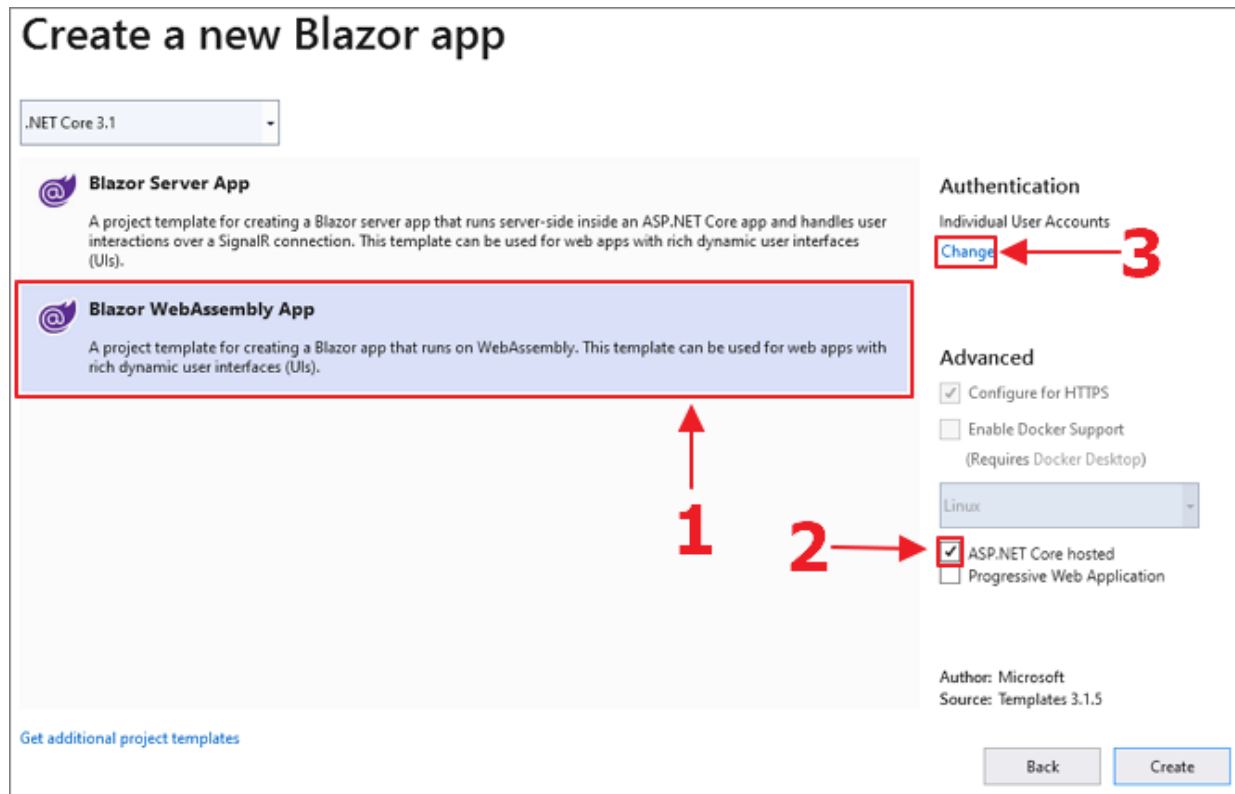
Figure 23: Change Authentication

On the Create a new Blazor app dialog, select **Blazor WebAssembly App** and **ASP.NET Core hosted** and click the **Change** link under Authentication.
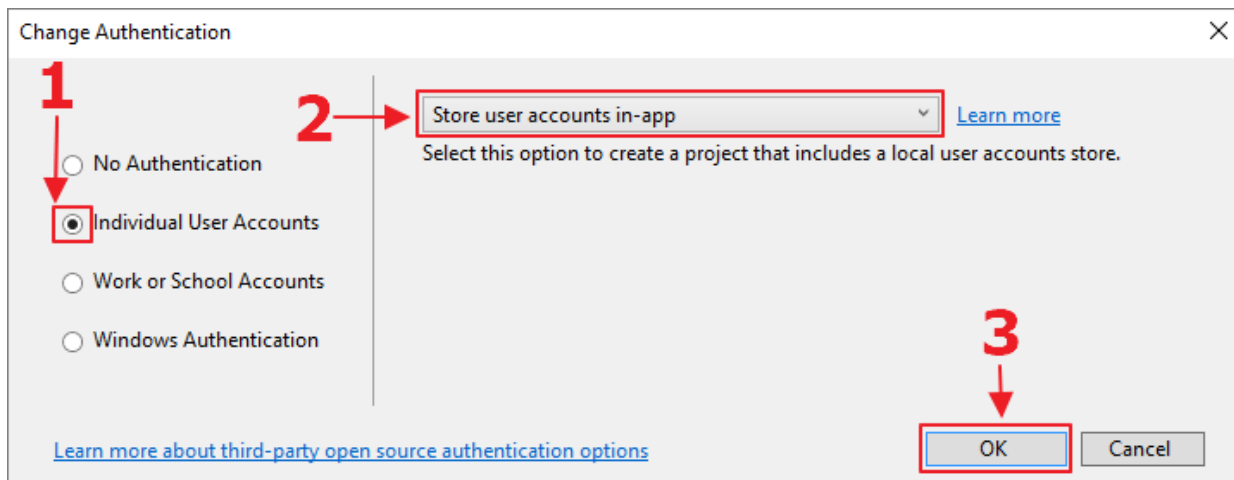
Figure 24: Set Authentication

In the Change Authentication dialog, select **Individual User Accounts** and **Store user accounts in-app**.

Click **OK**, and on the Create a new Blazor app dialog, click **Create**.
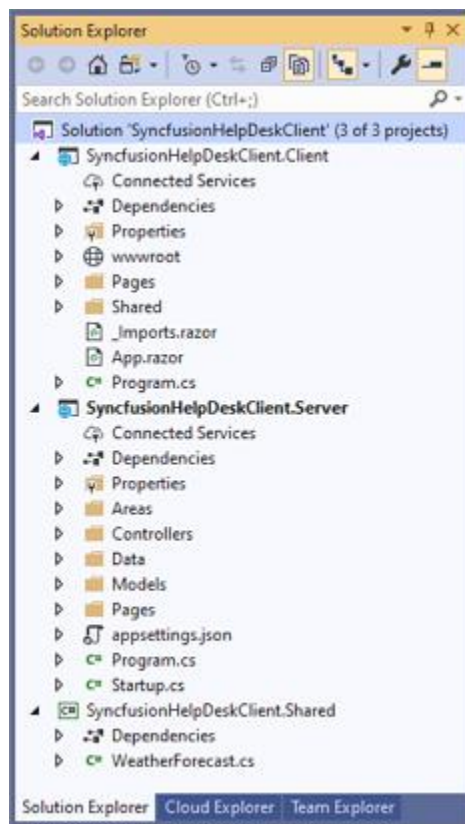
Figure 25: In Visual Studio

The project will be created and will open in Visual Studio.

Create the database

From the toolbar, click **View** and select the **SQL Server Object Explorer**.
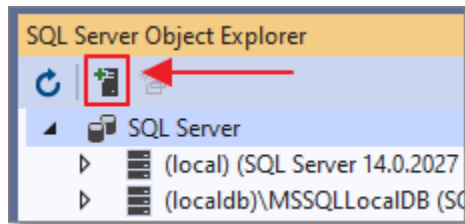
Figure 26: Add SQL Server

Click **Add SQL Server** to add a connection to your database server, if you don't already have it in the SQL Server list.

📋 *Note: For this example, we do not want to use the localdb connection (for SQL Express) that you may see in the list.*
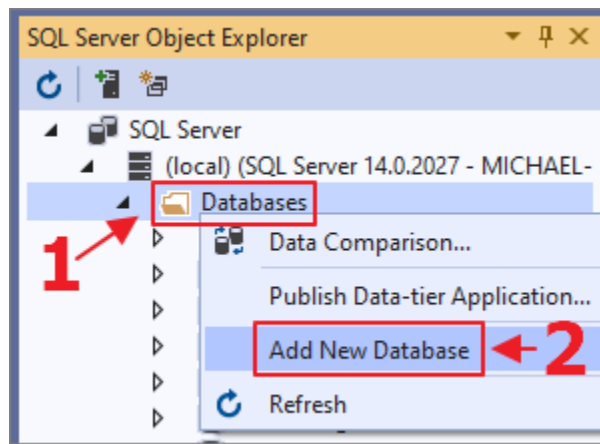
Figure 27: Add New Database

Expand the tree node for your SQL Server, then right-click **Databases** and select **Add New Database**. Name the database **SyncfusionHelpDesk**.
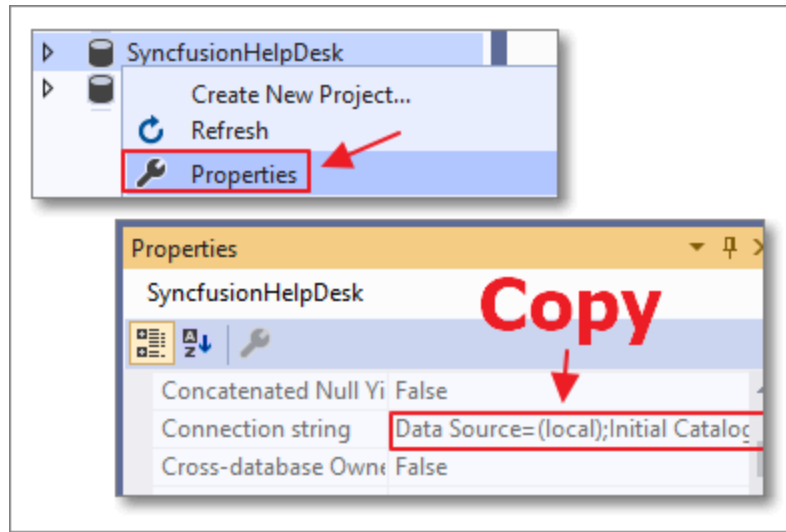
Figure 28: Copy Connection String

After the database has been created, right-click it and select **Properties**. In the Properties window, copy the connection string.
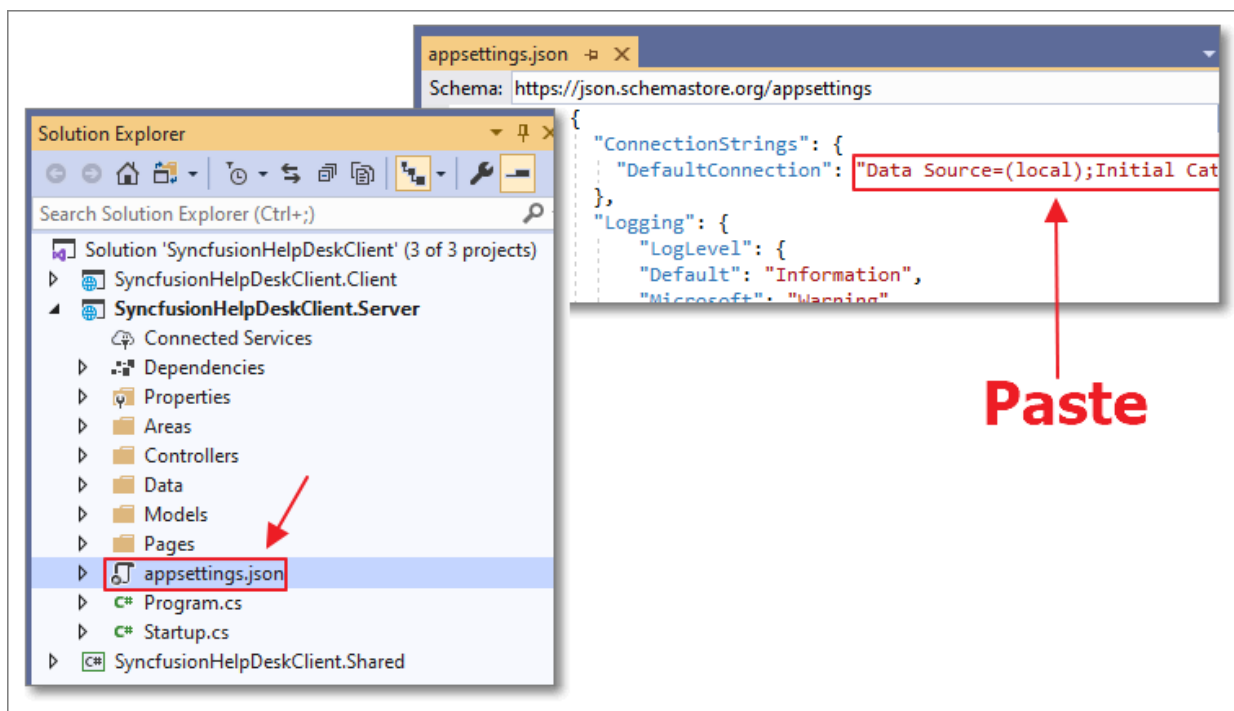
Figure 29: Paste Connection String

Open the appsettings.json file in the Server project and paste the connection string in the **DefaultConnection** property.

Save and close the file.

Blazor WebAssembly security

The default code the Visual Studio wizard creates will allow us to create new users. However, we want some users to be administrators. To do this, we must enable role management.

Open the **Startup.cs** file in the **Server** project and remove the following code

Code Listing 9: Original Identity Code

```
    services.AddDefaultIdentity<IdentityUser>(
        options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();

            services.AddIdentityServer()
                .AddApiAuthorization<ApplicationUser,
ApplicationDbContext>();
```

Replace the code with the following code to remove the requirement to confirm new user accounts and to enable role management.

Code Listing 10: Updated Identity Code

```
    services.AddDefaultIdentity<ApplicationUser>()
                .AddRoles<IdentityRole>() // Add roles.
                .AddEntityFrameworkStores<ApplicationDbContext>();

            // From:
https://github.com/dotnet/AspNetCore.Docs/issues/17649
            // Configure identity server to put the role claim into the
id token
            // and the access token and prevent the default mapping for
roles
            // in the JwtSecurityTokenHandler.
            services.AddIdentityServer()
                .AddApiAuthorization<ApplicationUser,
ApplicationDbContext>(options =>
                {
options.IdentityResources["openid"].UserClaims.Add("role");
                    options.ApiResources.Single().UserClaims.Add("role");
                });

            // Need to do this as it maps "role" to ClaimTypes.Role and
causes issues.
            System.IdentityModel.Tokens.Jwt.JwtSecurityTokenHandler
                .DefaultInboundClaimTypeMap.Remove("role");
```

In Visual Studio, press **F5** to run the application and open it in your web browser.

Figure 30: Register User

Click **Register** and create a user with the email Test@Email.

You will then see the following page.

Figure 31: Apply Migrations

Click **Apply Migrations**. After a few moments, you will see a message to refresh the page. Refresh the page in your web browser.
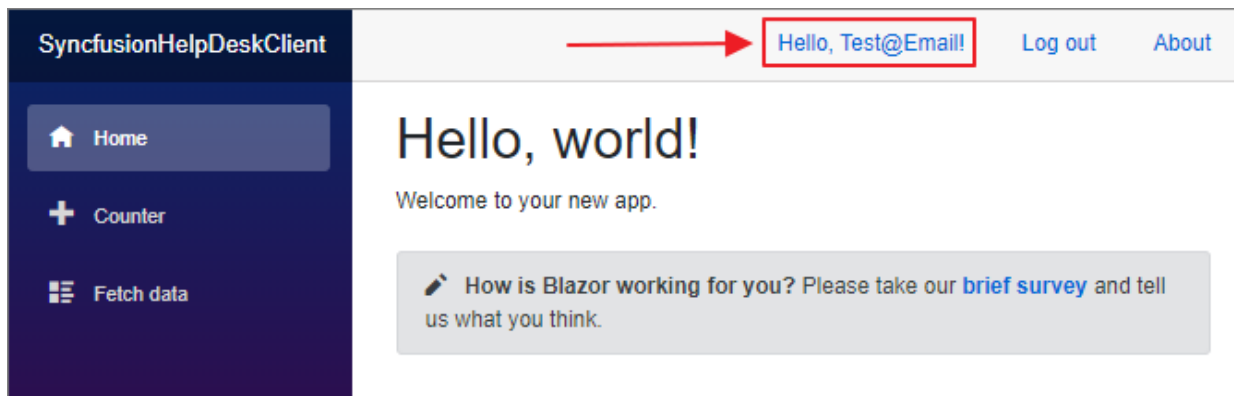
Figure 32: Logged In

The required tables will be created in the database and the application will display the home page and indicate that the **Test@Email** account is logged in.

