BLAZOR

SUCCINCTLY

BY **MICHAEL WASHINGTON**

Syncfusion®

Blazor Succinctly

By

Michael Washington

Foreword by Daniel Jebaraj

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
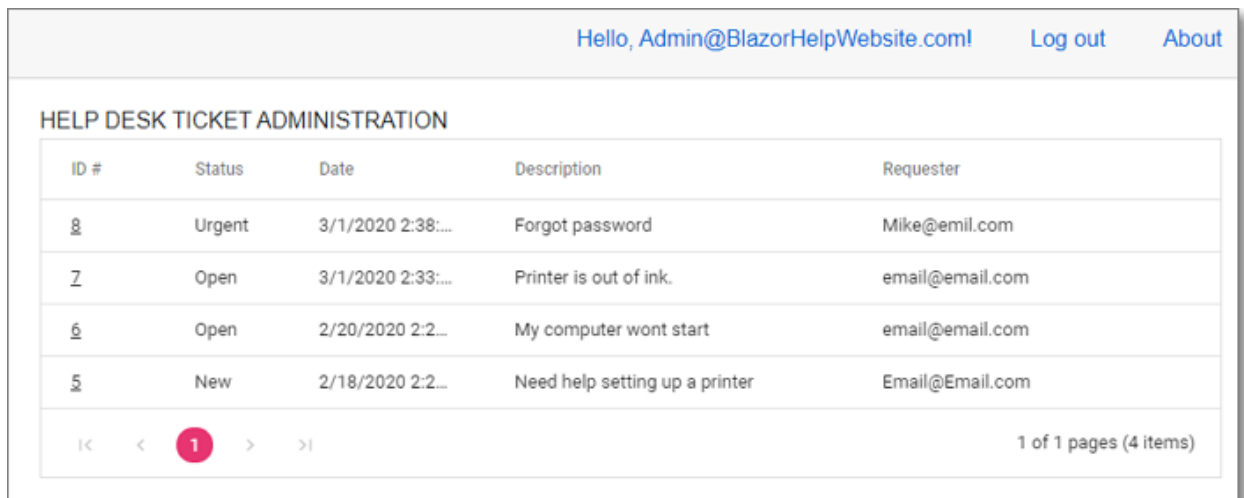
Michael Washington is a Microsoft MVP, an ASP.NET C# programmer, and the founder of BlazorHelpWebsite.com. He is the author of *An Introduction to Building Applications with Blazor*. He has extensive knowledge of process improvement, billing systems, and student information systems. He has a son, Zachary, 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 is an alternative to other single-page application frameworks such as Angular, React, or Vue.

Figure 1: Help Desk Administration

In this book, we will cover the core elements of Blazor, 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 the 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
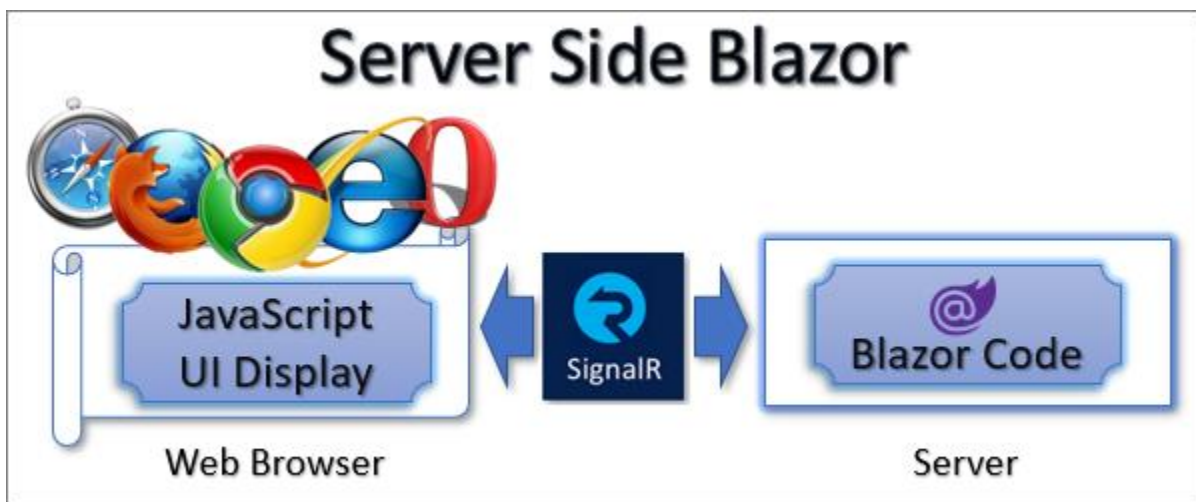
Figure 2: Server-Side Blazor
© BlazorHelpWebsite.com (used with permission)

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 web browser.

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

Client-side (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, server-side) using Razor technology written with the C# language.

Core Blazor features

Components and routing

A Blazor application is composed of components. A component is a chunk of code consisting of the user interface and the 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.

```
@page "/componentexample"

<h3>This is Component Example</h3>

@code {

}
```

The following shows what the component looks like in a running application.
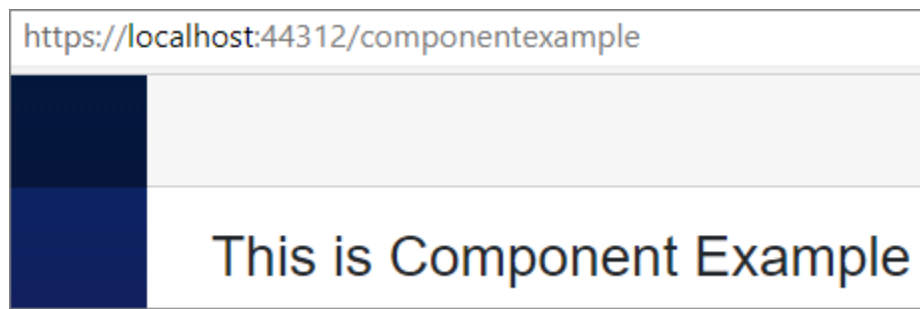
Figure 4: A Simple Component

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.
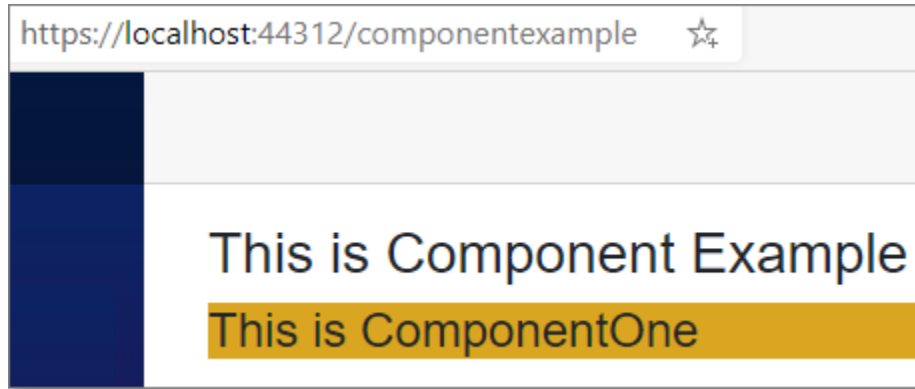
Figure 5: Nested Component

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** using the following code.

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.

Code Listing 5: ParameterExample.razor

```
@page "/parameterexample"
```

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

<ParameterExampleComponent Title="Passed from Parent" />

@code {

}
```

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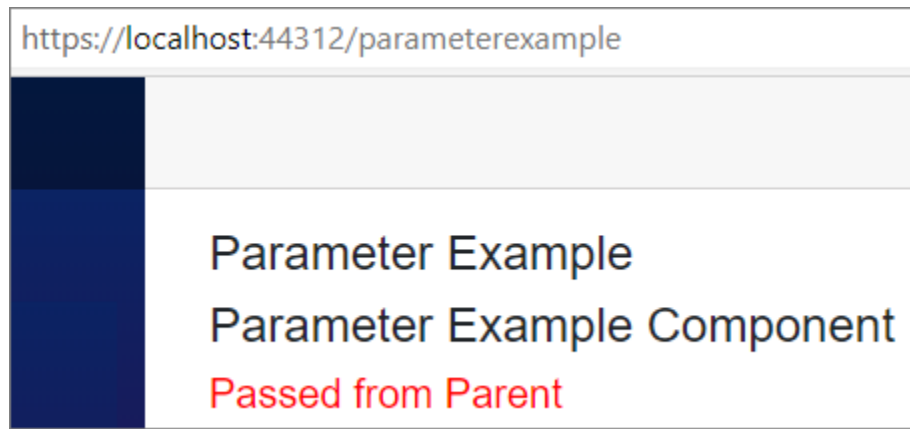
Figure 6: Parameter Example

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

```
<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, on the page, the value entered into the text input box as text is typed into the input box.

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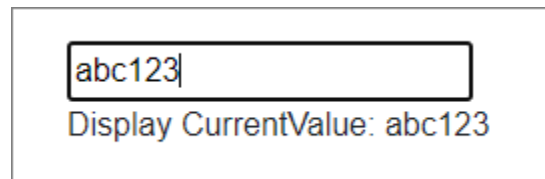
Figure 8: Two-Way Binding

Events

Raising events in Razor components is straightforward. The following example demonstrates using the **@onclick** event handler to execute a 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.
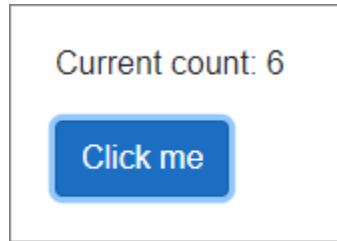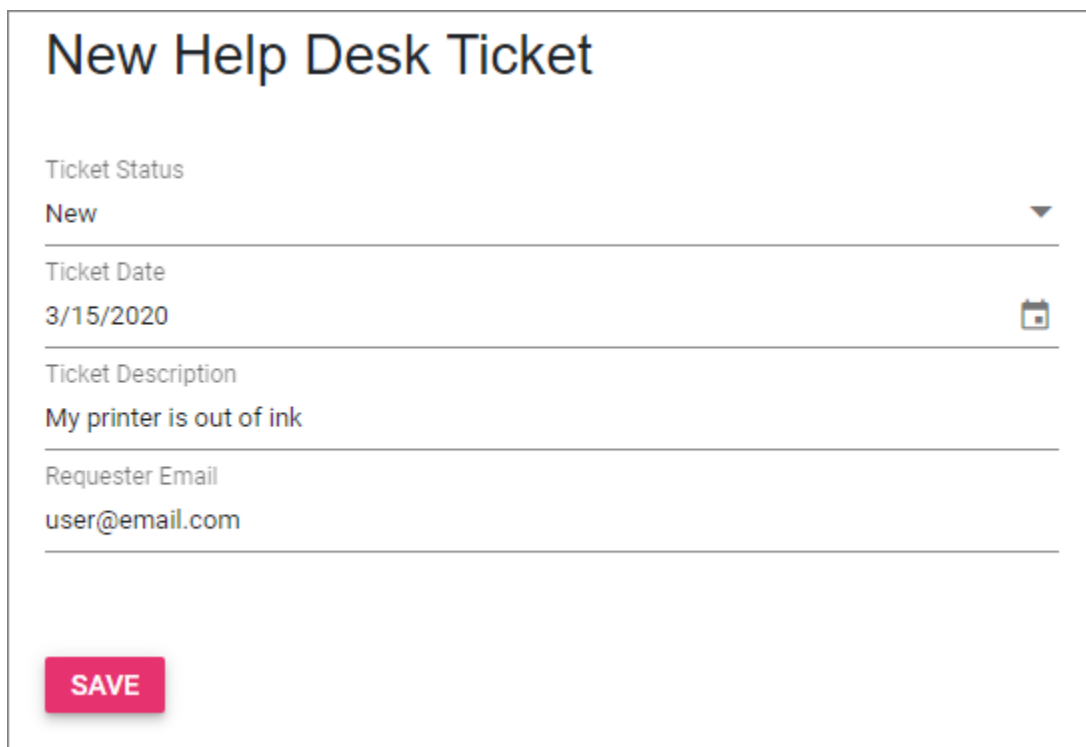
Figure 9: Simple Event