Close the web browser to stop the application and return to Visual Studio.

Create an administrator

We will now create code that will programmatically create an administrator role and add the **Administrator@Email** account to the administrator role.
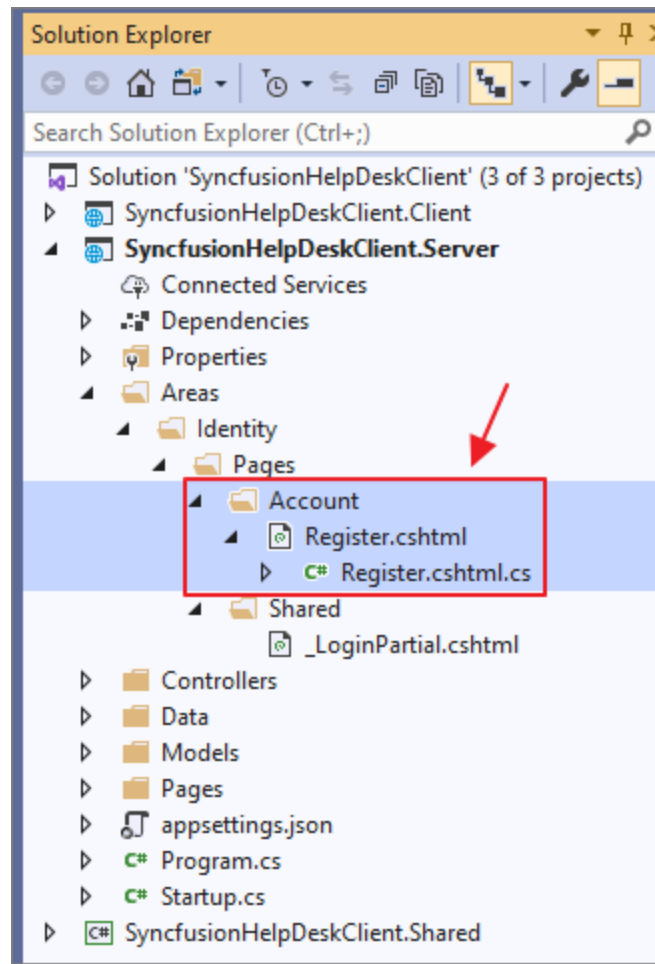
Figure 33: Add Custom Register Page

To do this, we need to override the registration page with a custom registration page.

In the **Server** project, create an **Account** folder in the **Areas/Identity/Pages** folder. Create a **Register.cshtml** page and a **Register.cshtml.cs** page, using the following code.

Code Listing 11: Register.cshtml

```
@page
@model RegisterModel
@{
    ViewData["Title"] = "Register";
}

<h1>@ViewData["Title"]</h1>

<div class="row">
    <div class="col-md-4">
        <form asp-route-returnUrl="@Model.ReturnUrl" method="post">
```

```html
            <h4>Create a new account.</h4>
            <hr />
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Input.Email"></label>
                <input asp-for="Input.Email" class="form-control" />
                <span asp-validation-for="Input.Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.Password"></label>
                <input asp-for="Input.Password" class="form-control" />
                <span asp-validation-for="Input.Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.ConfirmPassword"></label>
                <input asp-for="Input.ConfirmPassword" class="form-control" />
                <span asp-validation-for="Input.ConfirmPassword" class="text-danger"></span>
            </div>
            <button type="submit" class="btn btn-primary">Register</button>
        </form>
    </div>
    <div class="col-md-6 col-md-offset-2">

    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Text.Encodings.Web;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

using SyncfusionHelpDeskClient.Server.Models;

namespace SyncfusionHelpDeskClient.Server.Areas.Identity.Pages
{
    [AllowAnonymous]
    public class RegisterModel : PageModel
    {
        const string ADMINISTRATION_ROLE = "Administrators";
        const string ADMINISTRATOR_USERNAME = "Admin@email";

        private readonly SignInManager<ApplicationUser> _signInManager;
        private readonly UserManager<ApplicationUser> _userManager;
        private readonly RoleManager<IdentityRole> _roleManager;

        public RegisterModel(
            SignInManager<ApplicationUser> signInManager,
            UserManager<ApplicationUser> userManager,
            RoleManager<IdentityRole> roleManager)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _roleManager = roleManager;
        }

        [BindProperty]
        public InputModel Input { get; set; }

        public string ReturnUrl { get; set; }

        public IList<AuthenticationScheme> ExternalLogins { get; set; }

        public class InputModel
```

```csharp
        {
            [Required]
            [EmailAddress]
            [Display(Name = "Email")]
            public string Email { get; set; }

            [Required]
            [StringLength(100, ErrorMessage =
                "The {0} must be at least {2} and at max {1} characters
long.",
                MinimumLength = 6)]
            [DataType(DataType.Password)]
            [Display(Name = "Password")]
            public string Password { get; set; }

            [DataType(DataType.Password)]
            [Display(Name = "Confirm password")]
            [Compare("Password", ErrorMessage =
                "The password and confirmation password do not match.")]
            public string ConfirmPassword { get; set; }
        }

        public async Task OnGetAsync(string returnUrl = null)
        {
            ReturnUrl = returnUrl;
            ExternalLogins =
                (await _signInManager
                .GetExternalAuthenticationSchemesAsync())
                .ToList();
        }

        public async Task<IActionResult> OnPostAsync(string returnUrl =
null)
        {
            returnUrl = returnUrl ?? Url.Content("~/");
            ExternalLogins =
                (await _signInManager
                .GetExternalAuthenticationSchemesAsync())
                .ToList();

            if (ModelState.IsValid)
            {
                var user = new ApplicationUser
                { UserName = Input.Email, Email = Input.Email };

                var result =
                    await _userManager.CreateAsync(user, Input.Password);

                if (result.Succeeded)
```

```csharp
                {
                    // Set confirm Email for user.
                    user.EmailConfirmed = true;
                    await _userManager.UpdateAsync(user);

                    // Ensure there is a ADMINISTRATION_ROLE
                    var RoleResult = await _roleManager
                        .FindByNameAsync(ADMINISTRATION_ROLE);

                    if (RoleResult == null)
                    {
                        // Create ADMINISTRATION_ROLE role.
                        await _roleManager
                            .CreateAsync(new
IdentityRole(ADMINISTRATION_ROLE));
                    }

                    if (user.UserName.ToLower() ==
                        ADMINISTRATOR_USERNAME.ToLower())
                    {
                        // Put admin in Administrator role.
                        await _userManager
                            .AddToRoleAsync(user, ADMINISTRATION_ROLE);
                    }

                    // Log user in.
                    await _signInManager.SignInAsync(user, isPersistent:
false);

                    return LocalRedirect(returnUrl);
                }
                foreach (var error in result.Errors)
                {
                    ModelState.AddModelError(string.Empty,
error.Description);
                }
            }

        // If we got this far, something failed, redisplay form.
        return Page();
    }
}

}
```
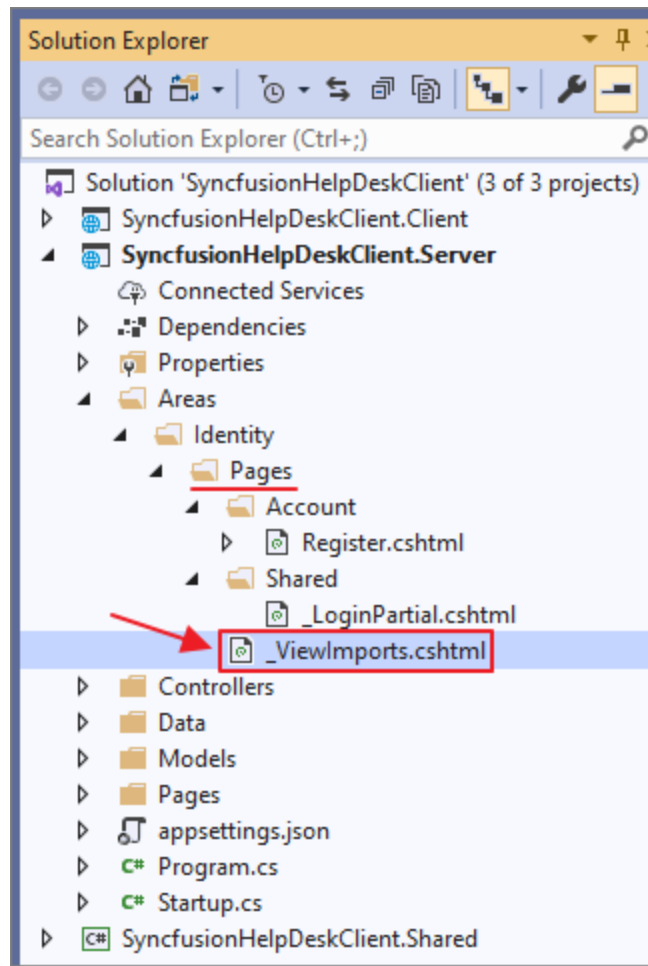
Figure 34: _ViewImports.cshtml

Next, add a _ViewImports.cshtml file in the Pages folder using the following code.

Code Listing 13: _ViewImports.cshtml

```
@using Microsoft.AspNetCore.Identity
@using SyncfusionHelpDeskClient.Server.Areas.Identity
@using SyncfusionHelpDeskClient.Server.Areas.Identity.Pages

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Finally, to test that we can restrict access to a user in the **Administrators** role, in the Server project, remove the following code in the Controllers\WeatherForecastController.cs file.

Code Listing 14: Remove Authorize

```
[Authorize]
```

Replace the code with the following.

```
[Authorize(Roles = "Administrators")] // Only allow Administrators.
```

📝 **Note: We must always enforce security in the server project because it is possible to bypass the client project and call code in the server project directly. Security in the client project (implemented later) is only for visual representation; it cannot enforce security.**

Even though security cannot be enforced in the client project, it is still helpful to implement code that will display differently according to the roles of the current user.

We will also implement code in the client project that will indicate if the current user is in the administrator role.
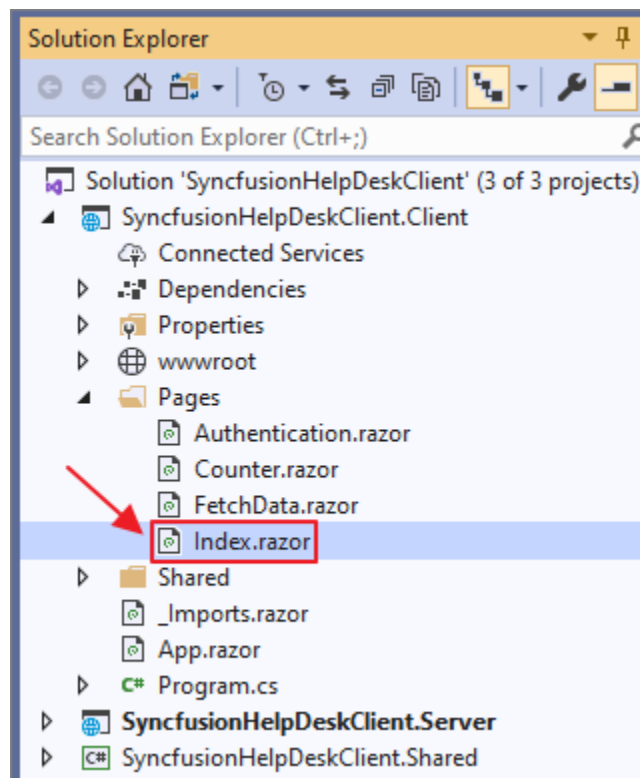
Figure 35: Update Index.razor

Open the **Index.razor** page in the **Client** project, and replace all the code with the following code.

Code Listing 16: Detect an Administrator

```
@page "/"

<AuthorizeView Roles="@ADMINISTRATION_ROLE">
```

```
    <p>You are an Administrator</p>
</AuthorizeView>

@code {
    string ADMINISTRATION_ROLE = "Administrators";

}
```
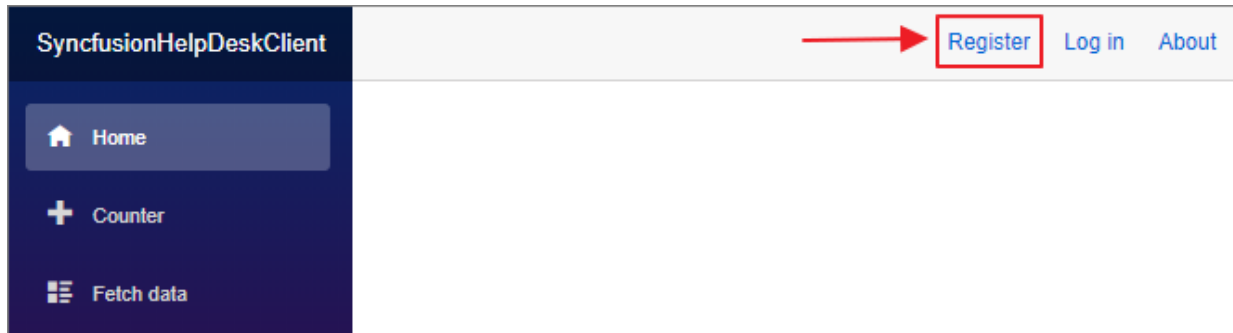

Figure 36: Register Administrator

Run the application and click **Register**.

Figure 37: Create Admin@Email

Create a new user with the email Admin@Email.
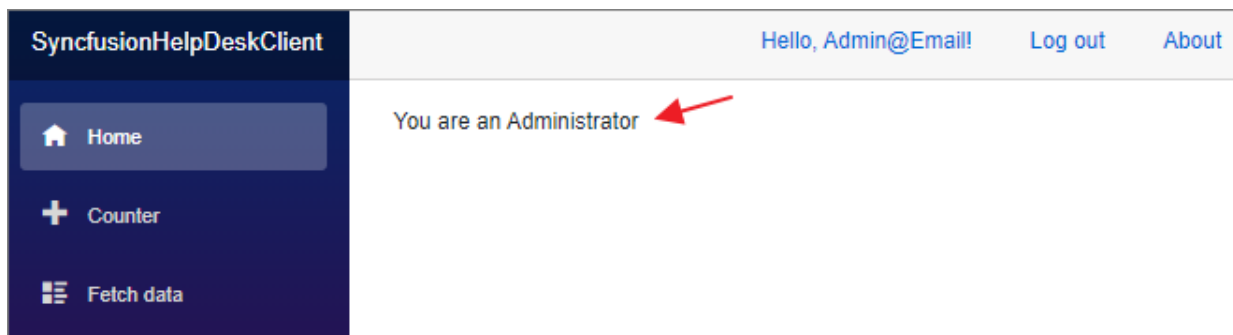
Figure 38: Administrator Display

The home page of the application will indicate that the user is an administrator.
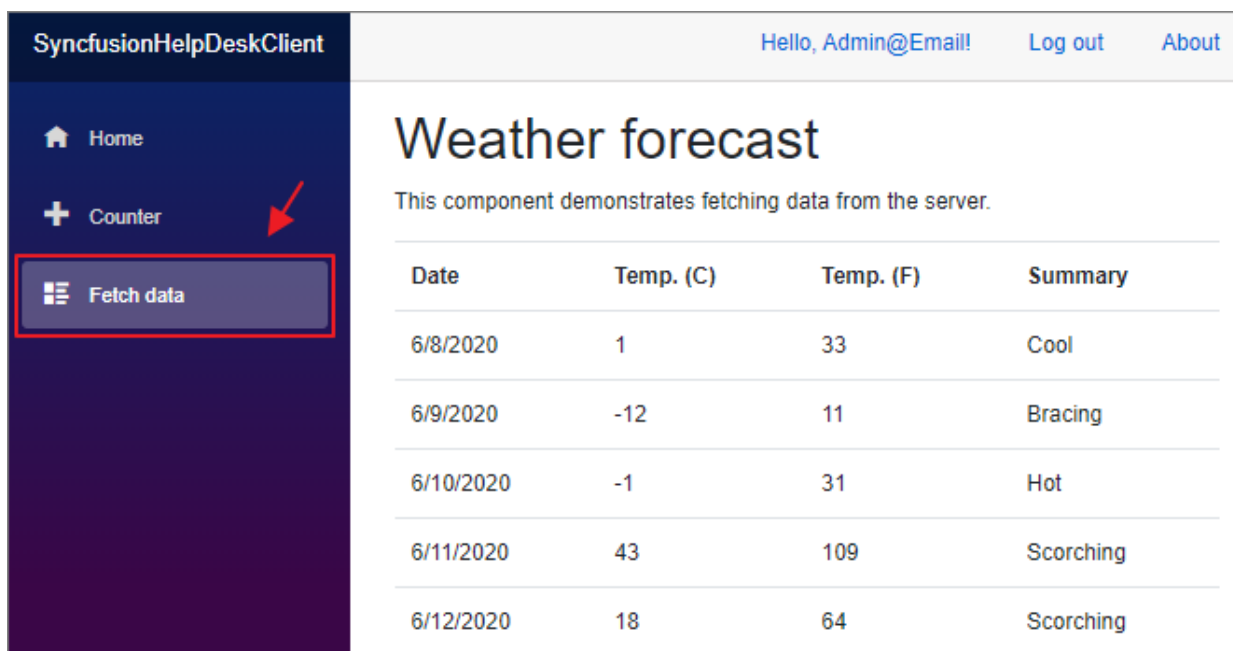
Figure 39: Fetch Data Page

The Fetch data page will also work.

When we log in, the code in the Index.razor page runs to create the administrator role and to add the Administrator@Email account to that role.

Unauthenticated users

Sometimes we want to allow unauthenticated users to call server-side controllers.

To demonstrate this, in the **Server** project, remove the following code in the **Controllers\WeatherForecastController.cs** file.

Code Listing 17: Remove Administrators

```
[Authorize(Roles = "Administrators")] // Only allow Administrators.
```

Replace the code with the following.

Code Listing 18: Allow All Users

```
[AllowAnonymous] // Allow everyone.
```

However, when we run the application and log in as a user who is not an administrator (**Test@Email**), we will be directed to the login page.

This happens because the Fetch data page is marked as *authorized*, and the default **HttpClient**, which makes the calls to the server-side code, requires the user to be authenticated.

To rectify this, we will need to create a second **HttpClient** that does not require authentication.
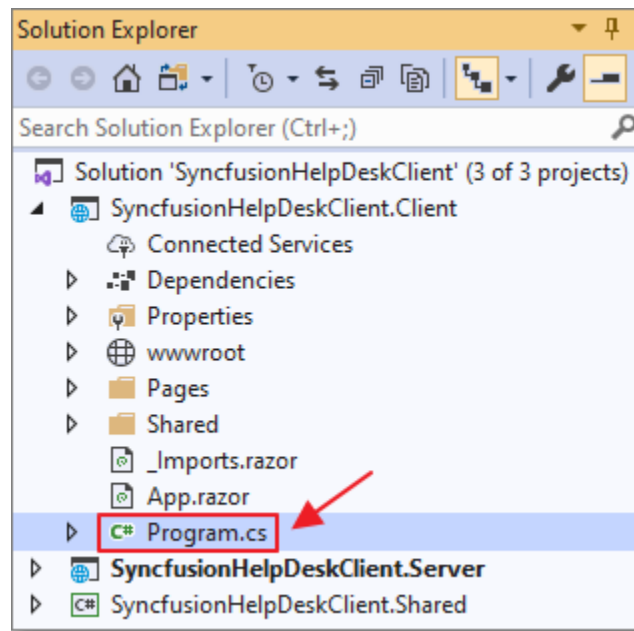
Figure 40: Update Program.cs

In the Client project, open the Program.cs file and insert the following code before **builder.Services.AddApiAuthorization()**.

```
    // This allows anonymous requests.
    // See: https://bit.ly/2Y3ET3K
    builder.Services.AddHttpClient("ServerAPI.NoAuthenticationClient",
        client => client.BaseAddress = new
Uri(builder.HostEnvironment.BaseAddress));
```
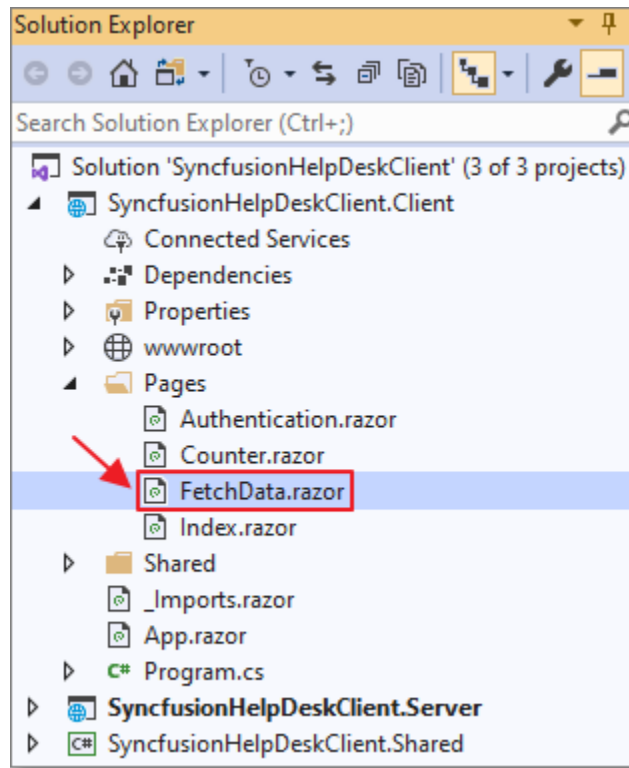

Figure 41: Update FetchData.razor

Next, open the **FetchData.razor** page and remove the following code, which requires users to be logged in to view the page.

Code Listing 20: Remove [Authorize]

```
@attribute [Authorize]
```

Add the code with the following that allows a *named* **HttpClient** to be used.

Code Listing 21: Add ClientFactory

```
@inject IHttpClientFactory ClientFactory
```

Finally, delete the following code.

Code Listing 22: Remove Default HttpClient

```
forecasts =
    await Http.GetFromJsonAsync<WeatherForecast[]>

    ("WeatherForecast");
```

Replace it with the following code that uses the new *named* **HttpClient**.

Code Listing 23: Add NoAuthenticationClient

```
// Use the NoAuthenticationClient
var client =
    ClientFactory.CreateClient("ServerAPI.NoAuthenticationClient");

forecasts =
    await client.GetFromJsonAsync<WeatherForecast[]>

    ("WeatherForecast");
```
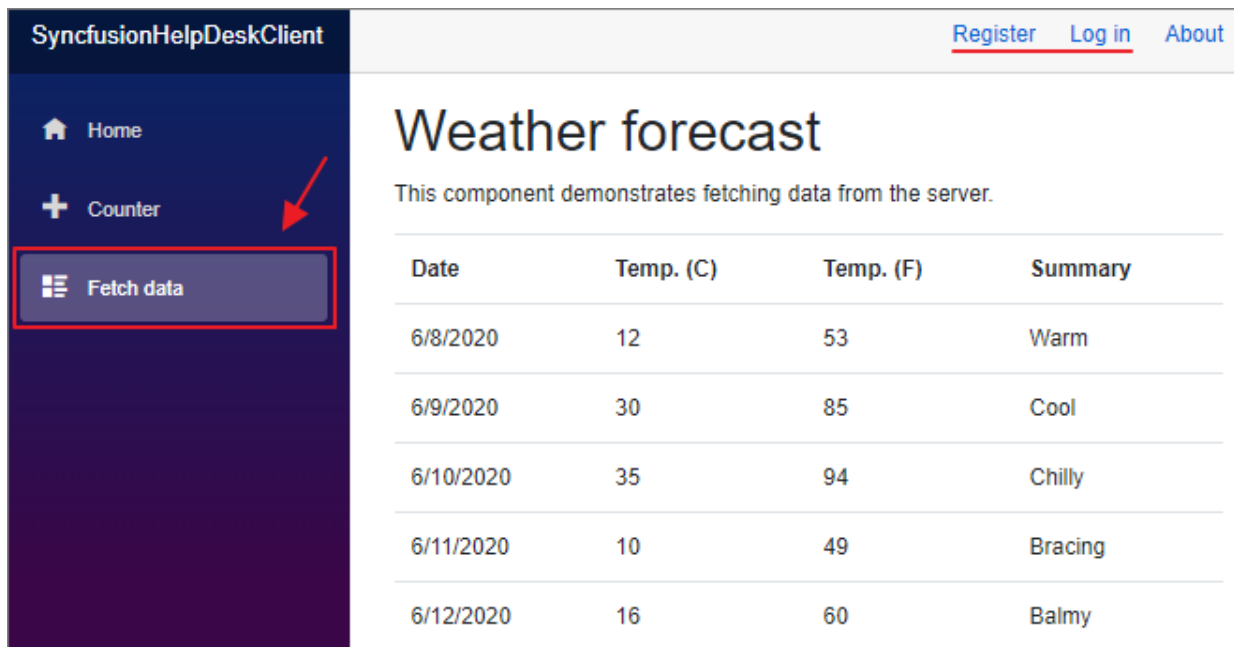

Figure 42: Page Works for Unauthenticated Users

When we run the application, the Fetch data page now works for unauthenticated users.

Chapter 4 Explore the Project

In this chapter, we will explore the Blazor project we just created.
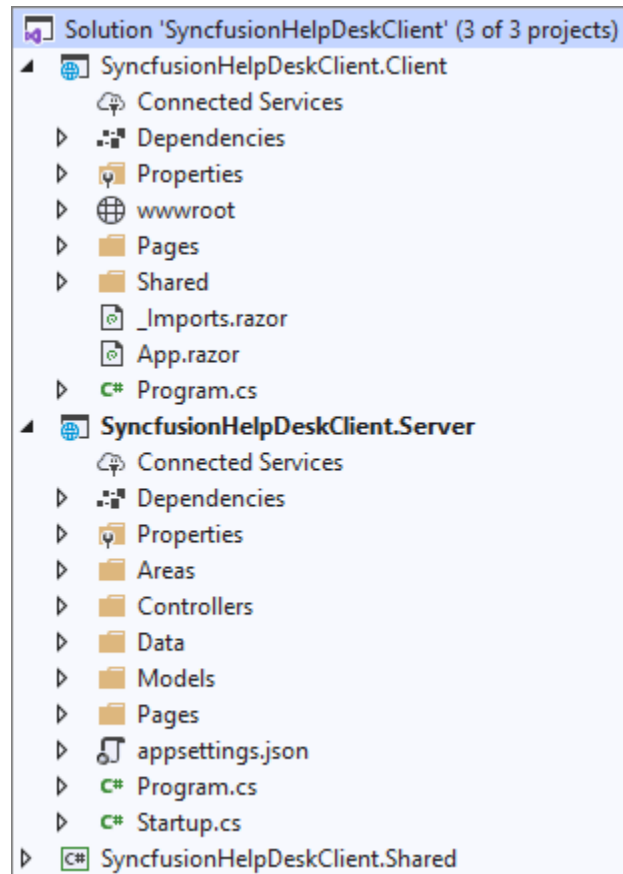
Figure 43: Explore the Project

Startup

The Program.cs file in the Server project is the entry point for the application and invokes the **Startup** class that is defined in the Startup.cs file.

Code Listing 24: Program.cs

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
```

```
            });
```

The **Startup** class in the Server project calls the method **MapFallbackToFile("index.html")** that sets up the index.html page, in the Client project, as the root page of the application.

Code Listing 25: Startup.cs

```
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
                endpoints.MapControllers();
                endpoints.MapFallbackToFile("index.html");
            });
```

The Program.cs file in the Client project specifies that the **App** component contained in the App.razor file is to be rendered as the root component of the application.

Code Listing 26: Program Class

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        builder.RootComponents.Add<App>("app");

        ...

        await builder.Build().RunAsync();
    }
}
```

Routing

The routing of page requests in the application is configured in the App.razor file, like this.

Code Listing 27: App.razor

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    @if (!context.User.Identity.IsAuthenticated)
                    {
```

```
                    <RedirectToLogin />
                }
                else
                {
                    <p>You are not authorized to access this
resource.</p>
                }
            </NotAuthorized>
        </AuthorizeRouteView>
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
  </Router>
</CascadingAuthenticationState>
```

This component intercepts web browser navigation and renders the page and the layout, after applying any configured authorization rules.

Layouts

The **App.razor** control specifies **MainLayout** (contained in the MainLayout.razor file) as the application's default layout component.

Code Listing 28: MainLayout.razor

```
@inherits LayoutComponentBase

<div class="sidebar">
    <NavMenu />
</div>

<div class="main">
    <div class="top-row px-4 auth">
        <LoginDisplay />
        <a href="https://docs.microsoft.com/aspnet/"
            target="_blank">About</a>
    </div>

    <div class="content px-4">
        @Body
    </div>
</div>
```

The **MainLayout** component inherits from **LayoutComponentBase** and will inject the content of the Razor page that the user is navigating to, at the location of the **@Body** parameter in the template.

The remaining page markup, including the **NavMenu** control (located in the NavMenu.razor file), and the **LoginDisplay** control (located in the LoginDisplay.razor file), will be displayed around the content, creating a consistent page layout.

Chapter 5 Add Syncfusion

The Syncfusion controls allow us to easily implement advanced functionality with a minimum amount of code. The Syncfusion controls are contained in a NuGet package. In this chapter we will cover the steps to obtain that package and configure it for our application.

Note: See the latest system requirements for using Syncfusion here.

Install NuGet packages
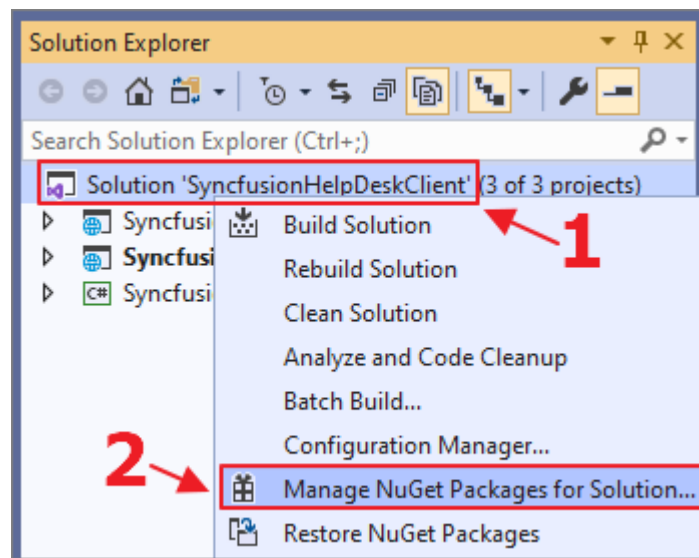
Figure 44: NuGet Packages

Right-click the **Solution** node and select **Manage NuGet Packages for Solution**.
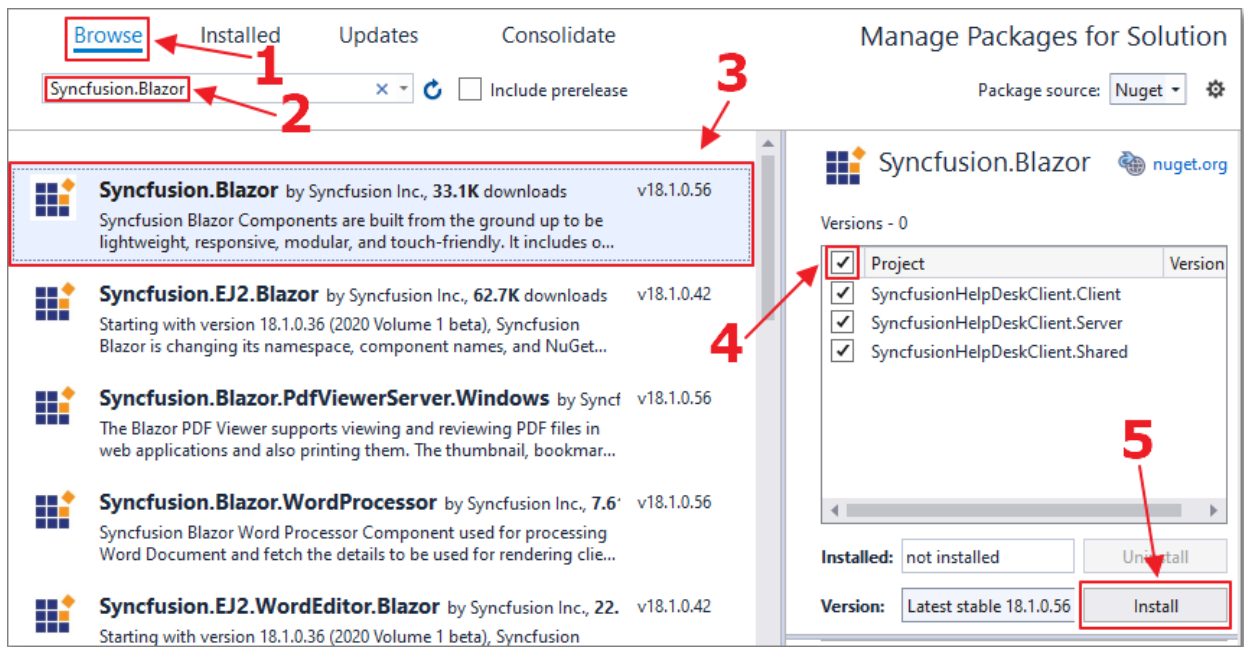
Figure 45: Install NuGet Packages

Select the **Browse** tab and install the following NuGet packages:

- Syncfusion.Blazor (in the Client, Server, and Shared projects)
- Microsoft.AspNetCore.Mvc.NewtonsoftJson (in the Server project)

Additional configuration

Open the **_Imports.razor** file and add the following.