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.
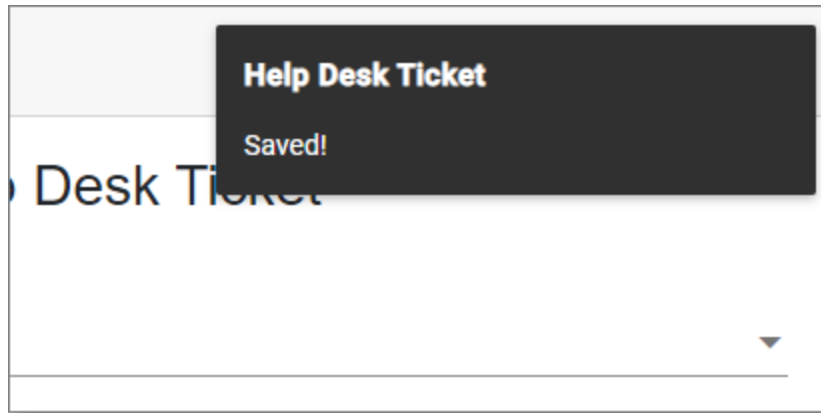
Figure 11: Syncfusion Toast

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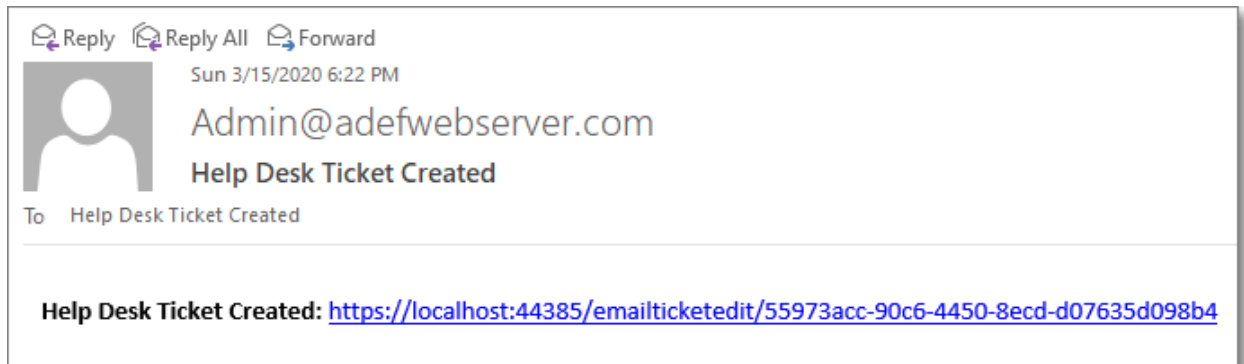
Figure 12: Email Notification

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 the desired help desk ticket details have been added, 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 also allows them to 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
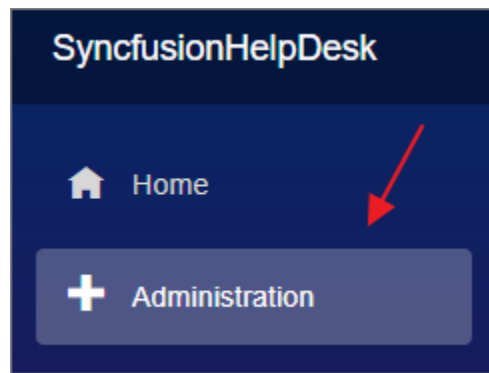
Figure 16: Administration Menu Link

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.
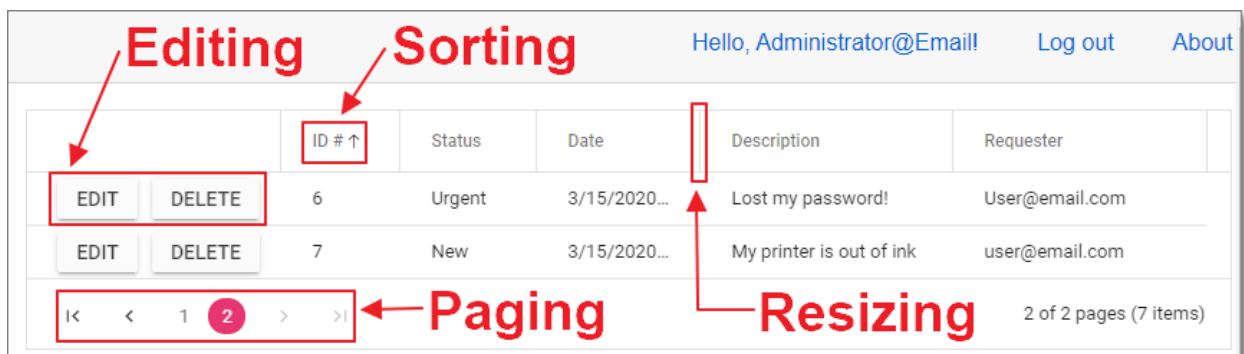
Figure 17: Syncfusion Data Grid

This will display the Syncfusion Data Grid that will allow the administrator to 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.
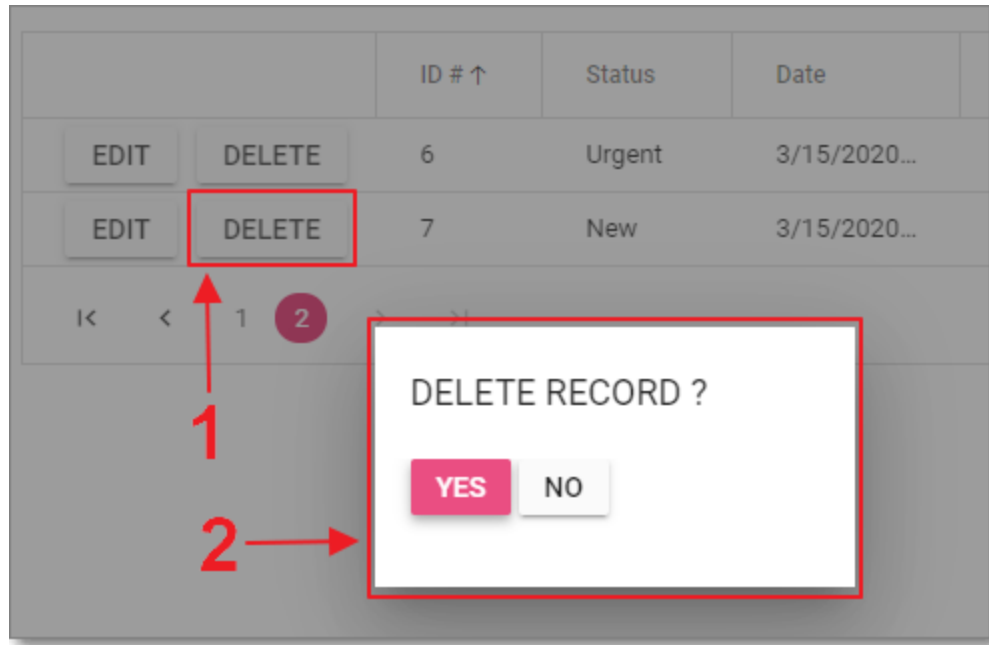
Figure 19: Delete Confirmation

Clicking the DELETE button next to a record in the Data Grid will open up 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 at:
https://github.com/ADefWebserver/SyncfusionHelpDesk.
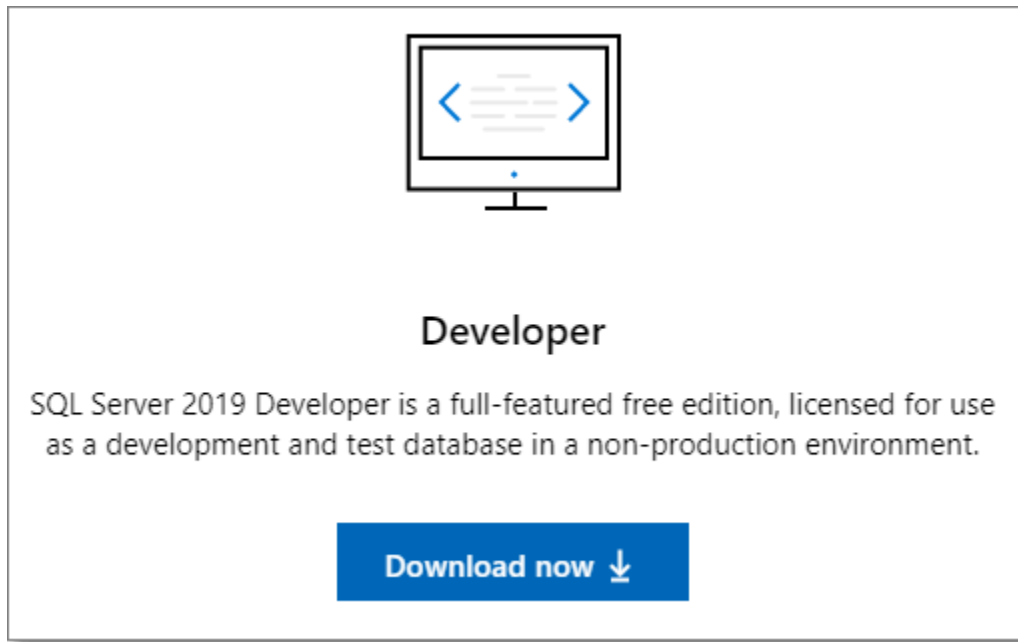
Install SQL Server

Figure 21: SQL Server

The application requires a database to store the data. Download and install the free SQL Server 2019 Developer Edition from the following link: https://www.microsoft.com/en-us/sql-server/sql-server-downloads.

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 (or later), from: https://dotnet.microsoft.com/download/dotnet-core.

- Install Visual Studio 2019 version 16.4 (or later), with the ASP.NET and web development workload from: https://visualstudio.microsoft.com/vs/.

📝 Note: The requirements to create applications using Blazor are constantly evolving. For the latest requirements, see the following link: https://docs.microsoft.com/en-us/aspnet/core/blazor/get-started.

> 📝 **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. See** https://docs.microsoft.com/en-us/dotnet/core/install/sdk?pivots=os-windows.
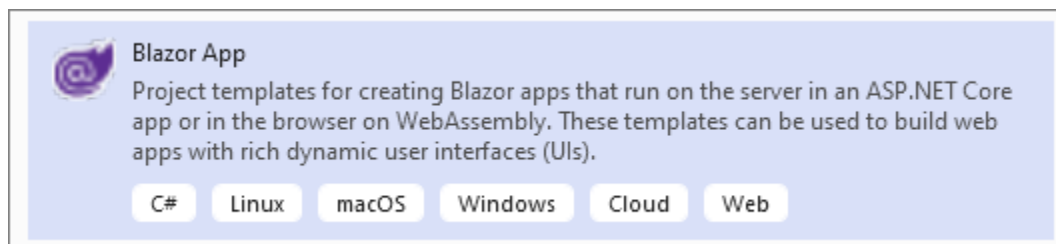
Create the project

Figure 22: Blazor App

Open **Visual Studio**, select **Create a New Project**, select **Blazor App**, then click **Next**.