Code Listing 29: _Imports.razor

```
@using Syncfusion.Blazor
@using Syncfusion.Blazor.Inputs
@using Syncfusion.Blazor.Popups
@using Syncfusion.Blazor.Data
@using Syncfusion.Blazor.DropDowns
@using Syncfusion.Blazor.Layouts
@using Syncfusion.Blazor.Calendars
@using Syncfusion.Blazor.Navigations
@using Syncfusion.Blazor.Lists
@using Syncfusion.Blazor.Grids
@using Syncfusion.Blazor.Buttons
@using Syncfusion.Blazor.Notifications
```

Open the Program.cs file in the Client project and add the following **using** statement.

```
using Syncfusion.Blazor;
```

Add the following code to the **Main** method, before the **builder.Build().RunAsync();** line.

Code Listing 31: Startup.cs AddSyncfusionBlazor

```
        // Syncfusion support
        builder.Services.AddSyncfusionBlazor(true);
```

Finally, add the following to the **<head>** element of the wwwroot/index.html page in the Client project.

Code Listing 32: Index.html

```
    <link href="_content/Syncfusion.Blazor/styles/material.css"
        rel="stylesheet" />
    <script src="_content/Syncfusion.Blazor/scripts/syncfusion-blazor.min.js"
        type="text/javascript"></script>
```

Serialization

The help desk ticket object, which we will create in later steps, contains a nested collection of **HelpDeskTicketDetail** objects. We need to add special code that uses the Microsoft.AspNetCore.Mvc.NewtonsoftJson assembly (added earlier) to properly serialize and deserialize the objects.

In the **startup.cs** file in the **Server** project, delete the following code.

Code Listing 33: AddControllersWithViews Before

```
services.AddControllersWithViews();
```

Replace it with the following code.

Code Listing 34: Updated AddControllersWithViews

```
        services.AddControllersWithViews()
            .AddNewtonsoftJson(
            options =>
            options.SerializerSettings.ReferenceLoopHandling =
            Newtonsoft.Json.ReferenceLoopHandling.Ignore
            );
```

Note: Later, after we add Syncfusion controls to the project, you will see the following message when you run the application:

This application was built using a trial version of Syncfusion Essential Studio. Please include a valid license to permanently remove this license validation message. You can also obtain a free 30-day evaluation license to temporarily remove this message during the evaluation period. Please refer to this help topic for more information.

Click the link in the message for instructions on obtaining a key to make the message go away.

Chapter 6 Creating a Data Layer

We will now add additional tables to the database to support the custom code we plan to write. To allow our code to communicate with these tables, we require a data layer. This data layer will also allow us to organize and reuse code efficiently.

Create the database tables

The first step is to add the database tables we will need.

In the **SQL Server Object Explorer**, right-click the database and select **New Query**.
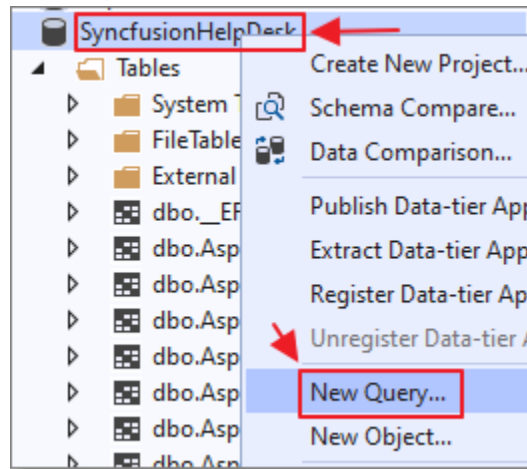
Figure 46: New Query

Enter the following script.

Code Listing 35: SQL Script

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[HelpDeskTicketDetails](
      [Id] [int] IDENTITY(1,1) NOT NULL,
      [HelpDeskTicketId] [int] NOT NULL,
      [TicketDetailDate] [datetime] NOT NULL,
      [TicketDescription] [nvarchar](max) NOT NULL,
 CONSTRAINT [PK_HelpDeskTicketDetails] PRIMARY KEY CLUSTERED
(
      [Id] ASC
```

```sql
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[HelpDeskTickets](
      [Id] [int] IDENTITY(1,1) NOT NULL,
      [TicketStatus] [nvarchar](50) NOT NULL,
      [TicketDate] [datetime] NOT NULL,
      [TicketDescription] [nvarchar](max) NOT NULL,
      [TicketRequesterEmail] [nvarchar](500) NOT NULL,
      [TicketGUID] [nvarchar](500) NOT NULL,
 CONSTRAINT [PK_HelpDeskTickets] PRIMARY KEY CLUSTERED
(
      [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
ALTER TABLE [dbo].[HelpDeskTicketDetails]
WITH CHECK
ADD  CONSTRAINT [FK_HelpDeskTicketDetails_HelpDeskTickets]
FOREIGN KEY([HelpDeskTicketId])
REFERENCES [dbo].[HelpDeskTickets] ([Id])
ON DELETE CASCADE
GO
ALTER TABLE [dbo].[HelpDeskTicketDetails]
CHECK CONSTRAINT [FK_HelpDeskTicketDetails_HelpDeskTickets]
GO
```

Click the Execute icon.
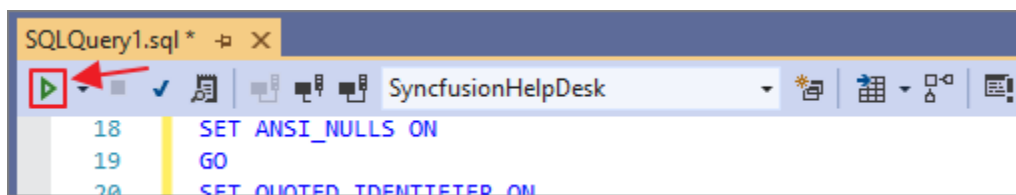
Figure 47: Click Execute

Close the SQLQuery1.sql window and refresh the view of the tables in the database.
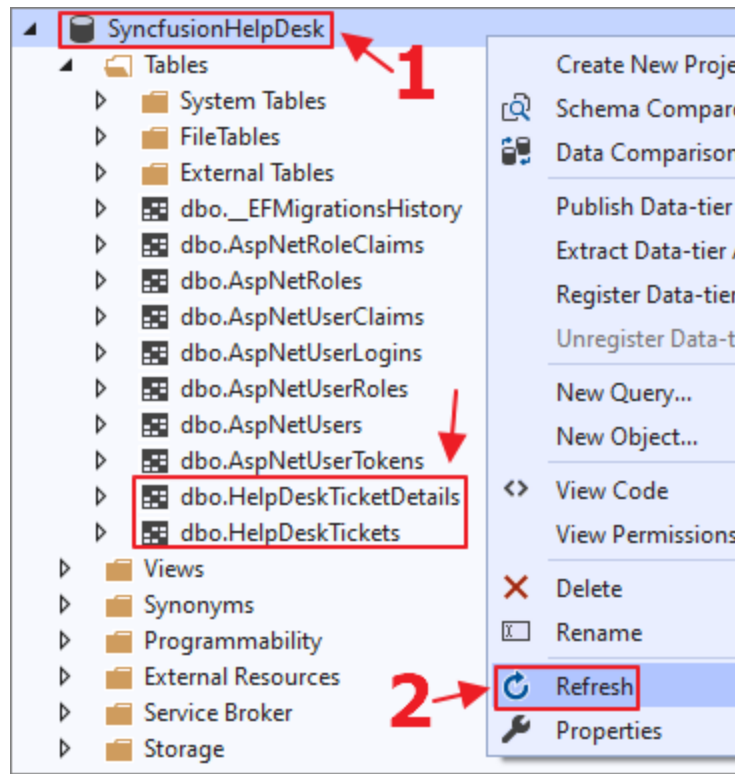
Figure 48: Refresh Database

You will see that the new tables have been added.
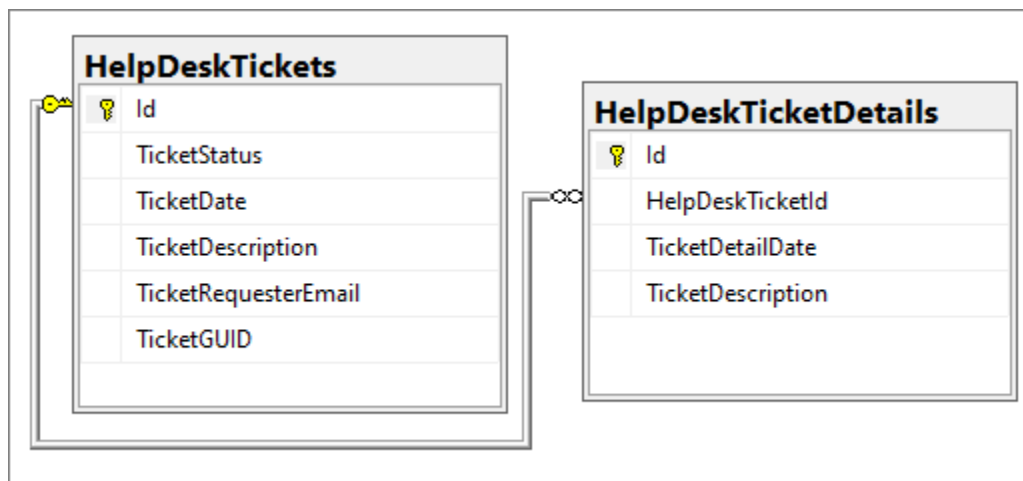
Figure 49: Database Diagram

The preceding diagram shows the relationship between the newly added tables. A HelpDeskTickets record is created first. As the issue is processed, multiple associated HelpSDeskTicketDetails records are added.

Create the DataContext (using EF Core Power Tools)

We will now create the DataContext code that will allow the server-side controller class, created in the following steps, to communicate with the database tables we just added.

Figure 50: EF Core Power Tools

Install EF Core Power Tools from the VS Marketplace.
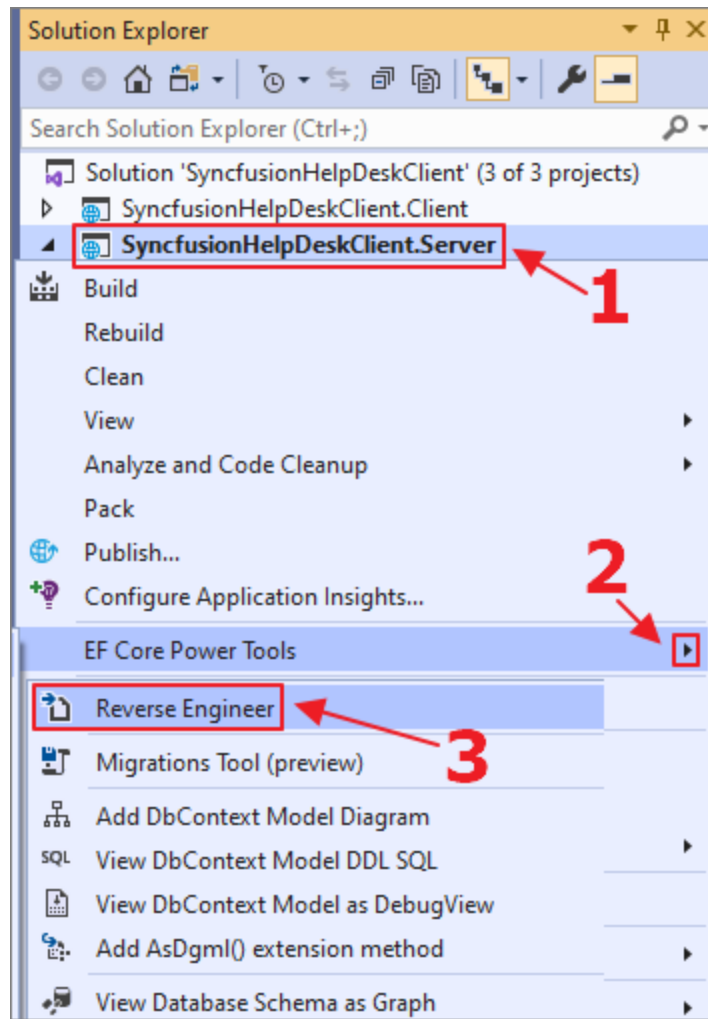
Figure 51: EF Core Power Tools

In the Server project, right-click the project node in the Solution Explorer and select **EF Core Power Tools** > **Reverse Engineer**.
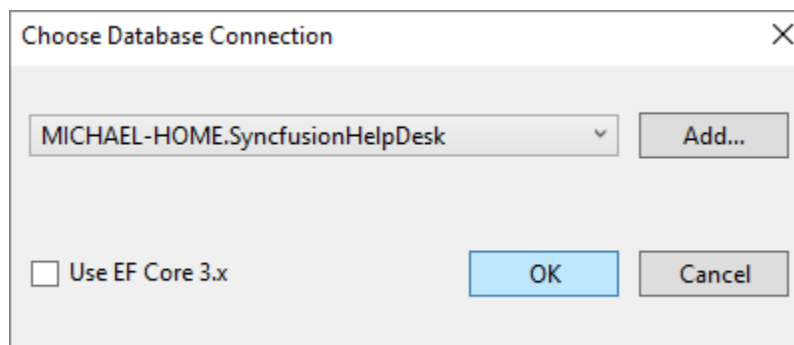
Figure 52: Create Connection

Click **Add** to create a connection to the database, if one does not already exist in the drop-down.

After you do that, or if the connection to the database is already in the drop-down, select the database connection in the drop-down and click **OK**.
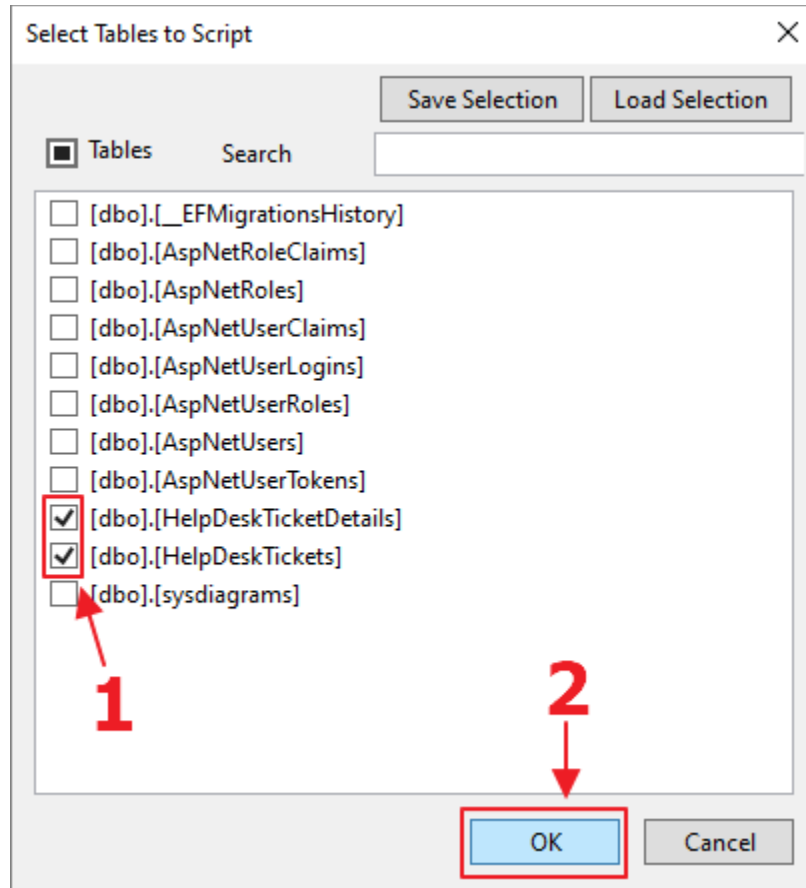
Figure 53: Select Tables

Select the **HelpDeskTicketDetails** table and the **HelpDeskTickets** table and click **OK**.
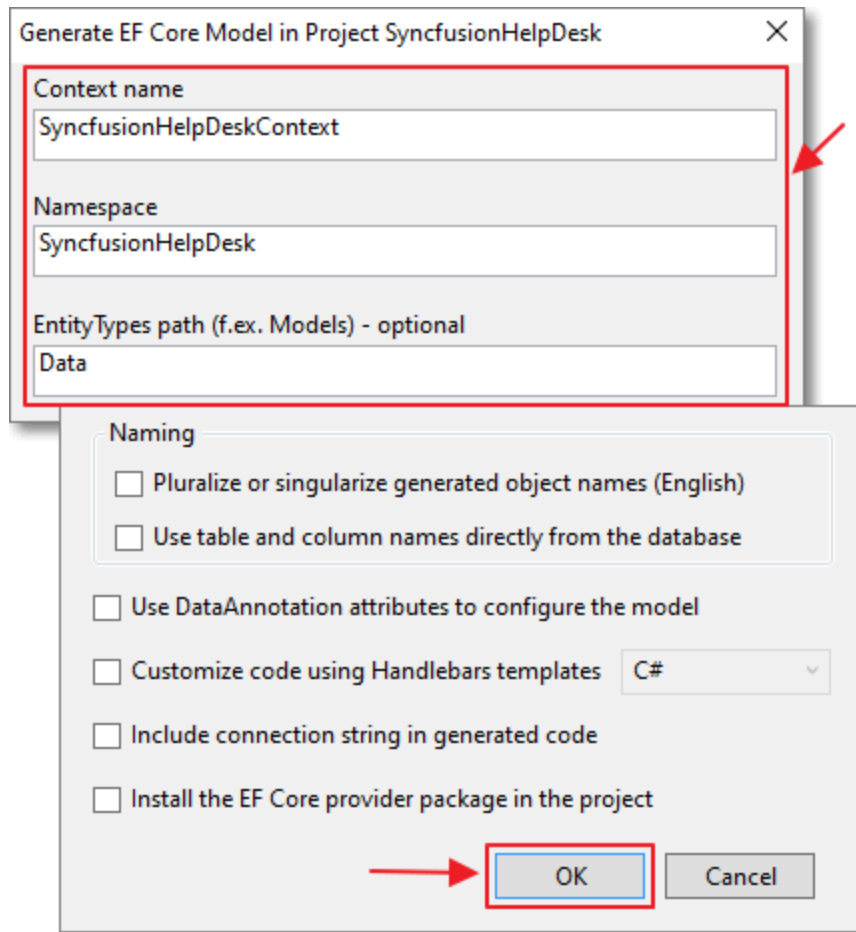
Figure 54: Configure DataContext

Set the following values:

- Context name: SyncfusionHelpDeskContext
- Namespace: SyncfusionHelpDesk
- EntityTypes path: Data

Click **OK**.
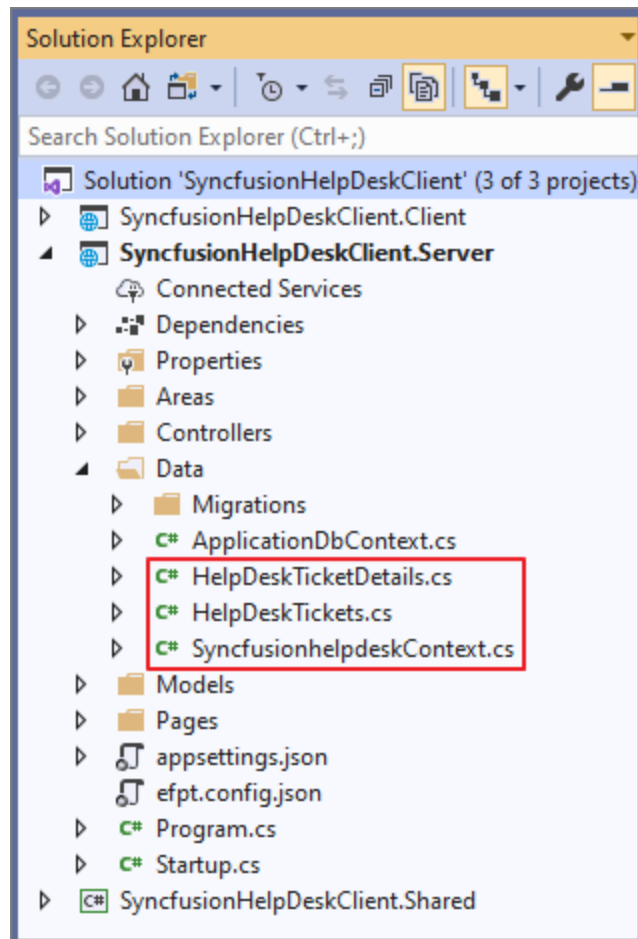
Figure 55: DataContext Created

In the Solution Explorer, you will see the DataContext has been created.

Set the database connection

The DataContext code needs the connection to the database to be set.

Open the Startup.cs file and add the following **using** statement.

Code Listing 36: Data Using Statement