Enter *SyncfusionHelpDesk* for the **Project name** and click **Create**.
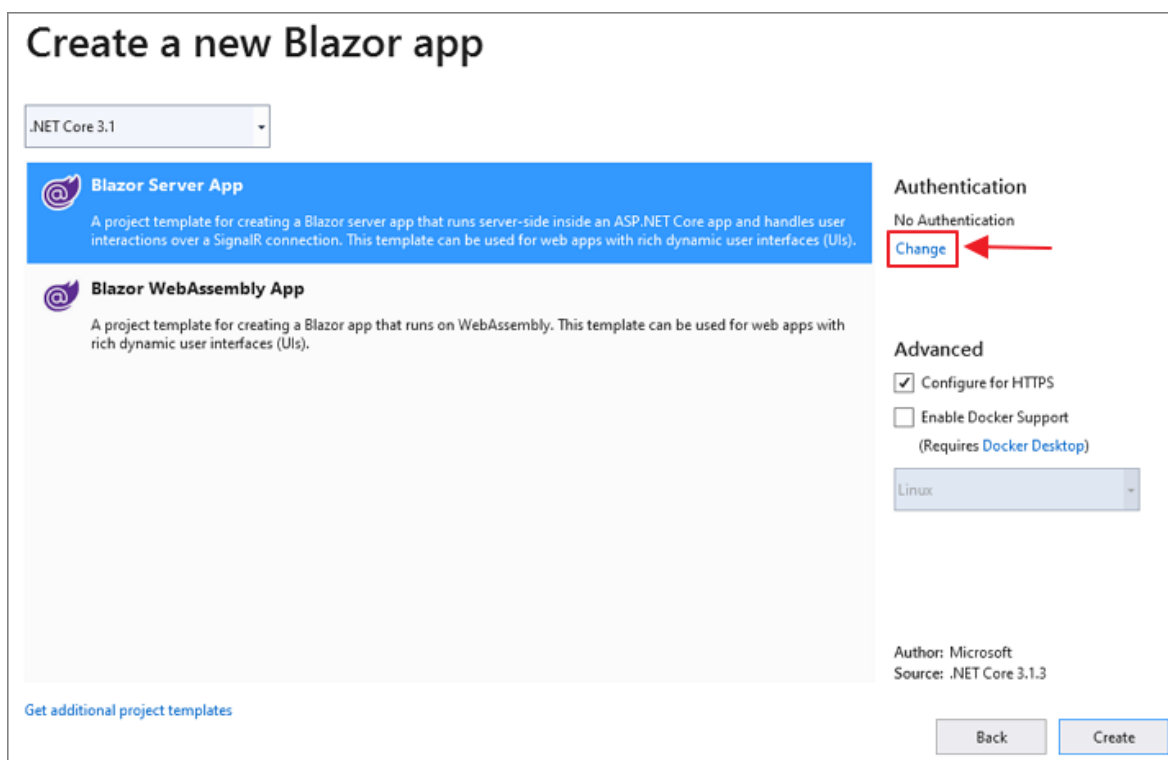
Figure 23: Change Authentication

In the **Create a new Blazor app** dialog, click the **Change** link under **Authentication**.
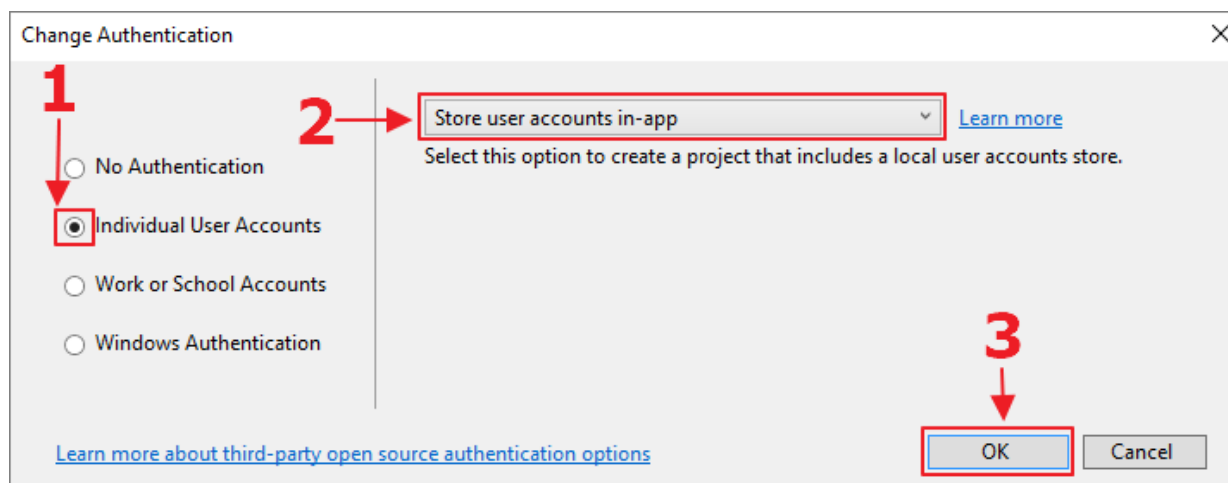
Figure 24: Set Authentication

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

Select **OK**. Then in the **Create a new Blazor app** dialog, click **Create**.
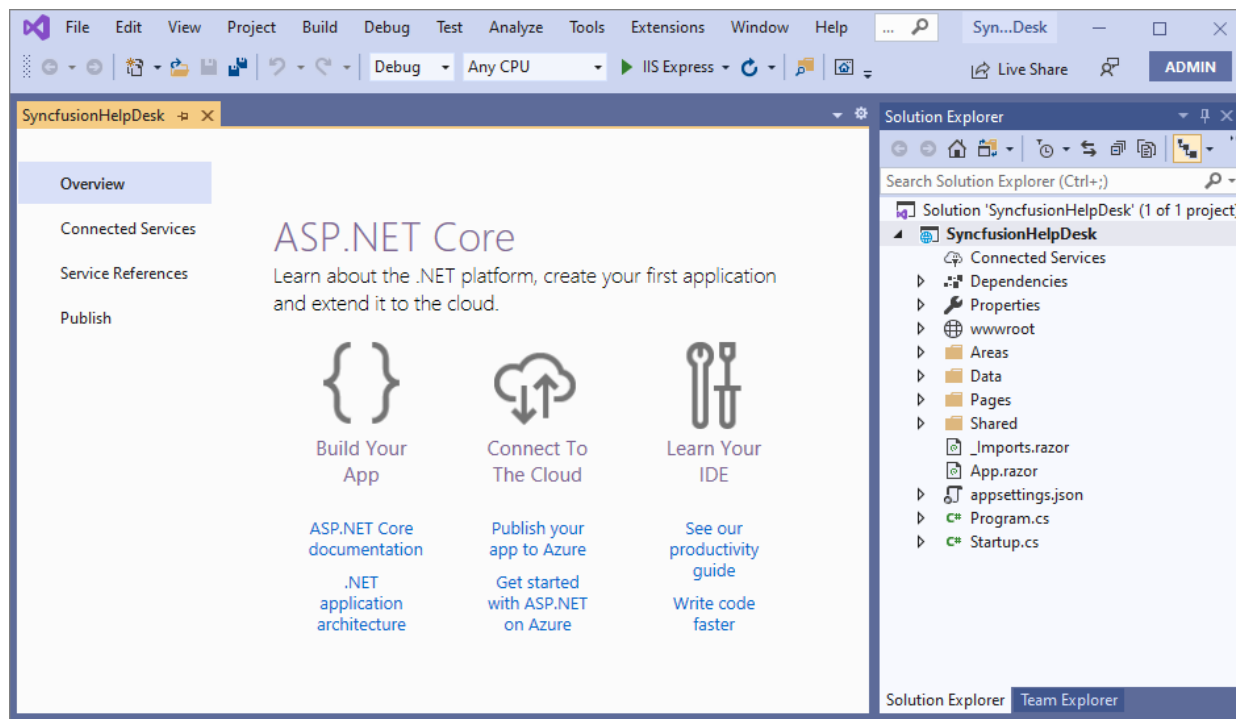
Figure 25: In Visual Studio

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

Create the database

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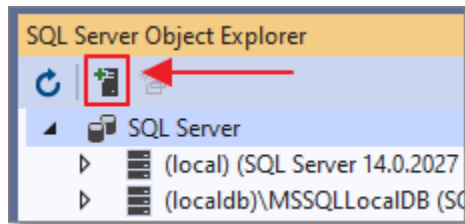
Figure 26: Add SQL Server

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.*
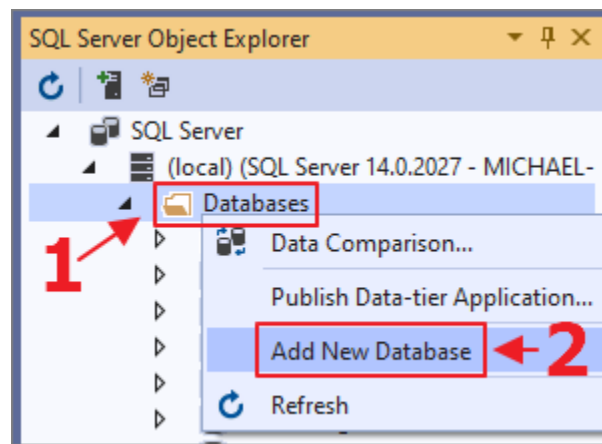
Figure 27: Add New Database

Expand the tree node for your SQL Server, then right-click on **Databases** and select **Add New Database**.

Name the database **SyncfusionHelpDesk**.
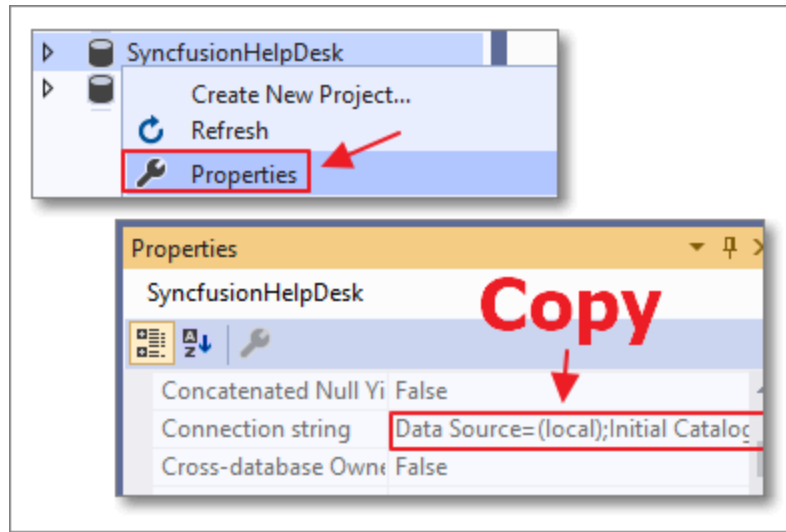
Figure 28: Copy Connection String

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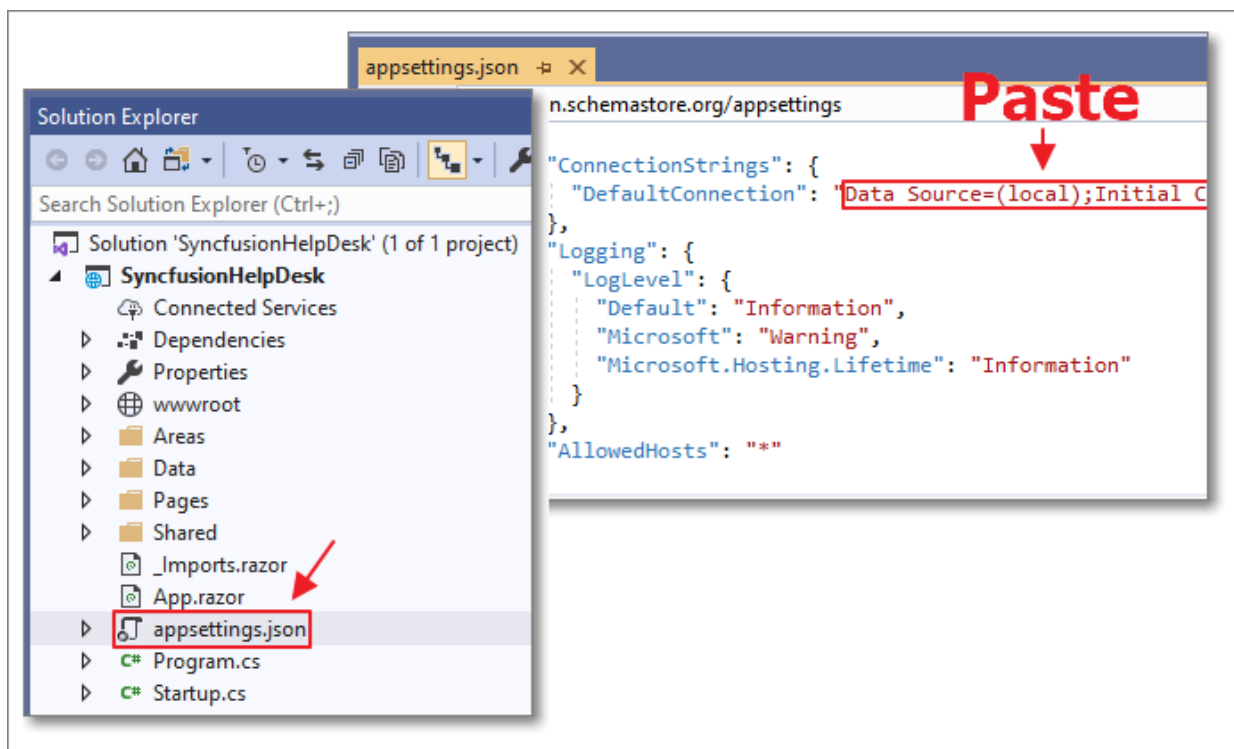
Figure 29: Paste Connection String

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

Save and close the file.

Enable role management

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

Open the **Startup.cs** file and replace the following code.

Code Listing 9: Original Identity Code

```
    services.AddDefaultIdentity<IdentityUser>(
        options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<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<IdentityUser>()
        .AddRoles<IdentityRole>() // Enable roles
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

In **Visual Studio**, select the **F5** key to run the application and open it in your web browser.
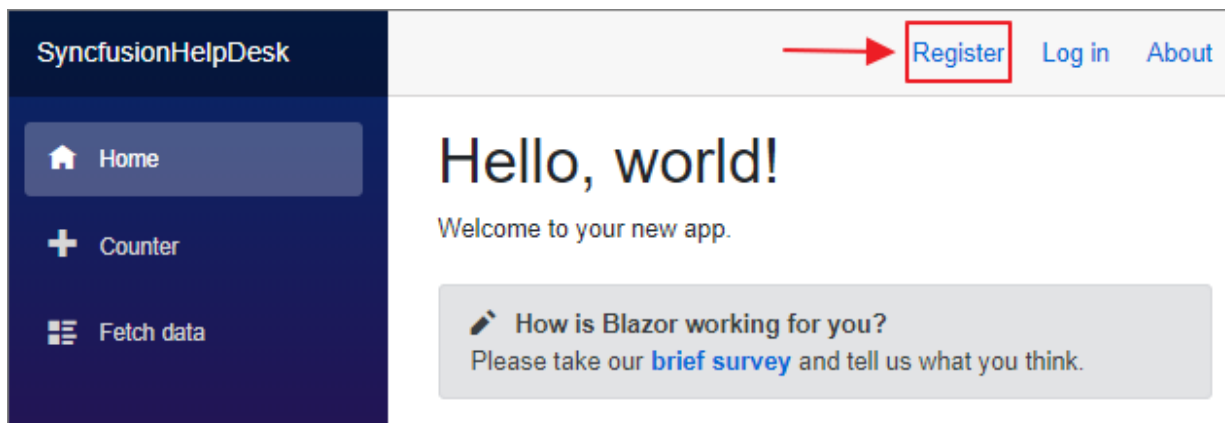
Figure 30: Register User

Click the **Register** link and create a user with the email: **Administrator@Email**.

You will then see the following page.

A database operation failed while processing the request.

SqlException: Invalid object name 'AspNetUsers'.

Applying existing migrations for ApplicationDbContext may resolve this issue

There are migrations for ApplicationDbContext that have not been applied to the database

- 00000000000000_CreateIdentitySchema

Apply Migrations

In Visual Studio, you can use the Package Manager Console to apply pending migrations to the database:

PM> Update-Database

Alternatively, you can apply pending migrations from a command prompt at your project directory:

> dotnet ef database update

Figure 31: Apply Migrations

Click **Apply Migrations** and, after a few moments, you will see a message telling you 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 **Administrator@Email** account is logged in.

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

Create the administrator

We will now create code that will programmatically create an administrator role and add the Administrator@Email account to the administrator role.
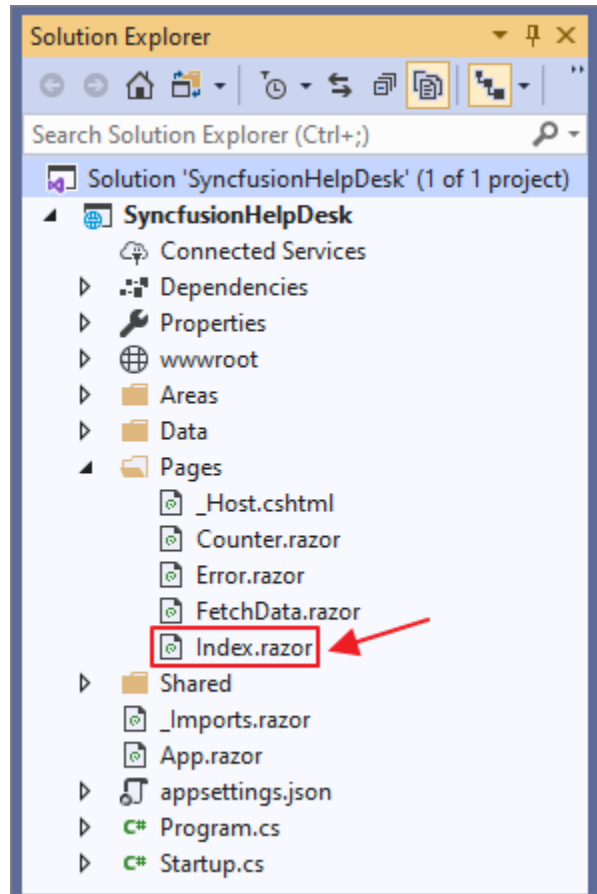
Figure 33: Open Index.razor Page

Open the **Index.razor** page (this is the home page of the application) and replace all the code with the following code.

Code Listing 11: Original Identity Code

```
@page "/"
@using System.Security.Claims;
@using Microsoft.AspNetCore.Identity;
@using Microsoft.AspNetCore.Components.Authorization;
@inject UserManager<IdentityUser> _UserManager
@inject RoleManager<IdentityRole> _RoleManager

@if (CurrentUser.IsInRole(ADMINISTRATION_ROLE))
{
    <p>You are an Administrator named: <b>@CurrentUser.Identity.Name</b></p>
}
```

```
@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }
    ClaimsPrincipal CurrentUser = new ClaimsPrincipal();
    string ADMINISTRATION_ROLE = "Administrators";

    protected override async Task OnInitializedAsync()
    {
        // Ensure there is an ADMINISTRATION_ROLE
        var RoleResult = await _RoleManager.FindByNameAsync(ADMINISTRATION_ROLE);
        if (RoleResult == null)
        {
            // Create ADMINISTRATION_ROLE role.
            await _RoleManager.CreateAsync(new IdentityRole(ADMINISTRATION_ROLE));
        }

        // Try to get the administrator account.
        var user = await _UserManager.FindByNameAsync("Administrator@Email");

        // Administrator may not be created yet.
        if (user != null)
        {
            // Is administrator account in the administrator role?
            var UserResult =
                await _UserManager.IsInRoleAsync(user, ADMINISTRATION_ROLE);

            if (!UserResult)
            {
                // Put admin in administrator role.
                await _UserManager.AddToRoleAsync(user, ADMINISTRATION_ROLE);
            }
        }

        // Get the current user.
        // Note: User may not be logged in.
        CurrentUser = (await authenticationStateTask).User;
    }

}
```

Save the page and run the application.

The home page of the application displays as an empty page. However, we can click the login link to log in to the application.
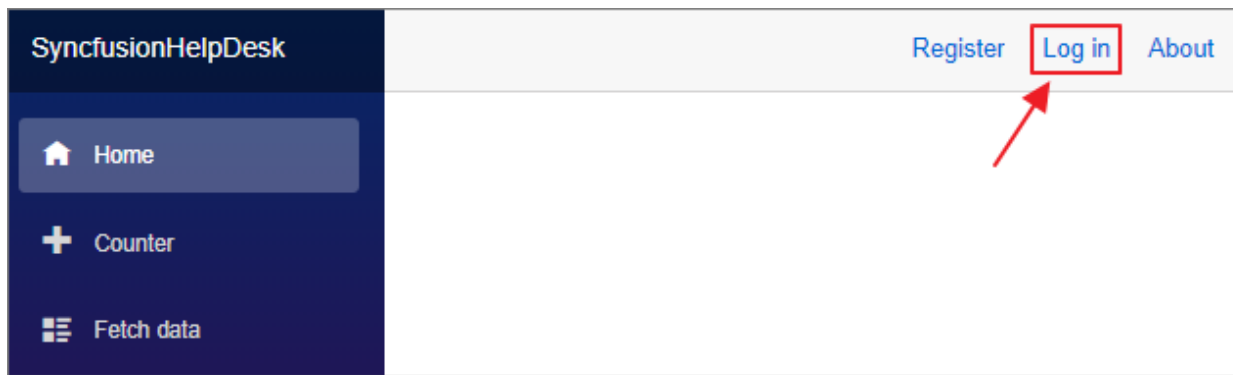
Figure 34: Log in to the Application

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.
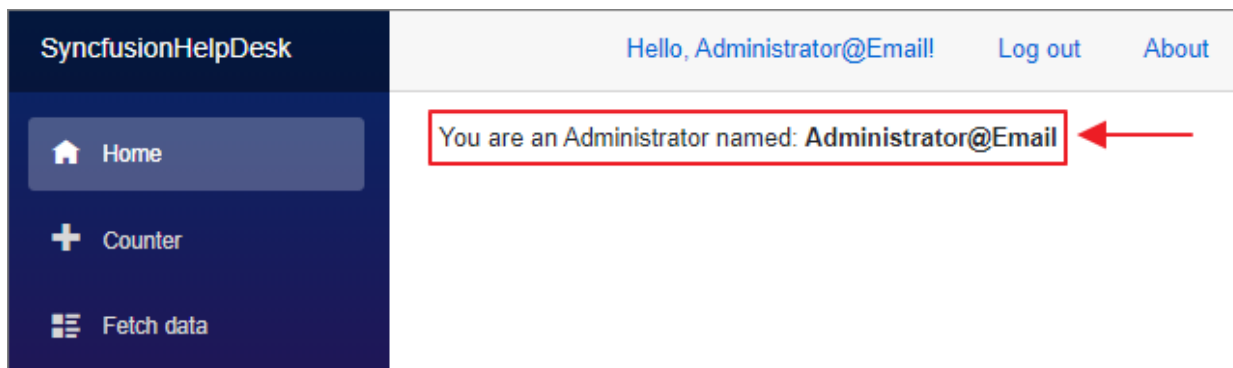
Figure 35: Administrator Created

Chapter 4 Explore the Project

In this chapter, we will explore the Blazor project we just created.
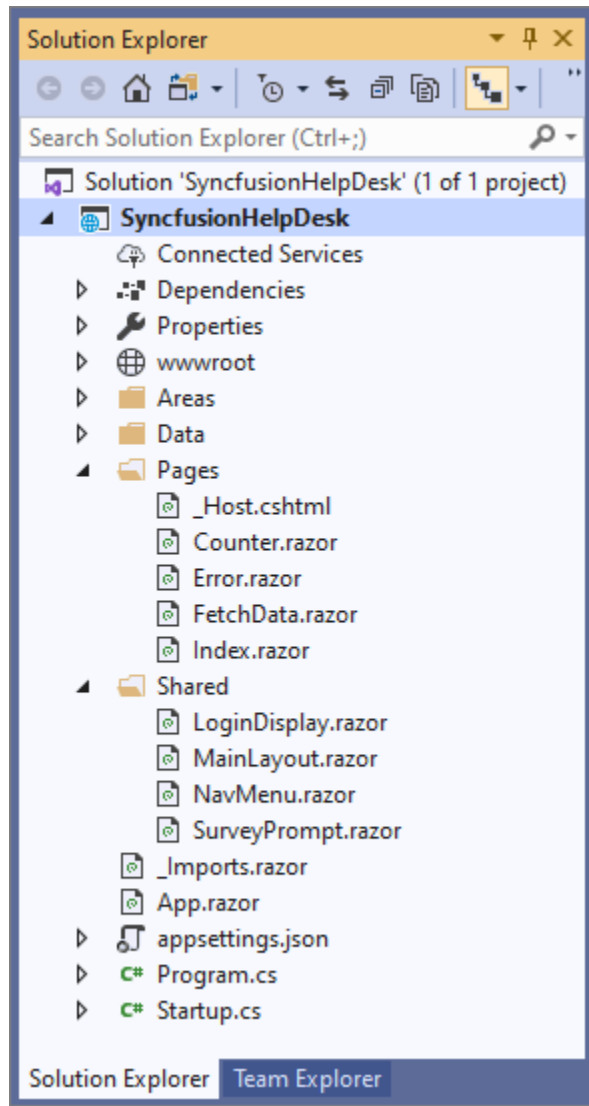
Figure 36: Explore the Project

Startup

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

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

The Startup class calls the MapBlazorHub method that configures the SignalR endpoint required for server-side Blazor operation. It also calls the method MapFallbackToPage("/_Host") that sets up the _Host.cshtml page as the root page of the application.

Code Listing 13: Startup.cs

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});
```