```
using SyncfusionHelpDesk.Data;
```

Add the following code to the ConfigureServices section.

Code Listing 37: Database Connection

```
        // To access HelpDesk tables.
```

```
            services.AddDbContext<SyncfusionHelpDeskContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
```

Save the file. Select **Build** > **Rebuild Solution**.

The application should build without any errors.

Create the Syncfusion help desk controller

We will now create the server-side controller that will provide all the remaining data access methods we will need for the application. The code in the Client project, created in later steps, will call this code to communicate with the database.

In the **Controllers** folder of the Server project, add a new file called **SyncfusionHelpDeskController.cs** with the following code.

Code Listing 38: SyncfusionHelpDeskController.cs

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Primitives;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Threading.Tasks;

namespace SyncfusionHelpDesk.Data
{
    [ApiController]
    [Route("[controller]")]
    public class SyncfusionHelpDeskController : ControllerBase
    {
        private readonly SyncfusionHelpDeskContext _context;

        public SyncfusionHelpDeskController(
            SyncfusionHelpDeskContext context)
        {
            _context = context;
        }

        // Only an Administrator can query.
        [Authorize(Roles = "Administrators")]
        [HttpGet]
        public object Get()
```

```csharp
    {
        StringValues Skip;
        StringValues Take;
        StringValues OrderBy;

        // Filter the data.
        var TotalRecordCount = _context.HelpDeskTickets.Count();

        int skip = (Request.Query.TryGetValue("$skip", out Skip))
            ? Convert.ToInt32(Skip[0]) : 0;

        int top = (Request.Query.TryGetValue("$top", out Take))
            ? Convert.ToInt32(Take[0]) : TotalRecordCount;

        string orderby =
            (Request.Query.TryGetValue("$orderby", out OrderBy))
            ? OrderBy.ToString() : "TicketDate";

        // Handle OrderBy direction.
        if (orderby.EndsWith(" desc"))
        {
            orderby = orderby.Replace(" desc", "");

            return new
            {
                Items = _context.HelpDeskTickets
                .OrderByDescending(orderby)
                .Skip(skip)
                .Take(top),
                Count = TotalRecordCount
            };
        }
        else
        {
            System.Reflection.PropertyInfo prop =
                typeof(HelpDeskTickets).GetProperty(orderby);

            return new
            {
                Items = _context.HelpDeskTickets
                .OrderBy(orderby)
                .Skip(skip)
                .Take(top),
                Count = TotalRecordCount
            };
        }
    }

    [HttpPost]
```

```csharp
    [AllowAnonymous]
    public Task
        Post(HelpDeskTickets newHelpDeskTickets)
    {
        // Add a new Help Desk Ticket.
        _context.HelpDeskTickets.Add(newHelpDeskTickets);
        _context.SaveChanges();

        return Task.FromResult(newHelpDeskTickets);
    }

    [HttpPut]
    [AllowAnonymous]
    public Task
        PutAsync(HelpDeskTickets UpdatedHelpDeskTickets)
    {
        // Get the existing record.
        // Note: Caller must have the TicketGuid.
        var ExistingTicket =
            _context.HelpDeskTickets
            .Where(x => x.TicketGuid ==
            UpdatedHelpDeskTickets.TicketGuid)
            .FirstOrDefault();

        if (ExistingTicket != null)
        {
            ExistingTicket.TicketDate =
                UpdatedHelpDeskTickets.TicketDate;

            ExistingTicket.TicketDescription =
                UpdatedHelpDeskTickets.TicketDescription;

            ExistingTicket.TicketGuid =
                UpdatedHelpDeskTickets.TicketGuid;

            ExistingTicket.TicketRequesterEmail =
                UpdatedHelpDeskTickets.TicketRequesterEmail;

            ExistingTicket.TicketStatus =
                UpdatedHelpDeskTickets.TicketStatus;

            // Insert any new TicketDetails.
            if (UpdatedHelpDeskTickets.HelpDeskTicketDetails != null)
            {
                foreach (var item in
                    UpdatedHelpDeskTickets.HelpDeskTicketDetails)
                {
                    if (item.Id == 0)
                    {
```

```csharp
                        // Create new HelpDeskTicketDetails record.
                        HelpDeskTicketDetails newHelpDeskTicketDetails
=
                            new HelpDeskTicketDetails();
                        newHelpDeskTicketDetails.HelpDeskTicketId =
                            UpdatedHelpDeskTickets.Id;
                        newHelpDeskTicketDetails.TicketDetailDate =
                            DateTime.Now;
                        newHelpDeskTicketDetails.TicketDescription =
                            item.TicketDescription;

                        _context.HelpDeskTicketDetails
                            .Add(newHelpDeskTicketDetails);
                }
            }
        }

        _context.SaveChanges();
    }
    else
    {
        return Task.FromResult(false);
    }

    return Task.FromResult(true);
}

// Only an Administrator can delete.
[Authorize(Roles = "Administrators")]
[HttpDelete]
public Task
    Delete(
    string HelpDeskTicketGuid)
{
    // Get the existing record.
    var ExistingTicket =
        _context.HelpDeskTickets
        .Include(x => x.HelpDeskTicketDetails)
        .Where(x => x.TicketGuid == HelpDeskTicketGuid)
        .FirstOrDefault();

    if (ExistingTicket != null)
    {
        // Delete the Help Desk Ticket.
        _context.HelpDeskTickets.Remove(ExistingTicket);
        _context.SaveChanges();
    }
    else
    {
```

```
            return Task.FromResult(false);
        }

        return Task.FromResult(true);
    }
}

// From: https://bit.ly/30ypMCp
public static class IQueryableExtensions
{
    public static IOrderedQueryable<T> OrderBy<T>(
        this IQueryable<T> source, string propertyName)
    {
        return source.OrderBy(ToLambda<T>(propertyName));
    }

    public static IOrderedQueryable<T> OrderByDescending<T>(
        this IQueryable<T> source, string propertyName)
    {
        return source.OrderByDescending(ToLambda<T>(propertyName));
    }

    private static Expression<Func<T, object>> ToLambda<T>(
        string propertyName)
    {
        var parameter = Expression.Parameter(typeof(T));
        var property = Expression.Property(parameter, propertyName);
        var propAsObject = Expression.Convert(property,
typeof(object));

        return Expression.Lambda<Func<T, object>>(propAsObject,
parameter);
    }
}
}
```

Handling multiple database contexts

Finally, we need to adjust the **ApplicationDbContext** class to use the generic options type. This is required because we have added the **SyncfusionHelpDeskContext**, and without using the generic constructor, the wrong class may be initialized.

In the Server project, open the ApplicationDbContext.cs file in the Data folder, and delete this code.

Code Listing 39: Old ApplicationDbContext Code

```
        public ApplicationDbContext(
            DbContextOptions options,
            IOptions<OperationalStoreOptions> operationalStoreOptions) :
            base(options, operationalStoreOptions)
        {
        }
```

Replace it with the following code.

Code Listing 40: New ApplicationDbContext Code

```
        public ApplicationDbContext(
            DbContextOptions<ApplicationDbContext> options,
            IOptions<OperationalStoreOptions> operationalStoreOptions) :
            base(options, operationalStoreOptions)
        {
        }
```

Chapter 7 Creating New Tickets

In this chapter, we will create a page that will contain a form to create new help desk tickets.

In the Pages folder of the Client project, open the index.razor page and add the following under the **@page** directive.

Code Listing 41: Using and Inject Code

```
@using Microsoft.AspNetCore.Components.Authorization;
@using SyncfusionHelpDeskClient.Shared
@inject HttpClient Http
@inject IHttpClientFactory ClientFactory
```

Blazor Toast component

We will use the Syncfusion Blazor **Toast** component to display a brief pop-up message when a help desk ticket is submitted.

Add the following code markup under the existing closing **AuthorizeView** tag.

Code Listing 42: Toast Control

```
<SfToast ID="toast_default" Height="50"
         @ref="ToastObj"
         Title="Help Desk Ticket"
         Content="@ToastContent" TimeOut="5000">
    <ToastPosition X="Right"></ToastPosition>
</SfToast>
```

Next, add the following to the **@code** section.

Code Listing 43: Toast Properties

```
    SfToast ToastObj;
    private string ToastContent { get; set; } = "";
```

The **ToastObj** will provide programmatic access to the control, for example, to open it and close it, and the **ToastContent** property will allow us to set the text that is displayed in the control.

Forms and validation

Blazor provides a method for you to create forms with validation to collect data.

Forms

Blazor provides an **EditForm** control that allows us to validate a form using data annotations. These data annotations are defined in a class that is specified in the **Model** property of the **EditForm** control.

Validation

The **EditForm** control defines a method to handle **OnValidSubmit**. This method is triggered only when the data in the form satisfies all the validation rules defined by the data annotations.

Any validation errors are displayed using the **DataAnnotationsValidator** and/or the **ValidationSummary** control.

HelpDeskTicket Class

To support the forms and validation that we will implement, add a new class called **HelpDeskTicket.cs** in the Shared project.
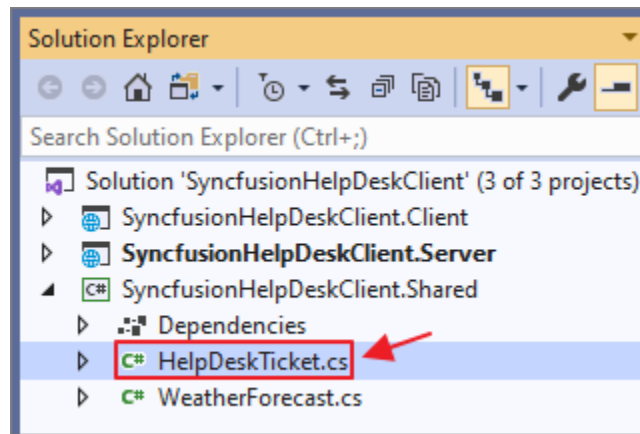
Figure 56: HelpDeskTicket.cs

Because this class is in the Shared project, it will be usable by both the Server and Client projects.

Use the following code for the class.

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace SyncfusionHelpDeskClient.Shared
{
    public class HelpDeskTicket
    {
        public int Id { get; set; }
        [Required]
        public string TicketStatus { get; set; }
        [Required]
        public DateTime TicketDate { get; set; }
        [Required]
        [StringLength(50, MinimumLength = 2,
            ErrorMessage =
            "Description must be a minimum of 2 and maximum of 50
characters.")]
        public string TicketDescription { get; set; }
        [Required]
        [EmailAddress]
        public string TicketRequesterEmail { get; set; }
        public string TicketGuid { get; set; }

        public List<HelpDeskTicketDetail> HelpDeskTicketDetails { get; set;
}
    }

    public class HelpDeskTicketDetail
    {
        public int Id { get; set; }
        public int HelpDeskTicketId { get; set; }
        public DateTime TicketDetailDate { get; set; }
        public string TicketDescription { get; set; }
    }
}
```

Syncfusion Blazor controls

We will also employ the following Syncfusion controls in our form:

- **TextBox**: Allows us to gather data from the user, and later display existing data.

- **DatePicker**: Allows the user to enter a date value, and later display an existing date value.
- **DropDownList**: Presents a list of predefined values (in our example, a list of ticket status values) that allows the user to choose a single value. Later we will use this control to also display an existing selected value.

The **DropDownList** control requires a data collection for the display options. To enable this, in the Shared project, add the following class.

Code Listing 45: HelpDeskStatus.cs

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace SyncfusionHelpDeskClient.Shared
{
    public class HelpDeskStatus
    {
        public string ID { get; set; }
        public string Text { get; set; }

        public static List<HelpDeskStatus> Statuses =
            new List<HelpDeskStatus>() {
            new HelpDeskStatus(){ ID= "New", Text= "New" },
            new HelpDeskStatus(){ ID= "Open", Text= "Open" },
            new HelpDeskStatus(){ ID= "Urgent", Text= "Urgent" },
            new HelpDeskStatus(){ ID= "Closed", Text= "Closed" },
            };
    }
}
```

Note: You can get more information on the Syncfusion Blazor controls on their website.

New ticket form

We will now add the code to display the form. In the Client project, add the following markup to the **index.razor** page.

Code Listing 46: Help Desk Form

```
<h3>New Help Desk Ticket</h3>
<br />
<EditForm ID="new-doctor-form" Model="@objHelpDeskTicket"
          OnValidSubmit="@HandleValidSubmit">
```

```
    <DataAnnotationsValidator></DataAnnotationsValidator>
    <div>
        <SfDropDownList TItem="HelpDeskStatus" TValue="string"
                        PopupHeight="230px" Index=0
                        Placeholder="Ticket Status"
                        DataSource="@HelpDeskStatus.Statuses"
                        FloatLabelType="@FloatLabelType.Always"
                        @bind-Value="@objHelpDeskTicket.TicketStatus">
            <DropDownListFieldSettings Text="Text"
                                       Value="ID">
            </DropDownListFieldSettings>
        </SfDropDownList>
    </div>
    <div>
        <SfDatePicker ID="TicketDate" Placeholder="Ticket Date"
                      FloatLabelType="@FloatLabelType.Always"
                      @bind-Value="@objHelpDeskTicket.TicketDate"
                      Max="DateTime.Now"
                      ShowClearButton="false"></SfDatePicker>
        <ValidationMessage For="@(() => objHelpDeskTicket.TicketDate)" />
    </div>
    <div>
        <SfTextBox Placeholder="Ticket Description"
                   FloatLabelType="@FloatLabelType.Always"
                   @bind-Value="@objHelpDeskTicket.TicketDescription">

        </SfTextBox>
        <ValidationMessage For="@(() =>
objHelpDeskTicket.TicketDescription)" />
    </div>
    <div>
        <SfTextBox Placeholder="Requester Email"
                   FloatLabelType="@FloatLabelType.Always"
                   @bind-Value="@objHelpDeskTicket.TicketRequesterEmail">

        </SfTextBox>
        <ValidationMessage For="@(() =>
objHelpDeskTicket.TicketRequesterEmail)" />
    </div>
    <br /><br />
    <div class="e-footer-content">
        <div class="button-container">
            <button type="submit" class="e-btn e-normal e-
primary">Save</button>
        </div>
    </div>
</EditForm>
```

Add the following to the **@code** section.

```
    // HttpClient to allow anonymous http calls.
    HttpClient NoAuthenticationClient;
    // Global property for the Help Desk Ticket.
    HelpDeskTicket objHelpDeskTicket = new HelpDeskTicket();
```

Finally, add the following code to insert the data into the database. This will call the **Post** method (because it is using **PostAsJsonAsync**) in the **SyncfusionHelpDeskController** (in the Server project), when the form has successfully passed validation.

Code Listing 48: HandleValidSubmit

```
    protected override void OnInitialized()
    {
        // Create a HttpClient to use for non-authenticated calls.
        NoAuthenticationClient =
            ClientFactory.CreateClient(
                "ServerAPI.NoAuthenticationClient");
    }

    public async Task HandleValidSubmit(EditContext context)
    {
        try
        {
            // Save the new Help Desk Ticket.
            // Create a new GUID for this Help Desk Ticket.
            objHelpDeskTicket.TicketGuid =
                System.Guid.NewGuid().ToString();

            await NoAuthenticationClient.PostAsJsonAsync(
                "SyncfusionHelpDesk", objHelpDeskTicket);

            // Clear the form.
            objHelpDeskTicket = new HelpDeskTicket();

            // Show the Toast.
            ToastContent = "Saved!";
            StateHasChanged();
            await this.ToastObj.Show();

        }
        catch (Exception ex)
        {
            ToastContent = ex.Message;
            StateHasChanged();
            await this.ToastObj.Show();
        }
    }
```
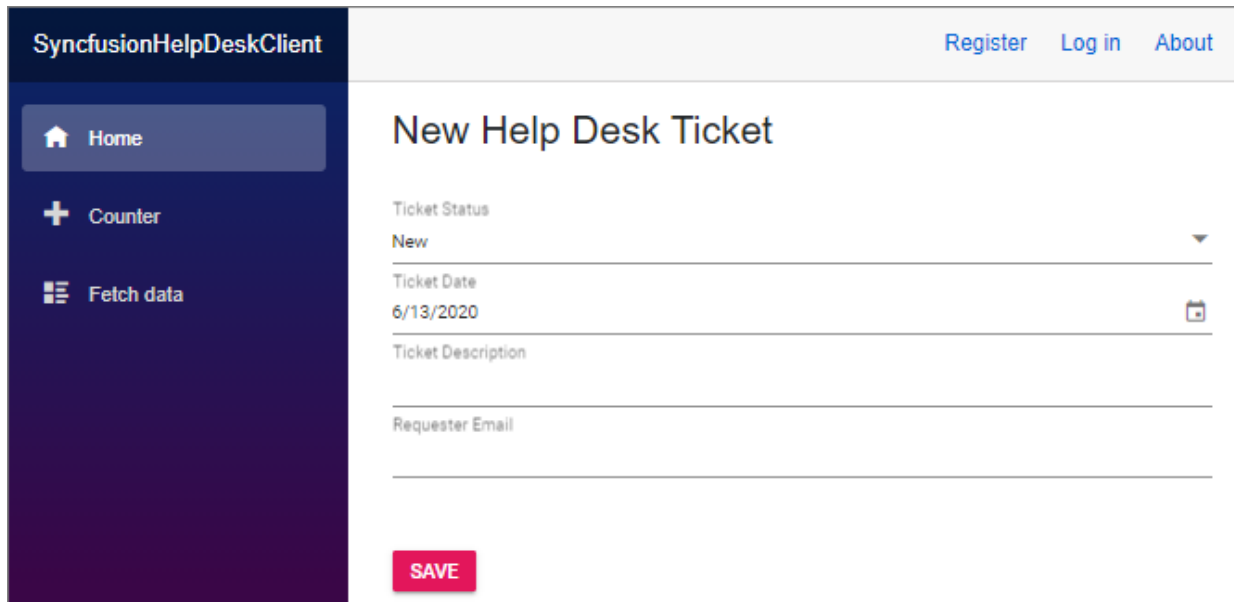
Test the form

We can now run the project and enter new help desk tickets.

Figure 57: New Help Desk Ticket Form

The user is presented with a new help desk ticket form.

Figure 58: Select Ticket Status

The user can select a ticket status using the **DropDownList** control.

Figure 59: Select Ticket Date

The user can select a ticket date using the **DatePicker** control.

Figure 60: Validation Errors

If the user tries to save an incomplete record, they will see validation errors.

Figure 61: Save Ticket

With a properly completed form, the user can click **Save** to save the data.
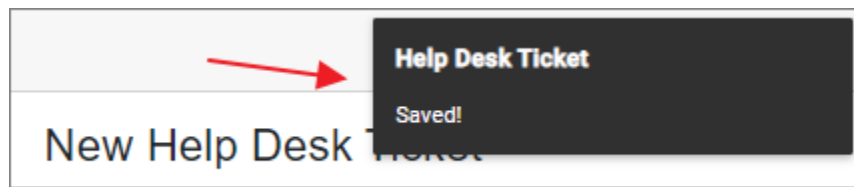
Figure 62: Toast Confirmation

The user will see a brief confirmation message by the `Toast` control.

Figure 63: View Data in the Database

If we look in the database, we will see the data has been added.

Enter data for at least six help desk tickets so that you will have enough data to demonstrate paging in the Syncfusion Data Grid covered in the following chapter.

Chapter 8 Help Desk Ticket Administration

In this chapter, we will detail the steps to create the screens to allow the administrators to manage the help desk tickets. In the ticket administration we will construct, help desk tickets can be updated and deleted, but ticket details that are part of a help desk ticket can only be added and deleted.

Create the Administration page
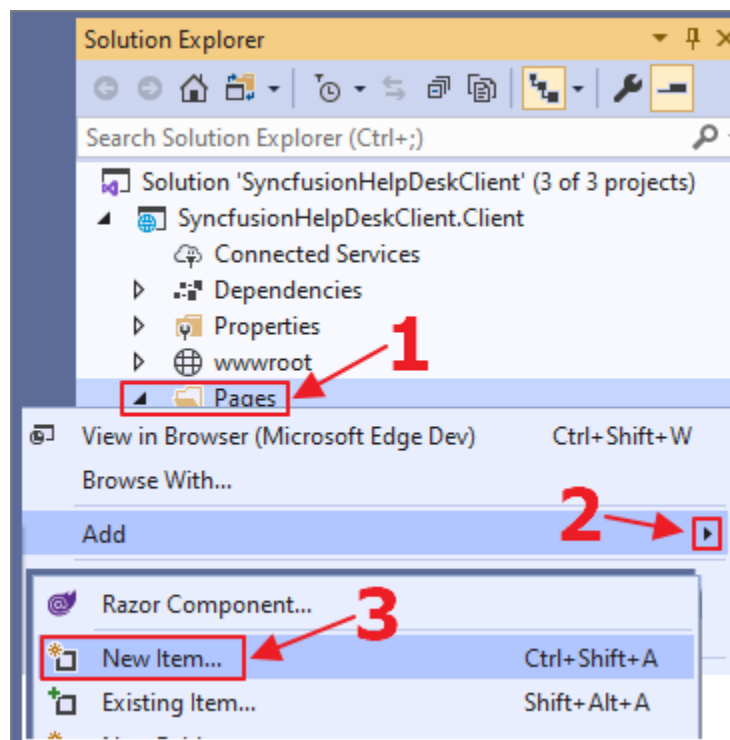
Figure 64: New Item

Add the Administration page to the Client project by right-clicking the **Pages** folder and selecting **Add** > **New Item**.
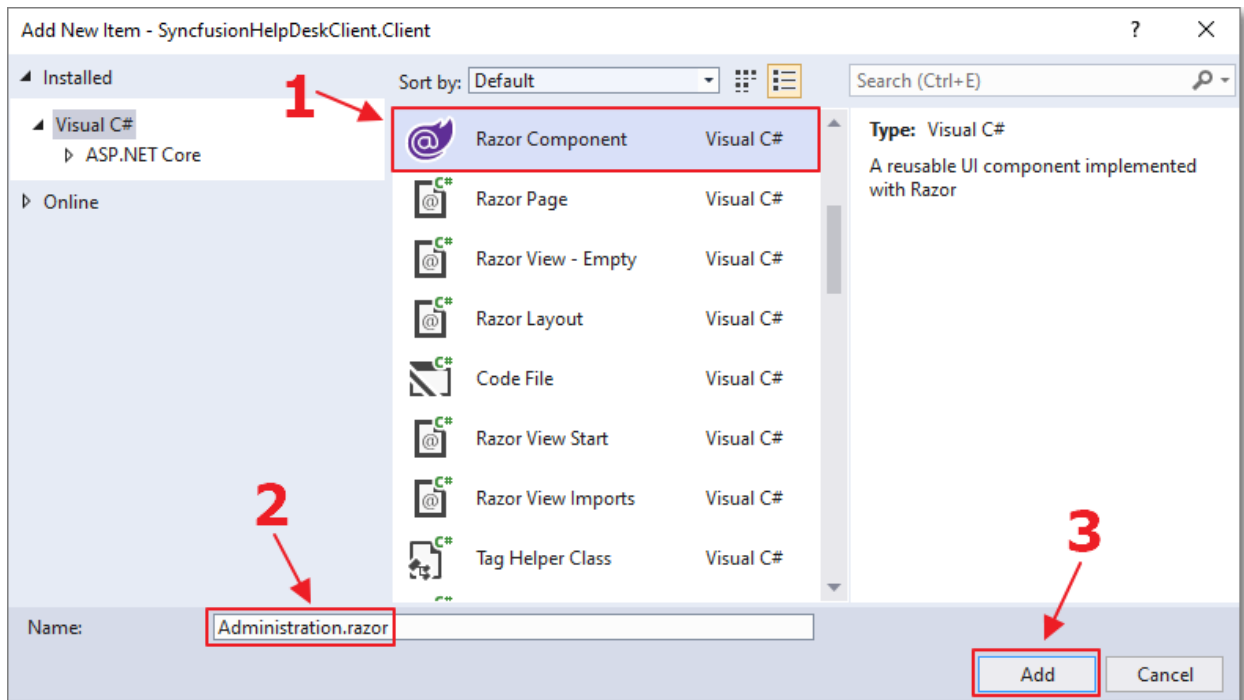
Figure 65: Add New Item

Select the **Razor Component** template, name the control Administration.razor, and click **Add**.

Use the following code for the **razor** control.

Code Listing 49: Administration.razor

```
@page "/administration"
@using SyncfusionHelpDeskClient.Shared
@inject HttpClient Http
@inject IHttpClientFactory ClientFactory

@*AuthorizeView control ensures that *@
@*Only users in the Administrators role can view this content*@
<AuthorizeView Roles="Administrators">

</AuthorizeView>
@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }
}
```

Add Link in NavMenu.razor

We will now add a link to the Administration control in the navigation menu.

Open the **NavMenu.razor** control in the Shared folder of the Client project and remove the following code (for the links to the counter and fetchdata pages).