The _Host.cshtml page specifies that the App component, contained in the App.razor file, is to be rendered as the root component of the application.

Code Listing 14: _Host.cshtml Page

```
<body>
    <app>
        <component type="typeof(App)"
                   render-mode="ServerPrerendered" />
    </app>

...

    <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

The page also loads the blazor.webassembly.js file, which sets up the SignalR connection between the user's web browser and the server-side code.

Routing

The routing of page requests in the application is configured in the **App.razor** control.

Code Listing 15: App.razor

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
```

```
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
DefaultLayout="@typeof(MainLayout)" />
        </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 16: 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 control inherits from LayoutComponentBase and will inject the content of the Razor page to which the user is navigating 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 implement advanced functionality easily 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: For the latest requirements to use Syncfusion, see the following link: https://blazor.syncfusion.com/documentation/system-requirements/.

Install NuGet packages
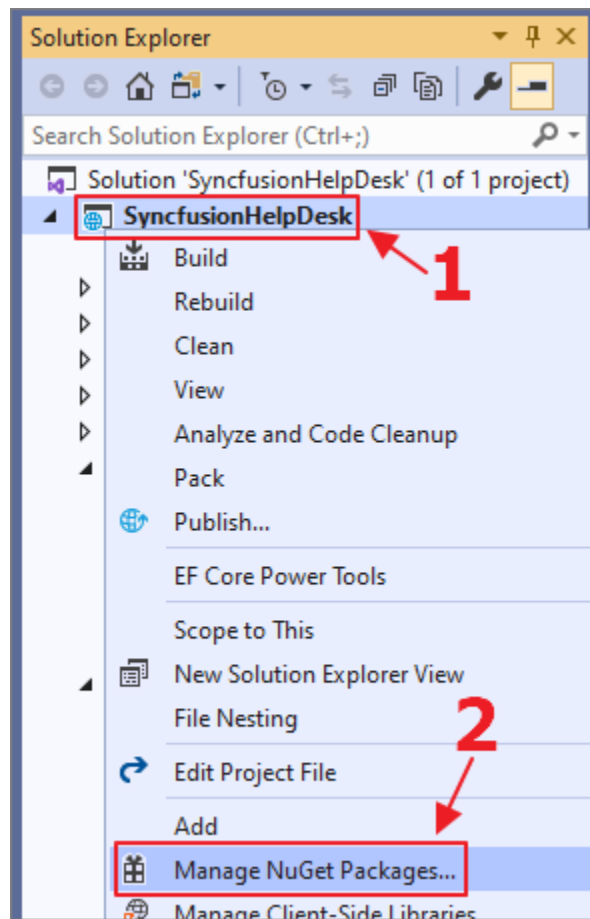
Figure 37: NuGet Packages

Right-click on the project node and select Manage NuGet Packages.
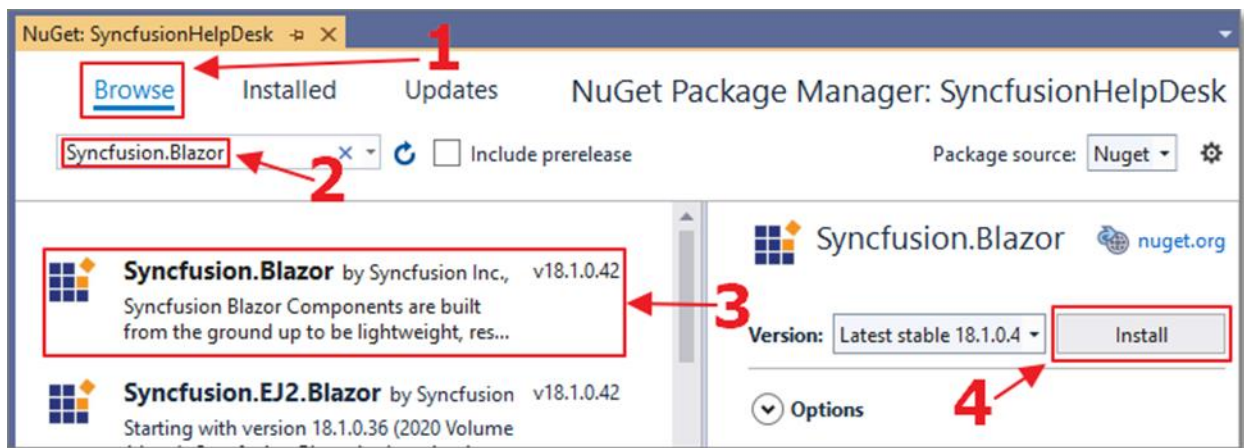
Figure 38: Install NuGet Packages

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

- Syncfusion.Blazor

- System.Text.Json

Additional configuration

Open the _Imports.razor file and add the following.

Code Listing 17: _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 Startup.cs file and add the following using statement.

Code Listing 18: Startup.cs Using Statement

```
using Syncfusion.Blazor;
```

Also, add the following to the ConfigureServices section.

Code Listing 19: Startup.cs AddSyncfusionBlazor

```
            // Syncfusion support
            services.AddSyncfusionBlazor();
```

Add the following to the <head> element of the _Host.cshtml page.

Code Listing 20: _Host.cshtml

```
<link href="https://cdn.syncfusion.com/blazor/18.1.42/styles/material.css"
rel="stylesheet" />
```

Note: When you run the application, you will see a message:

"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 on the database and select **New Query**.
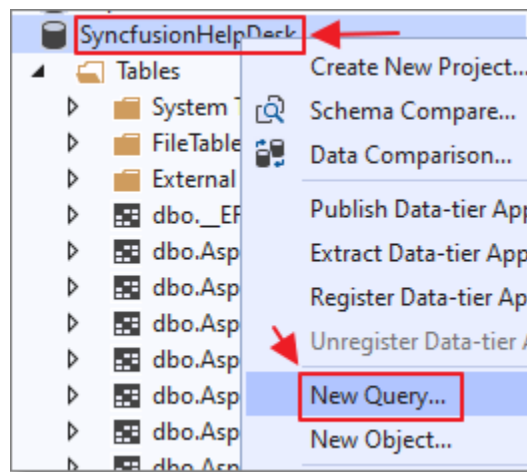
Figure 39: New Query

Enter the following script.

Code Listing 21: SQL Script

```sql
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.
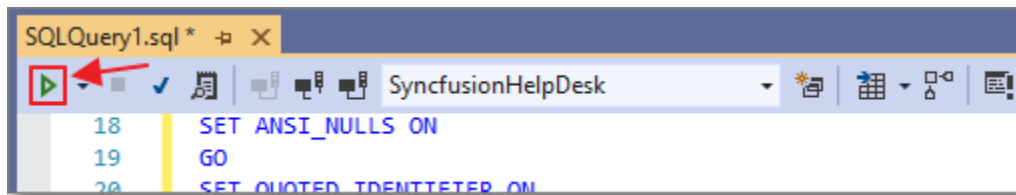
Figure 40: Click Execute

Close the SQLQuery1.sql window and refresh the view of the tables in the database.
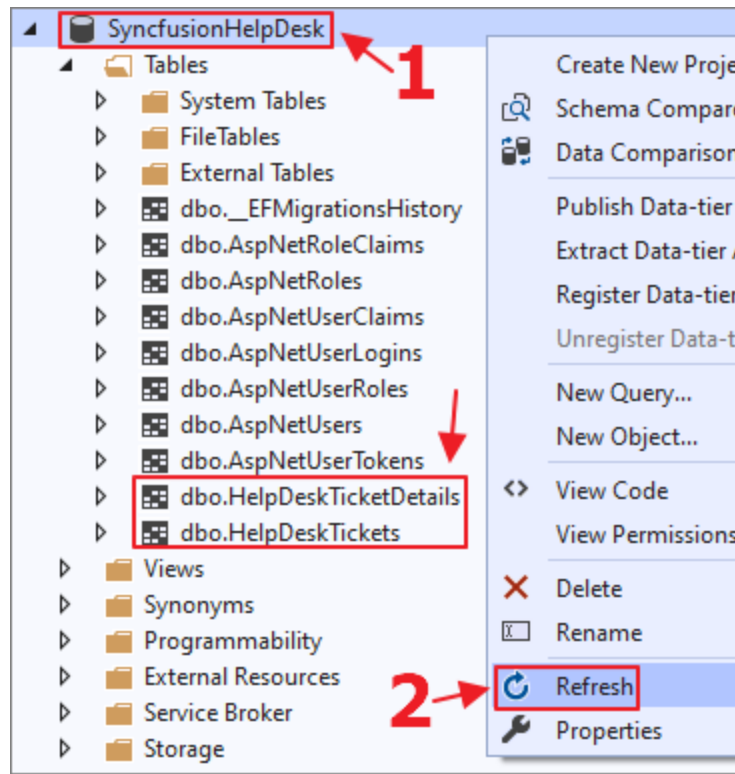
Figure 41: Refresh Database

You will see that the new tables have been added.
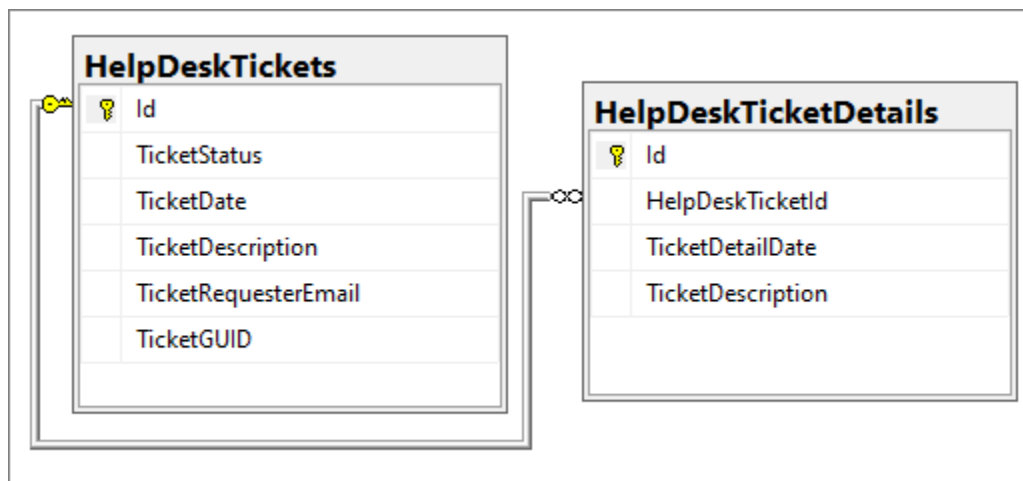
Figure 42: Database Diagram

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

Create the DataContext (using EF Core tools)

We will now create the DataContext code that will allow the data service, created in the following steps, to communicate with the database tables we just added.