Code Listing 50: Remove NavMenu Code

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="counter">
        <span class="oi oi-plus" aria-hidden="true"></span> Counter
    </NavLink>
</li>
<li class="nav-item px-3">
    <NavLink class="nav-link" href="fetchdata">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch
data
    </NavLink>
</li>
```

Add the following code in its place.

Code Listing 51: Add NavMenu Code

```
<AuthorizeView Roles="Administrators">
    <li class="nav-item px-3">
        <NavLink class="nav-link" href="administration">
            <span class="oi oi-plus" aria-hidden="true"></span>
Administration
        </NavLink>
    </li>
</AuthorizeView>
```

This will display a link to the Administration.razor page, but this link will only display for administrators because it is wrapped in an **AuthorizeView** control with the **Roles** property set to **Administrators**.
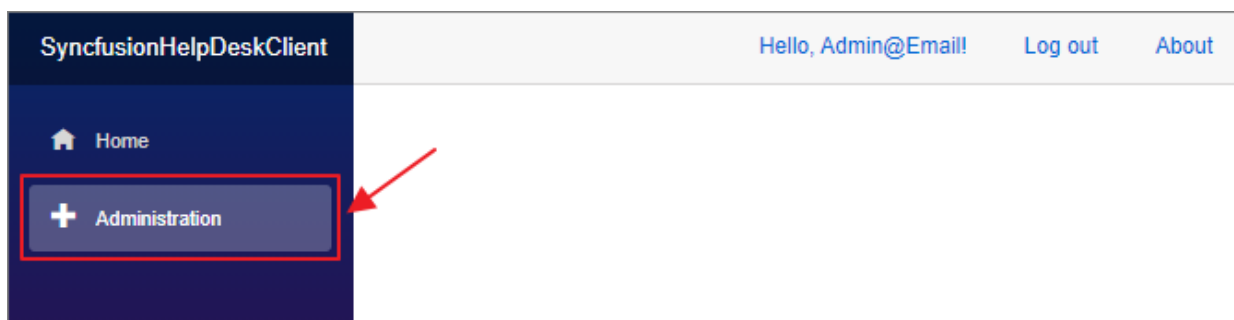
Figure 66: Administration Link

When we run the application and log in as the administrator, we see the Administration link. When we click on the link, we are navigated to the Administration page (currently blank).

Using Syncfusion Data Grid

The Syncfusion **Data Grid** control (programmatically called **SfGrid**) allows you to display tabular data. We will use it to display the help desk tickets. This control will allow us to page, sort, and trigger editing of the records.

It has a **DataSource** property that we will bind to a collection of the help desk tickets. It also has properties to allow for paging and sorting.

The control also allows us to define the columns that will contain the tabular data, as well as Edit and Delete buttons.

Enter the following inside the **AuthorizeView** control tag.

Code Listing 52: Data Grid

```
<div>
<div id="target" style="height: 500px;">
<SfGrid ID="Grid"
        DataSource="@colHelpDeskTickets"
        AllowPaging="true"
        AllowSorting="true"
        AllowResizing="true"
        AllowReordering="true">
    <SfDataManager Url="SyncfusionHelpDesk"
Adaptor="Adaptors.WebApiAdaptor">
    </SfDataManager>
    <GridPageSettings PageSize="5"></GridPageSettings>
    <GridEvents CommandClicked="OnCommandClicked"
                TValue="HelpDeskTicket">
    </GridEvents>
    <GridColumns>
        <GridColumn HeaderText="" TextAlign="TextAlign.Left" Width="150">
            <GridCommandColumns>
                <GridCommandColumn Type=CommandButtonType.Edit
                                    ButtonOption="@(new
CommandButtonOptions()
                                    { Content = "Edit" })">
                </GridCommandColumn>
                <GridCommandColumn Type=CommandButtonType.Delete
                                    ButtonOption="@(new
CommandButtonOptions()
                                    { Content = "Delete" })">
                </GridCommandColumn>
            </GridCommandColumns>
        </GridColumn>
        <GridColumn IsPrimaryKey="true" Field=@nameof(HelpDeskTicket.Id)
                    HeaderText="ID #" TextAlign="@TextAlign.Left"
                    Width="70">
```

```
        </GridColumn>
        <GridColumn Field=@nameof(HelpDeskTicket.TicketStatus)
                    HeaderText="Status" TextAlign="@TextAlign.Left"
                    Width="80">
        </GridColumn>
        <GridColumn Field=@nameof(HelpDeskTicket.TicketDate)
                    HeaderText="Date" TextAlign="@TextAlign.Left"
                    Width="80">
        </GridColumn>
        <GridColumn Field=@nameof(HelpDeskTicket.TicketDescription)
                    HeaderText="Description" TextAlign="@TextAlign.Left"
                    Width="150">
        </GridColumn>
        <GridColumn Field=@nameof(HelpDeskTicket.TicketRequesterEmail)
                    HeaderText="Requester" TextAlign="@TextAlign.Left"
                    Width="150">
        </GridColumn>
    </GridColumns>
</SfGrid>
</div>
</div>
```

Enter the following in the **@code** section.

Code Listing 53: Data Grid Code

```
    // SfDataManager calls the Get method in the SyncfusionHelpDesk
    // controller in the server project that the
    // SfGrid will use to only pull records for the
    // page that is currently selected.
    public IQueryable<HelpDeskTicket> colHelpDeskTickets { get; set; }

    public async void OnCommandClicked(
        CommandClickEventArgs<HelpDeskTicket> args)
    {
        // Code to be added later.
    }
```

The **SfDataManager** control populates the **colHelpDeskTickets** collection that is bound to the **DataSource** property of the data grid.

Figure 67: Data Grid Control

When we run the application, log in as an administrator, and navigate to the Administration page. We will see the help desk ticket records displayed in the data grid.

Figure 68: Sorting

We can sort by clicking the column headers.

Figure 69: Paging

The grid also enables paging.

Deleting a record

We will implement the functionality to delete a help desk ticket record.

To enable this, the **DataGrid** control allows us to create custom command buttons using the **CommandClicked** property that we currently have wired to the **OnCommandClicked** method.

We will use the **OnCommandClicked** method to display a pop-up, using the Syncfusion **Dialog** control, which will require the user to confirm that they want to delete the record.

Syncfusion Dialog

The **Dialog** control is used to display information and accept user input. It can display as a *modal control* that requires the user to interact with it before continuing to use any other part of the application.

Enter the following inside the **AuthorizeView** control tag.

Code Listing 54: Confirm Delete Dialog

```
<SfDialog Target="#target"
          Width="100px"
          Height="130px"
          IsModal="true"
          ShowCloseIcon="false"
          @bind-Visible="DeleteRecordConfirmVisibility">
    <DialogTemplates>
        <Header> DELETE RECORD ? </Header>
        <Content>
            <div class="button-container">
                <button type="submit"
                        class="e-btn e-normal e-primary"
                        @onclick="ConfirmDeleteYes">
                    Yes
                </button>
                <button type="submit"
                        class="e-btn e-normal"
                        @onclick="ConfirmDeleteNo">
                    No
                </button>
            </div>
        </Content>
    </DialogTemplates>
</SfDialog>
```

Enter the following in the **@code** section.

Code Listing 55: Help Desk Ticket Object and Property

```
    // Global property for the Help Desk Ticket.
    private HelpDeskTicket SelectedTicket = new HelpDeskTicket();

    // Property to control the delete dialog.
    public bool DeleteRecordConfirmVisibility { get; set; } = false;
```

Change the **OnCommandClicked** method to the following to respond to the Delete button being clicked for a help desk record.

This method will open the confirmation dialog.

Code Listing 56: Open Confirmation Dialog

```
public async void OnCommandClicked(
        CommandClickEventArgs<HelpDeskTicket> args)
{
    if (args.CommandColumn.ButtonOption.Content == "Delete")
    {
        // We only need the TicketGuid
        // of the selected Help Desk Ticket.
        SelectedTicket = new HelpDeskTicket();
        SelectedTicket.TicketGuid = args.RowData.TicketGuid;

        // Open Delete confirmation dialog.
        this.DeleteRecordConfirmVisibility = true;
        StateHasChanged();
    }
}
```

> **Note:** *StateHasChanged*, **used in the preceding code, notifies a component (in this case the Administration page) that its state has changed and causes that component to rerender. This is usually only required when that state was changed by a JavaScript interop call. The Syncfusion Dialog control uses JavaScript interop in the underlying code to open the dialog, so** *StateHasChanged* **is required in this case.**

Add the following method that will simply close the dialog if the user clicks the No button on the dialog.

Code Listing 57: Confirmation Close Dialog

```
public void ConfirmDeleteNo()
{
    // Open the dialog
    // to give the user a chance
    // to confirm they want to delete the record.
    this.DeleteRecordConfirmVisibility = false;
}
```

Using @ref (capture references to components)

Adding an **@ref** attribute to a component allows you to programmatically access and manipulate a control or component. To implement it, you add the **@ref** attribute to a component, then define a field with the same type as the component.

Add the following property to the **SfGrid** control.

Code Listing 57: Data Grid @ref

```
@ref="gridObj"
```

Now, add the following corresponding field to the **@code** section.

Code Listing 58: Data Grid Property

```
SfGrid<HelpDeskTicket> gridObj;
```

Now that we have done this, we can add the following method to delete the record, if the user clicks the Yes button in the dialog.

This method will refresh the data grid by calling **gridObj.Refresh()**, which uses the **gridObj** object defined with the **@ref** attribute.

Code Listing 59: Delete Record

```
public async void ConfirmDeleteYes()
{
    // The user selected Yes to delete the
    // selected Help Desk Ticket.
    // Delete the record.
    await Http.DeleteAsync(
        "SyncfusionHelpDesk?HelpDeskTicketGuid=" +
        SelectedTicket.TicketGuid);

    // Close the dialog.
    this.DeleteRecordConfirmVisibility = false;
    StateHasChanged();

    // Refresh the SfGrid
    // so the deleted record will not show.
    gridObj.Refresh();
}
```

When we run the application, we can click **Delete** to open the dialog.
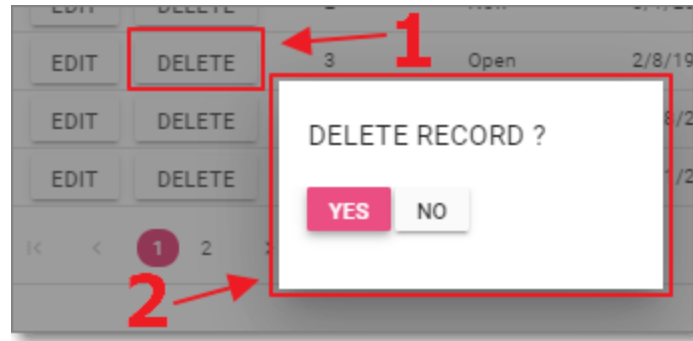
Figure 70: Delete Confirmation

Clicking the **No** button will simply close the dialog. Clicking the **Yes** button will delete the selected record and refresh the data grid.

Edit ticket control

Figure 71: Edit Ticket Control

We will construct an `EditTicket` control that will be placed inside a dialog in the Administration page and displayed when an administrator wants to edit a help desk ticket. We do this to allow this control to be reused in the EmailTicketEdit.razor page (covered in the following chapter).

In the Pages folder of the Client project, create a new control called `EditTicket.razor` using the following code.

```razor
@using System.Security.Claims;
@using Syncfusion.Blazor.DropDowns
@using SyncfusionHelpDeskClient.Shared
@inject HttpClient Http
@inject IHttpClientFactory ClientFactory

<div>
    <SfDropDownList TItem="HelpDeskStatus" Enabled="!isReadOnly"
                    TValue="string" PopupHeight="230px" Index=0
                    Placeholder="Ticket Status"
                    DataSource="@HelpDeskStatus.Statuses"
                    FloatLabelType="@FloatLabelType.Always"
                    @bind-Value="@SelectedTicket.TicketStatus">
        <DropDownListFieldSettings Text="Text"
                                   Value="ID">
        </DropDownListFieldSettings>
    </SfDropDownList>
</div>
<div>
    <SfDatePicker ID="TicketDate" Enabled="!isReadOnly"
                  Placeholder="Ticket Date"
                  FloatLabelType="@FloatLabelType.Always"
                  @bind-Value="@SelectedTicket.TicketDate"
                  Max="DateTime.Now"
                  ShowClearButton="false">
    </SfDatePicker>
</div>
<div>
    <SfTextBox Enabled="!isReadOnly" Placeholder="Ticket Description"
               FloatLabelType="@FloatLabelType.Always"
               @bind-Value="@SelectedTicket.TicketDescription">
    </SfTextBox>
</div>
<div>
    <SfTextBox Enabled="!isReadOnly" Placeholder="Requester Email"
               FloatLabelType="@FloatLabelType.Always"
               @bind-Value="@SelectedTicket.TicketRequesterEmail">
    </SfTextBox>
</div>
@if (SelectedTicket.HelpDeskTicketDetails != null)
{
    @if (SelectedTicket.HelpDeskTicketDetails.Count() > 0)
    {
        <table class="table">
            <thead>
                <tr>
                    <th>Date</th>
```

```html
                    <th>Description</th>
                </tr>
            </thead>
            <tbody>
                @foreach (var TicketDetail in
              SelectedTicket.HelpDeskTicketDetails)
              {
                    <tr>
                        <td>

@TicketDetail.TicketDetailDate.ToShortDateString()
                        </td>
                        <td>
                            @TicketDetail.TicketDescription
                        </td>
                    </tr>
              }
            </tbody>
        </table>
    }
    <SfTextBox Placeholder="NewHelp Desk Ticket Detail"
               @bind-Value="@NewHelpDeskTicketDetailText">
    </SfTextBox>
    <SfButton CssClass="e-small e-success"
              @onclick="AddHelpDeskTicketDetail">
        Add
    </SfButton>
}
<br />
@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

    [Parameter]
    public HelpDeskTicket SelectedTicket { get; set; }

    public bool isReadOnly = true;
    ClaimsPrincipal CurrentUser = new ClaimsPrincipal();

    string NewHelpDeskTicketDetailText = "";

    protected override async Task OnInitializedAsync()
    {
        // Get the current user.
        CurrentUser = (await authenticationStateTask).User;

        // If there is a logged in user
        // they are an Administrator.
        // Enable editing.
```

```csharp
        isReadOnly = !CurrentUser.Identity.IsAuthenticated;
    }

    private void AddHelpDeskTicketDetail()
    {
        // Create New HelpDeskTicketDetails record.
        HelpDeskTicketDetail NewHelpDeskTicketDetail =
            new HelpDeskTicketDetail();

        NewHelpDeskTicketDetail.HelpDeskTicketId =
        SelectedTicket.Id;

        NewHelpDeskTicketDetail.TicketDetailDate =
        DateTime.Now;

        NewHelpDeskTicketDetail.TicketDescription =
        NewHelpDeskTicketDetailText;

        // Add to collection.
        SelectedTicket.HelpDeskTicketDetails
            .Add(NewHelpDeskTicketDetail);

        // Clear the Text Box.
        NewHelpDeskTicketDetailText = "";
    }
}
```

Notice that this exposes a **SelectedTicket** parameter (of type **HelpDeskTicket**) that will accept a reference of a help desk ticket record.

In the **Administration.razor** control, add the following markup to display the EditTicket.razor page in a **Dialog** control.

Code Listing 61: EditTicket Dialog

```razor
<SfDialog Target="#target"
          Width="500px"
          Height="500px"
          IsModal="true"
          ShowCloseIcon="true"
          @bind-Visible="EditDialogVisibility">
    <DialogTemplates>
        <Header> EDIT TICKET # @SelectedTicket.Id</Header>
        <Content>
            <EditTicket SelectedTicket="@SelectedTicket" />
        </Content>
        <FooterTemplate>
            <div class="button-container">
```

```
                <button type="submit"
                        class="e-btn e-normal e-primary"
                        @onclick="SaveTicket">
                    Save
                </button>
            </div>
        </FooterTemplate>
    </DialogTemplates>
</SfDialog>
```

Note that this instantiates the **EditTicket** control (the EditTicket.razor code page) and passes a reference to the currently selected help desk ticket record (**@SelectedTicket**) through the **SelectedTicket** attribute.

Add the following to the **@code** section to implement the functionality for the Save button on the dialog.

Code Listing 62: SaveTicket Method

```
    public bool EditDialogVisibility { get; set; } = false;

    public async Task SaveTicket()
    {
        // Update the selected Help Desk Ticket.
        await Http.PutAsJsonAsync(
        "SyncfusionHelpDesk", SelectedTicket);

        // Close the Edit dialog.
        this.EditDialogVisibility = false;

        // Refresh the SfGrid
        // so the changes to the selected
        // Help Desk Ticket are reflected.
        gridObj.Refresh();
    }
```

Next, add the following code to the **OnCommandClicked** method to open the dialog when the Edit button is clicked on a row in the data grid.

Code Listing 63: Open Dialog

```
        if (args.CommandColumn.ButtonOption.Content == "Edit")
        {
            // Get the selected Help Desk Ticket.
            SelectedTicket =
                await Http.GetFromJsonAsync<HelpDeskTicket>(
                    "Email?HelpDeskTicketGuid=" +
                    args.RowData.TicketGuid);
```

```
            // Open the Edit dialog.
            this.EditDialogVisibility = true;
            StateHasChanged();
        }
```

Finally, in the Server project, create a new class in the Controllers folder called **EmailSender.cs** using the following code.

Code Listing 64: EmailSender.cs

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Primitives;

using SyncfusionHelpDeskClient.Shared;

namespace SyncfusionHelpDesk.Data
{
    [ApiController]
    [Route("[controller]")]
    public class EmailController : ControllerBase
    {
        private readonly IConfiguration configuration;
        private readonly IHttpContextAccessor httpContextAccessor;
        private readonly SyncfusionHelpDeskContext _context;

        public EmailController(
            IConfiguration Configuration,
            IHttpContextAccessor HttpContextAccessor,
            SyncfusionHelpDeskContext context)
        {
            configuration = Configuration;
            httpContextAccessor = HttpContextAccessor;
            _context = context;
        }

        [HttpGet]
        [AllowAnonymous]
        public object Get()
        {
            // Return only one Ticket.
```

```
            StringValues HelpDeskTicketGuidProperty;

            string HelpDeskTicketGuid =
                (Request.Query.TryGetValue("HelpDeskTicketGuid",
                out HelpDeskTicketGuidProperty))
                ? HelpDeskTicketGuidProperty.ToString() : "";

            var ExistingTicket = _context.HelpDeskTickets
                .Include(x => x.HelpDeskTicketDetails)
                .Where(x => x.TicketGuid == HelpDeskTicketGuid)
                .FirstOrDefault();

            return ExistingTicket;
        }
    }
}
```

This class is called by the **OnCommandClicked** method in the **Administration.razor** control to get the details of the currently selected record, so that it can be displayed in the dialog and updated.

This class will later be expanded when we add email functionally, in the next chapter.

Figure 72: Edit Ticket Dialog

When we run the application, we can click Edit next to a record to open it up in the dialog.

Any of the help desk ticket values at the top of the form can be edited and saved by clicking **Save** at the bottom of the dialog.

Near the bottom of the form, help desk ticket detail records can be added by entering text in the text box and clicking **Add**.

Figure 73: Edit Ticket

The help desk detail record will be added, but it will not be saved until the **Save** button is clicked.

Chapter 9 Sending Emails
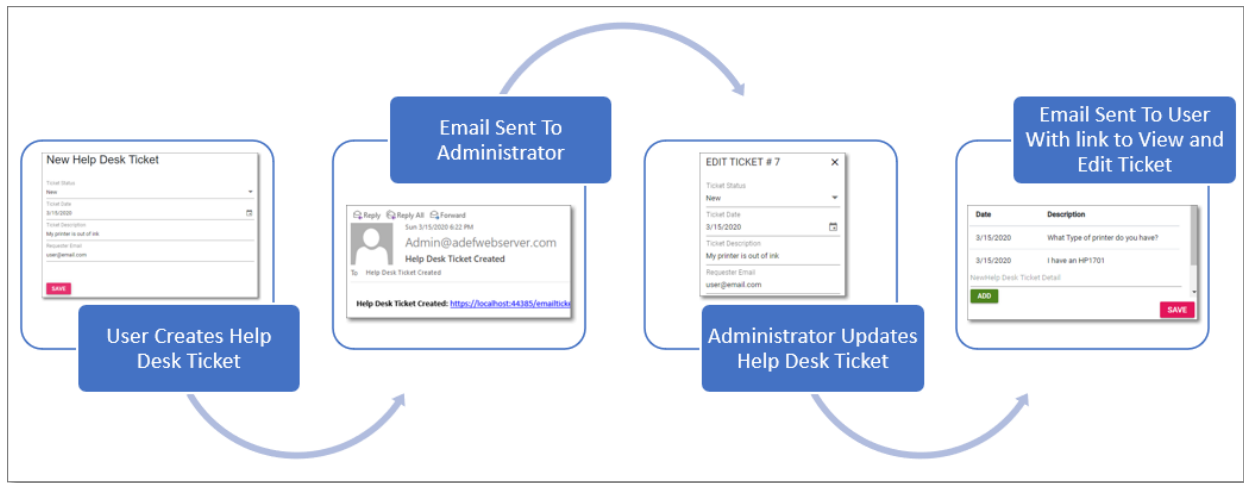
Figure 74: Email Process

In this chapter, we will create code that will allow help desk tickets to be updated by help desk ticket creators and administrators by simply clicking a link in an email.

We will create the code that will email the administrators when a new help desk ticket has been created, as well as email help desk ticket creators and administrators when help desk tickets are updated.

Email using SendGrid

To enable emails, create a free account [here](#) and obtain an email API key.

Open the **appsettings.json** file and add the following two lines below the opening curly bracket, entering your SendGrid key for the **SENDGRID_APIKEY** property, and your email address for the **SenderEmail** property.

Code Listing 65: appsettings.json

```
"SENDGRID_APIKEY": "enter your key from app.sendgrid.com",
"SenderEmail": "enter your email address",
```
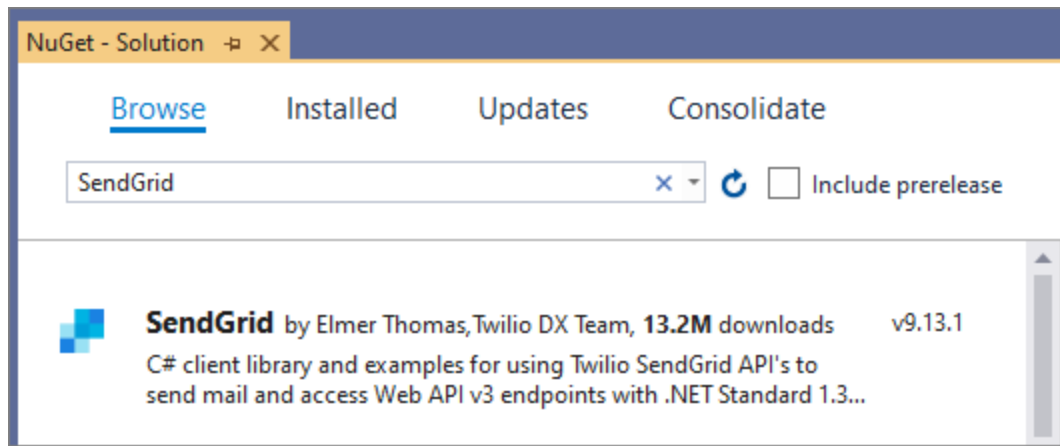
Figure 75: SendGrid NuGet Package

Install the SendGrid NuGet package in the Server project.

EmailSender class

We will update the **EmailSender.cs** class to read the settings from the appsettings.json file and send emails.

Send emails—new help desk ticket

We will need a class to pass the information needed to send emails from the Client project to code in the Server project.

Add a new class to the Shared project named **HelpDeskEmail.cs** using the following code.

Code Listing 66: HelpDeskEmail.cs

```csharp
using System;
using System.Collections.Generic;
using System.Text;

namespace SyncfusionHelpDeskClient.Shared
{
    public class HelpDeskEmail
    {
        public string EmailType { get; set; }
        public string EmailAddress { get; set; }
        public string TicketGuid { get; set; }
    }
}
```

To actually send the email, open the **EmailController.cs** file in the **Controllers** folder of the **Server** project, and add the following **using** statements.

Code Listing 67: EmailController Using Statements

```csharp
using SendGrid;
using SendGrid.Helpers.Mail;
```

Add the following method to read the **SENDGRID_APIKEY** from the **appsettings.json** file and send the emails.

Code Listing 68: EmailController Send Email Code

```csharp
    [HttpPost]
    [AllowAnonymous]
    public Task Post(
        HelpDeskEmail objHelpDeskEmail)
    {
        try
        {
            // Email settings.
            SendGridMessage msg = new SendGridMessage();
            var apiKey = configuration["SENDGRID_APIKEY"];
            var senderEmail = configuration["SenderEmail"];
            var client = new SendGridClient(apiKey);

            var FromEmail = new EmailAddress(
                senderEmail,
                senderEmail
                );

            // Format Email contents.
            string strPlainTextContent =
                $"{objHelpDeskEmail.EmailType}: " +
                $"{GetHelpDeskTicketUrl(objHelpDeskEmail.TicketGuid)}";

            string strHtmlContent =
                $"<b>{objHelpDeskEmail.EmailType}:</b> ";
            strHtmlContent = strHtmlContent +
                $"<a href='{
GetHelpDeskTicketUrl(objHelpDeskEmail.TicketGuid) }'>";
            strHtmlContent = strHtmlContent +
                $"{GetHelpDeskTicketUrl(objHelpDeskEmail.TicketGuid)}</a>";

            if (objHelpDeskEmail.EmailType == "Help Desk Ticket Created")
            {
                msg = new SendGridMessage()
                {
                    From = FromEmail,
```

```csharp
                Subject = objHelpDeskEmail.EmailType,
                PlainTextContent = strPlainTextContent,
                HtmlContent = strHtmlContent
            };

            // Created Email always goes to Administrator.
            // Send to senderEmail configured in appsettings.json
            msg.AddTo(
                new EmailAddress(senderEmail,
objHelpDeskEmail.EmailType)
                );
        }

        if (objHelpDeskEmail.EmailType == "Help Desk Ticket Updated")
        {
            // Must pass a valid GUID.
            // Get the existing record.
            if (_context.HelpDeskTickets
                .Where(x => x.TicketGuid ==
objHelpDeskEmail.TicketGuid)
                .FirstOrDefault() != null)
            {
                // See if the user is the Administrator.
                if (!this.User.IsInRole("Administrators"))
                {
                    // Always send email to Administrator.
                    objHelpDeskEmail.EmailAddress = senderEmail;
                }

                msg = new SendGridMessage()
                {
                    From = FromEmail,
                    Subject = objHelpDeskEmail.EmailType,
                    PlainTextContent = strPlainTextContent,
                    HtmlContent = strHtmlContent
                };

                // Send Email.
                msg.AddTo(new EmailAddress(
                    objHelpDeskEmail.EmailAddress,
                    objHelpDeskEmail.EmailType)
                    );
            }
            else
            {
                Task.FromResult("Error - Bad TicketGuid");
            }
        }
```

```
                client.SendEmailAsync(msg);
        }
        catch
        {
            // Could not send email.
            // Perhaps SENDGRID_APIKEY not set in
            // appsettings.json
        }

        return Task.FromResult("");
    }

    // Utility

    #region private string GetHelpDeskTicketUrl(string TicketGuid)
    private string GetHelpDeskTicketUrl(string TicketGuid)
    {
        var request = httpContextAccessor.HttpContext.Request;

        var host = request.Host.ToUriComponent();

        var pathBase = request.PathBase.ToUriComponent();

        return
$@"{request.Scheme}://{host}{pathBase}/emailticketedit/{TicketGuid}";
    }
    #endregion
}
```

Next, open the Index.razor file in the Client project, and add the following code to the end of the **HandleValidSubmit** method (under the **PostAsJsonAsync** line).

Code Listing 69: HandleValidSubmit method

```
        // Send Email
        HelpDeskEmail objHelpDeskEmail = new HelpDeskEmail();
        objHelpDeskEmail.EmailType = "Help Desk Ticket Created";
        objHelpDeskEmail.EmailAddress = "";
        objHelpDeskEmail.TicketGuid = objHelpDeskTicket.TicketGuid;

        await NoAuthenticationClient.PostAsJsonAsync(
            "Email", objHelpDeskEmail);
```

This code will use **PostAsJsonAsync** to call the **Post** method of the EmailController.cs file to send the emails. This code will be called when a user creates a new help desk ticket.

Send emails—updated help desk ticket

To send an email to the help desk ticket creator when the help desk ticket is updated, open the **Administration.razor** control and add the following code to the end of the **SaveTicket** method.

Code Listing 70: Sending Emails in the Administration Control

```
// Send Email
HelpDeskEmail objHelpDeskEmail =
    new HelpDeskEmail();

objHelpDeskEmail.EmailType =
    "Help Desk Ticket Updated";

objHelpDeskEmail.EmailAddress =
    SelectedTicket.TicketRequesterEmail;

objHelpDeskEmail.TicketGuid =
    SelectedTicket.TicketGuid;

await Http.PostAsJsonAsync(
"Email", objHelpDeskEmail);
```

Route parameters

When a help desk ticket is initially created and saved to the database, it is assigned a unique GUID value. When an email is sent to notify the help desk ticket creator and administrator, the email will contain a link that passes this GUID to the Blazor control that we will create. This control will be decorated with a **@page** directive that contains a route parameter.

In the Pages folder of the Client project, create a new control called **EmailTicketEdit.razor** with the following code.

Code Listing 71: EmailTicketEdit Route

```
@page "/emailticketedit/{TicketGuid}"
```

This line, together with a field in the **@code** section called **TicketGuid** (of type **string**), will allow this control to be loaded and passed a value for **TicketGuid** from a link in the email.

Enter the following code as the remaining code for the file.

Code Listing 72: EmailTicketEdit Code

```
@using SyncfusionHelpDeskClient.Shared
@inject HttpClient Http
```

```razor
@inject IHttpClientFactory ClientFactory

<div id="target" style="height: 500px;">
    @if (!EditDialogVisibility)
    {
        <h2>Your response has been saved</h2>
        <h4>Thank You!</h4>
    }
</div>
<SfDialog Target="#target"
              Width="500px"
              Height="500px"
              IsModal="true"
              ShowCloseIcon="false"
              @bind-Visible="EditDialogVisibility">
    <DialogTemplates>
        <Header> EDIT TICKET # @SelectedTicket.Id</Header>
        <Content>
            <EditTicket SelectedTicket="@SelectedTicket" />
        </Content>
        <FooterTemplate>
            <div class="button-container">
                <button type="submit"
                        class="e-btn e-normal e-primary"
                        @onclick="SaveTicket">
                    Save
                </button>
            </div>
        </FooterTemplate>
    </DialogTemplates>
</SfDialog>

@code {
    [Parameter] public string TicketGuid { get; set; }

    HttpClient NoAuthenticationClient;
    private HelpDeskTicket SelectedTicket = new HelpDeskTicket();
    private bool EditDialogVisibility = true;

    protected override async Task
        OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            // Create a httpClient to use for non-authenticated calls.
            NoAuthenticationClient =

ClientFactory.CreateClient("ServerAPI.NoAuthenticationClient");
```

```csharp
            // Get the Help Desk Ticket associated with
            // the GUID that was passed to the control.
            SelectedTicket =
            await NoAuthenticationClient.GetFromJsonAsync<HelpDeskTicket>(
                "Email?HelpDeskTicketGuid=" + TicketGuid);

            StateHasChanged();
        }
    }

    public async Task SaveTicket()
    {
        // Save the Help Desk Ticket.
        await NoAuthenticationClient.PutAsJsonAsync(
            "SyncfusionHelpDesk", SelectedTicket);

        // Close the Dialog.
        EditDialogVisibility = false;

        // Send Email.
        HelpDeskEmail objHelpDeskEmail = new HelpDeskEmail();
        objHelpDeskEmail.EmailType = "Help Desk Ticket Updated";
        objHelpDeskEmail.EmailAddress =
SelectedTicket.TicketRequesterEmail;
        objHelpDeskEmail.TicketGuid = SelectedTicket.TicketGuid;

        await NoAuthenticationClient.PostAsJsonAsync(
        "Email", objHelpDeskEmail);

        return;
    }
}
```

Notice that this page also includes the **EditTicket** control, effectively reusing that control in both this page and the Administration page.

Email link

Help Desk Ticket Created: https://localhost:44325/emailticketedit/0d3766d2-c04d-4cf2-bdb4-c956ee35ef2d

Figure 76: Email Link

When we run the application and create a new help desk ticket, the administrator is sent an email with a link that will take the administrator directly to the help desk ticket.

We've now seen how Blazor technology enables you to create sophisticated, manageable, and extensible single-page applications using C# and Razor syntax. Try it for yourself!