Figure 43: EF Core Power Tools

Install EF Core Power Tools from the following link:
https://marketplace.visualstudio.com/items?itemName=ErikEJ.EFCorePowerTools.
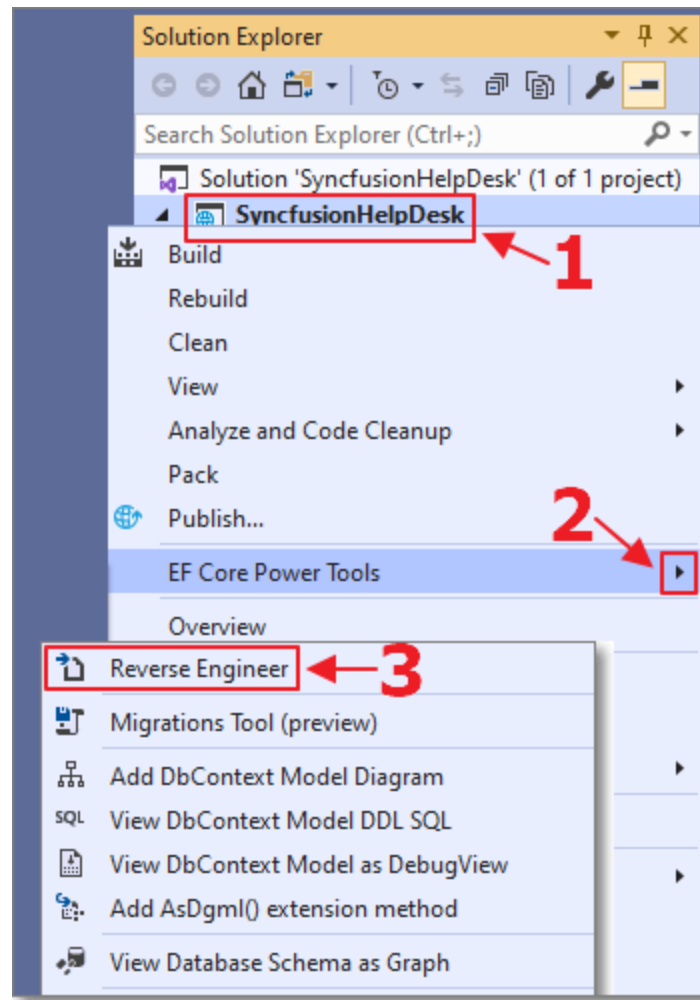
Figure 44: EF Core Power Tools

Right-click on the project node in the Solution Explorer and select **EF Core Power Tools**, then **Reverse Engineer**.
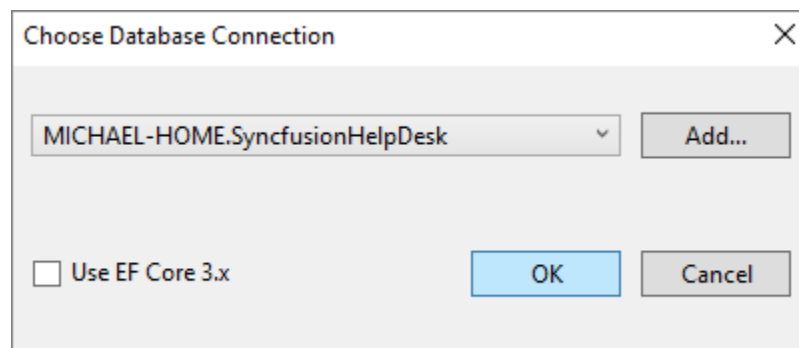
Figure 45: Create Connection

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**.
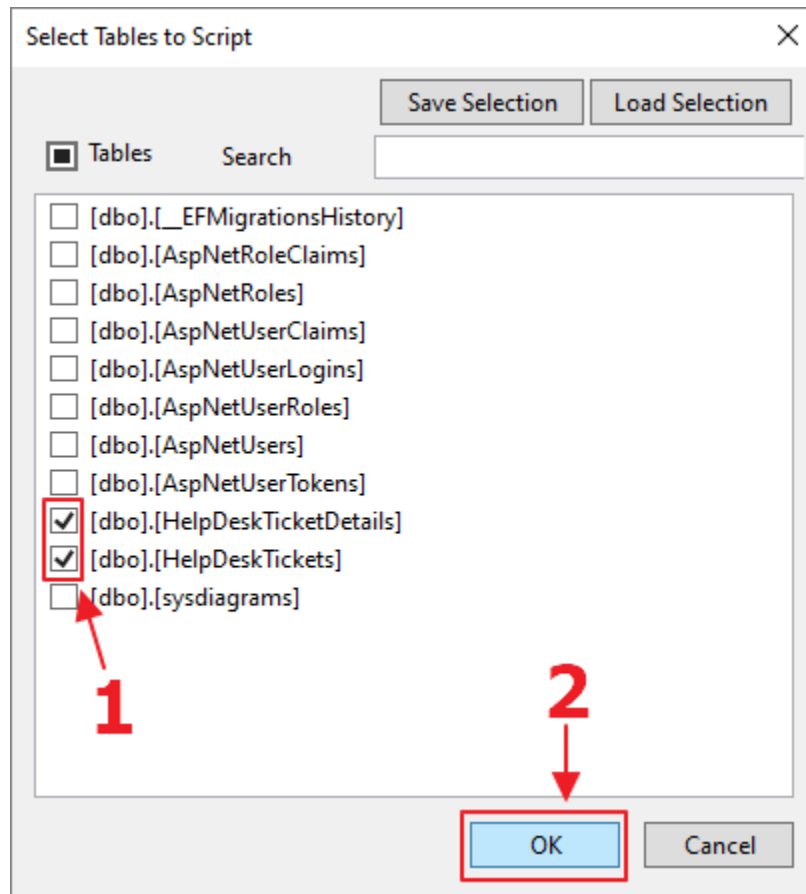
Figure 46: Select Tables

Select the **HelpDeskTicketDetails** table and the **HelpDeskTickets** table and click **OK**.
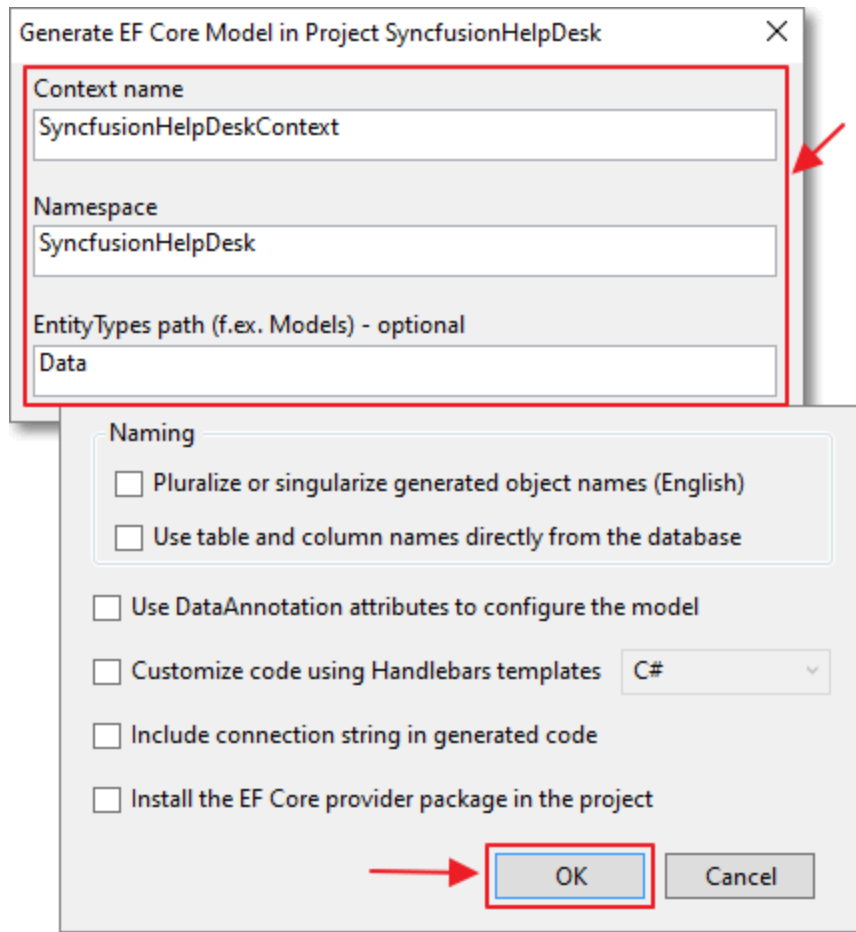
Figure 47: Configure DataContext

Set the following values:

- Context name: SyncfusionHelpDeskContext

- Namespace: SyncfusionHelpDesk

- EntityTypes path: Data

Click **OK**.
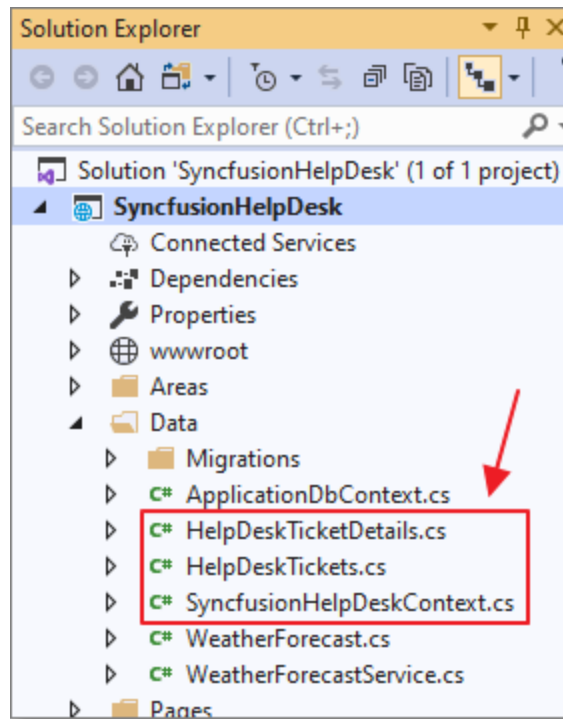
Figure 48: DataContext Created

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 code to the ConfigureServices section.

Code Listing 22: Database Connection

```
        // To access HelpDesk tables.
        services.AddDbContext<SyncfusionHelpDeskContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
```

Save the file.

Select **Build**, and then **Rebuild Solution**.

The application should build without any errors.

Create the SyncfusionHelpDeskService

We will now create the service that will provide all the remaining data access methods we will need for the application.

In the Data folder, add a new file called SyncfusionHelpDeskService.cs with the following code.

Code Listing 23: SyncfusionHelpDeskService

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

namespace SyncfusionHelpDesk.Data
{
    public class SyncfusionHelpDeskService
    {
        private readonly SyncfusionHelpDeskContext _context;

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

        public IQueryable<HelpDeskTickets>
            GetHelpDeskTickets()
        {
            // Return all HelpDesk tickets as IQueryable.
            // SfGrid will use this to only pull records
            // for the page that it is currently displaying.
            // Note: AsNoTracking() is used because it is
            // quicker to execute, and we do not need
            // Entity Framework change tracking at this point.
            return _context.HelpDeskTickets.AsNoTracking();
        }

        public async Task<HelpDeskTickets>
            GetHelpDeskTicketAsync(string HelpDeskTicketGuid)
        {
            // Get the existing record.
            var ExistingTicket = await _context.HelpDeskTickets
                .Include(x => x.HelpDeskTicketDetails)
                .Where(x => x.TicketGuid == HelpDeskTicketGuid)
                .AsNoTracking()
                .FirstOrDefaultAsync();

            return ExistingTicket;
        }

        public Task<HelpDeskTickets>
            CreateTicketAsync(HelpDeskTickets newHelpDeskTickets)
```

```csharp
    {
        try
        {
            // Add a new help desk ticket.
            _context.HelpDeskTickets.Add(newHelpDeskTickets);
            _context.SaveChanges();

            return Task.FromResult(newHelpDeskTickets);
        }
        catch (Exception ex)
        {
            DetachAllEntities();
            throw;
        }
    }

    public Task<bool>
        UpdateTicketAsync(
        HelpDeskTickets UpdatedHelpDeskTickets)
    {
        try
        {
            // Get the existing record.
            var ExistingTicket =
                _context.HelpDeskTickets
                .Where(x => x.Id == UpdatedHelpDeskTickets.Id)
                .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)
                        {
                            // Create new HelpDeskTicketDetails record.
                            HelpDeskTicketDetails newHelpDeskTicketDetails =
                                new HelpDeskTicketDetails();
```

```csharp
                            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);
        }
        catch (Exception ex)
        {
            DetachAllEntities();
            throw ex;
        }
    }

    public Task<bool>
        DeleteHelpDeskTicketsAsync(
        HelpDeskTickets DeleteHelpDeskTickets)
    {
        // Get the existing record.
        var ExistingTicket =
            _context.HelpDeskTickets
            .Include(x => x.HelpDeskTicketDetails)
            .Where(x => x.Id == DeleteHelpDeskTickets.Id)
            .FirstOrDefault();

        if (ExistingTicket != null)
        {
            // Delete the help desk ticket.
            _context.HelpDeskTickets.Remove(ExistingTicket);
            _context.SaveChanges();
        }
        else
        {
            return Task.FromResult(false);
        }

        return Task.FromResult(true);
    }

    // Utility
```

```
        #region public void DetachAllEntities()
        public void DetachAllEntities()
        {
            // When we have an error, we need
            // to remove EF Core change tracking.
            var changedEntriesCopy = _context.ChangeTracker.Entries()
                .Where(e => e.State == EntityState.Added ||
                            e.State == EntityState.Modified ||
                            e.State == EntityState.Deleted)
                .ToList();

            foreach (var entry in changedEntriesCopy)
                entry.State = EntityState.Detached;
        }
        #endregion

    }
}
```

Register the SyncfusionHelpDeskService

Finally, we need to register this service so that we can make it available to our code pages.

Open the Startup.cs file and add the following code to the ConfigureServices section.

Code Listing 24: SyncfusionHelpDeskService

```
        // To access SyncfusionHelpDeskService
        services.AddScoped<SyncfusionHelpDeskService>();
```

We will later inject this service into our code pages using `OwningComponentBase` to scope the service to a single webpage.

Chapter 7 Creating New Tickets

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

OwningComponentBase

We need to use the **SyncfusionHelpDeskService**, created in the previous chapter, to access the database. However, with Blazor Server, a service will persist for the duration of the time a user is connected to the application, once it's loaded. Meaning, even when they go to another page, the service will persist in memory.

To scope the service to only the current page, we use **OwningComponentBase**.

Open the index.razor page and add the following under the **@page** directive.

Code Listing 25: OwningComponentBase

```
@using SyncfusionHelpDesk.Data;
@inherits OwningComponentBase<SyncfusionHelpDeskService>
```

The **@using** statement is required to support access to the types in the **SyncfusionHelpDeskService** service.

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. To enable this, add the following code markup.

Code Listing 26: Toast Control

```
<SfToast ID="toast_default"
         @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 27: 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, in the Data folder, create a new folder named Models, add the following class that will be bound to the form, and provide the validation attributes.

Code Listing 28: HelpDeskTicket.cs

```
using System;
using System.ComponentModel.DataAnnotations;

namespace SyncfusionHelpDesk.Data
{
    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; }
    }
}
```

Syncfusion Blazor controls

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

- Syncfusion TextBox: A control that allows us to gather data from the user and later display existing data.

- Syncfusion DatePicker: A control that allows the user to enter a date value and later display an existing date value.

- Syncfusion Dropdown List: 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 display an existing selected value, also.

The Dropdown List control requires a data collection for the display options. To enable this, in the Models folder, add the following class.

Code Listing 29: HelpDeskStatus.cs

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

namespace SyncfusionHelpDesk.Data
{
    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" },
        };
    }
}
```

New ticket form

We will now add the code to display the form.

Add the following markup.

Code Listing 30: 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.

Code Listing 31: HelpDeskTicket Property

```
    // Global property for the help desk ticket.
    HelpDeskTicket objHelpDeskTicket =
    new HelpDeskTicket() { TicketDate = DateTime.Now };
```

Finally, add the following code to insert the data into the database by calling the **CreateTicketAsync** method of the **SyncfusionHelpDeskService** when the form has successfully passed validation.

Code Listing 32: HandleValidSubmit

```
    public async Task HandleValidSubmit(EditContext context)
    {
        try
        {
            // Create a HelpDeskTickets.
            HelpDeskTickets NewHelpDeskTickets =
                new HelpDeskTickets();

            // Set the values to the values entered
            // in the form.
            NewHelpDeskTickets.TicketDate =
                objHelpDeskTicket.TicketDate;
            NewHelpDeskTickets.TicketDescription =
                objHelpDeskTicket.TicketDescription;
            NewHelpDeskTickets.TicketRequesterEmail =
                objHelpDeskTicket.TicketRequesterEmail;
            NewHelpDeskTickets.TicketStatus =
                objHelpDeskTicket.TicketStatus;
            // Create a new GUID for this help desk ticket.
            NewHelpDeskTickets.TicketGuid =
                System.Guid.NewGuid().ToString();

            // Save the new help desk ticket.
            var result =
                @Service.CreateTicketAsync(NewHelpDeskTickets);

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

            // Show the Toast.
            ToastContent = "Saved!";
            await Task.Delay(100);
            await this.ToastObj.Show();
```

```
        }
        catch (Exception ex)
        {
            ToastContent = ex.GetBaseException().Message;
            await this.ToastObj.Show();
        }
    }
}
```

Test the form

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

Figure 49: New Help Desk Ticket Form

The user is presented with a New Help Desk Ticket form.
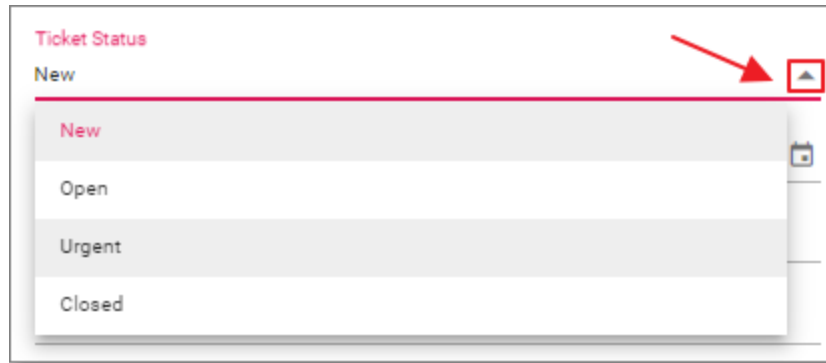
Figure 50: Select Ticket Status

The user can select a ticket status using the Dropdown List control.
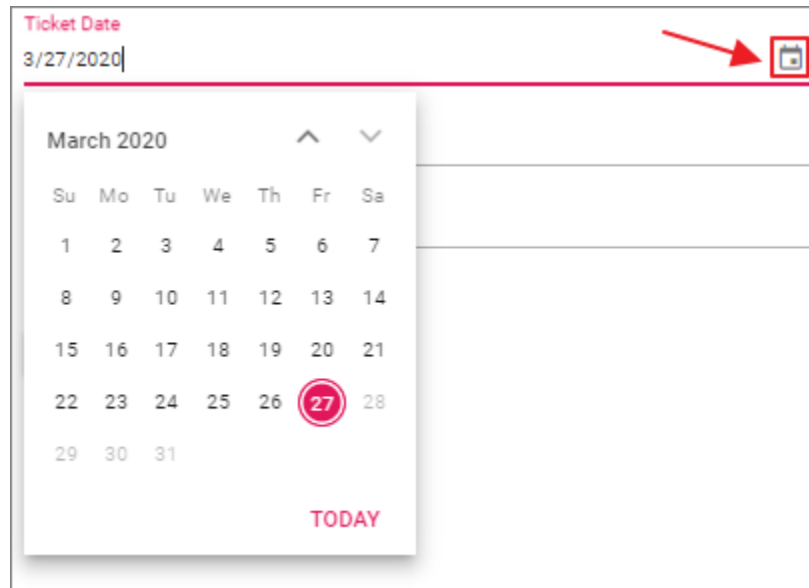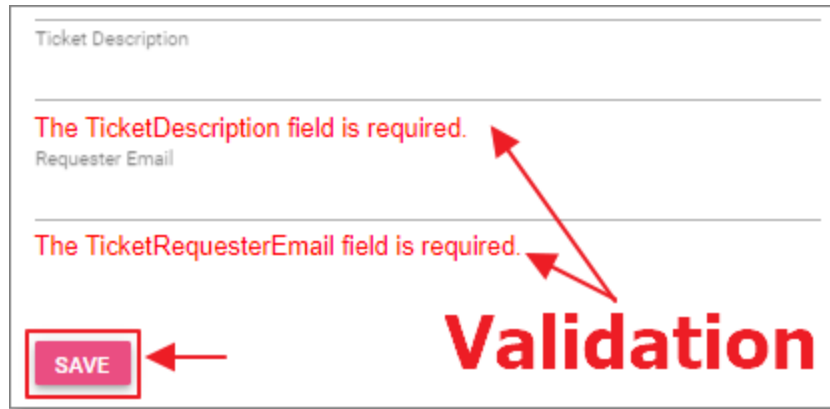
Figure 51: Select Ticket Date
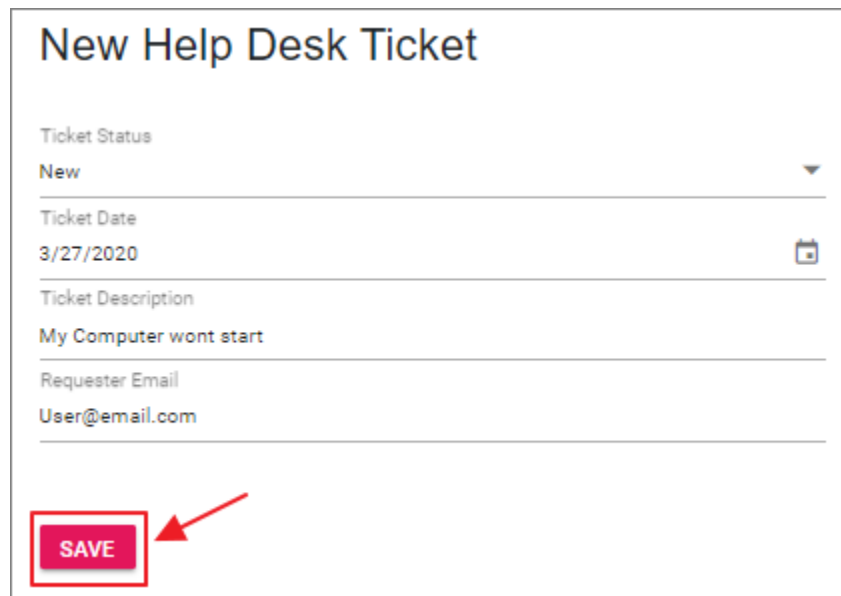
The user can select a Ticket Date using the DatePicker control.

Figure 52: Validation Errors

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

Figure 53: Save Ticket

With a properly completed form, the user can click **SAVE** to save the data.

Figure 54: Toast Confirmation

They will see a brief confirmation message by the Toast control.
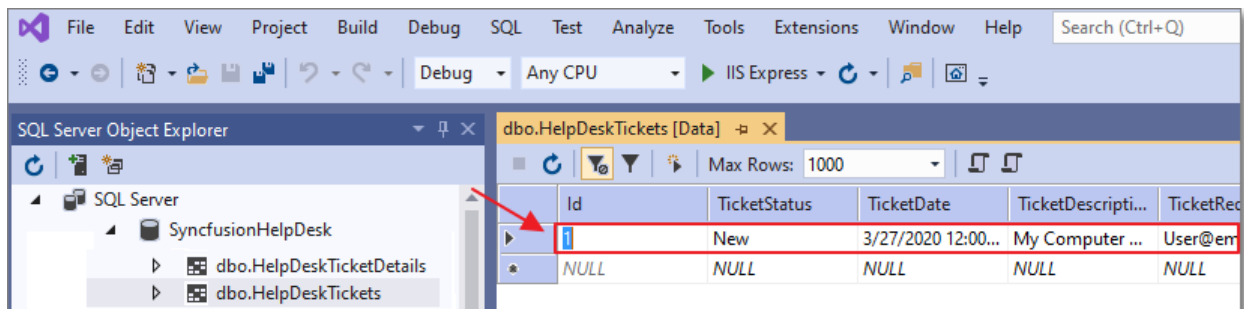
Figure 55: View Data in the Database

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 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
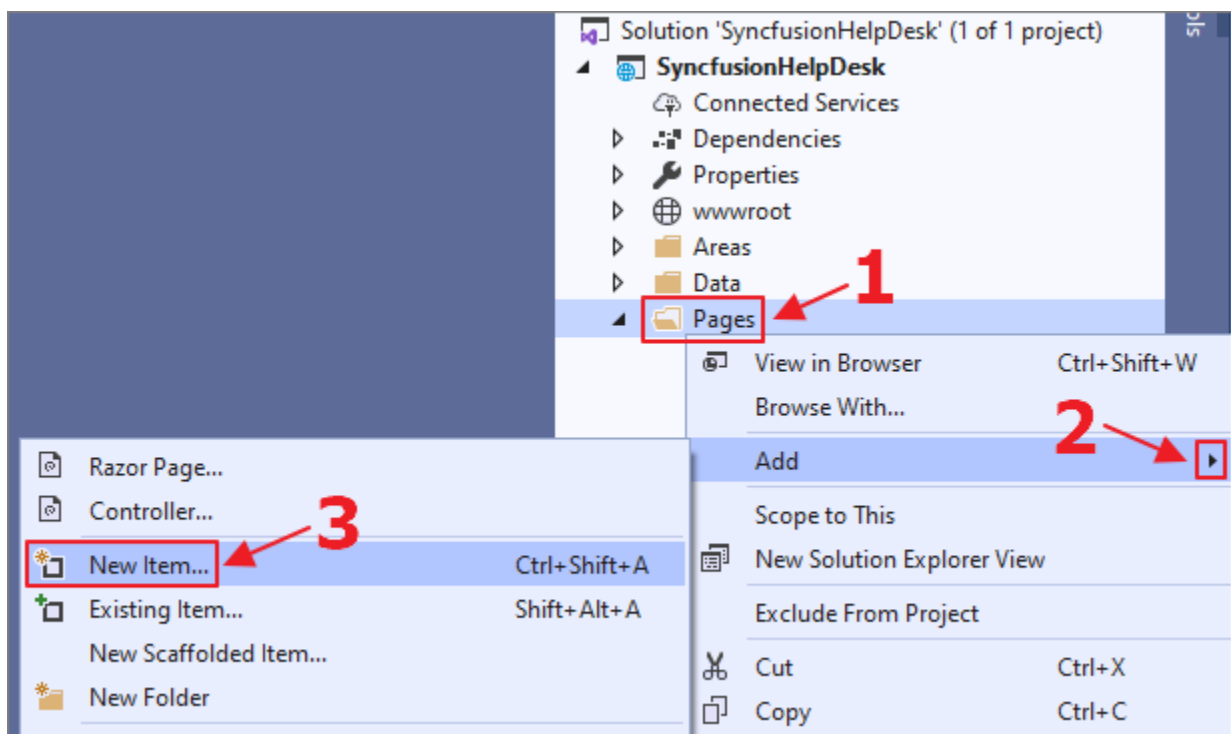
Figure 56: New Item

Add the Administration page to the project by right-clicking on the **Pages** folder and selecting **Add**, then **New Item**.
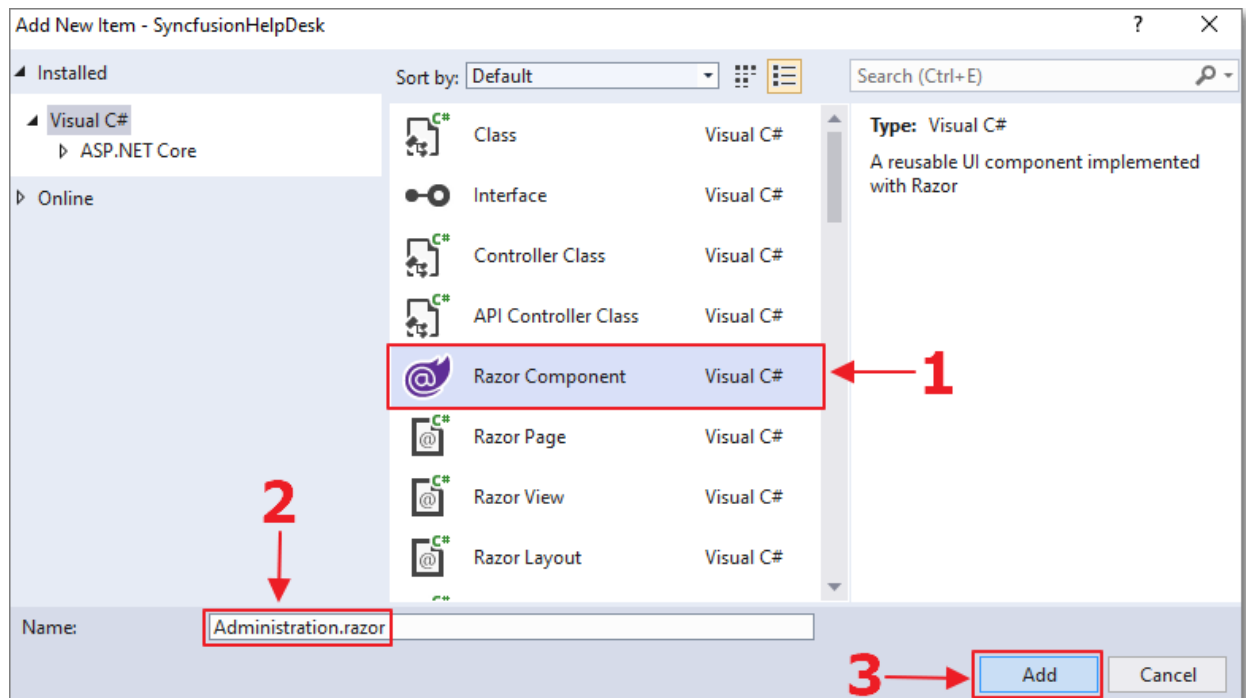
Figure 57: Add New Item

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

Use the following code for the Razor control.

Code Listing 33: Administration.razor

```
@page "/administration"
@using SyncfusionHelpDesk.Data;

@inherits OwningComponentBase<SyncfusionHelpDeskService>

@*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 and remove the following code (for the links to the counter and fetchdata pages).

Code Listing 34: 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 35: 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*.
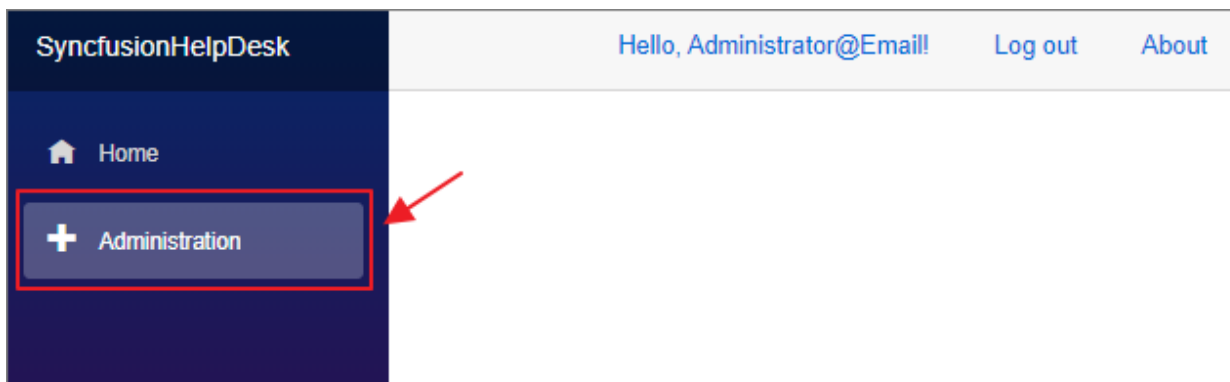
Figure 58: Administration Link

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 (that is currently blank).

Using Syncfusion Data Grid

The Syncfusion Data Grid (programmatically called **SfGrid**) is a control that 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 36: Data Grid

```
<div>
    <div id="target" style="height: 500px;">
        <SfGrid ID="Grid"
                    DataSource="@colHelpDeskTickets"
                    AllowPaging="true"
                    AllowSorting="true"
                    AllowResizing="true"
                    AllowReordering="true">
            <GridPageSettings PageSize="5"></GridPageSettings>
            <GridEvents CommandClicked="OnCommandClicked"
                        TValue="HelpDeskTickets">
            </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(HelpDeskTickets.Id)
                            HeaderText="ID #" TextAlign="@TextAlign.Left"
                            Width="70">
                </GridColumn>
                <GridColumn Field=@nameof(HelpDeskTickets.TicketStatus)
                            HeaderText="Status" TextAlign="@TextAlign.Left"
                            Width="80">
                </GridColumn>
                <GridColumn Field=@nameof(HelpDeskTickets.TicketDate)
                            HeaderText="Date" TextAlign="@TextAlign.Left"
                            Width="80">
                </GridColumn>
```
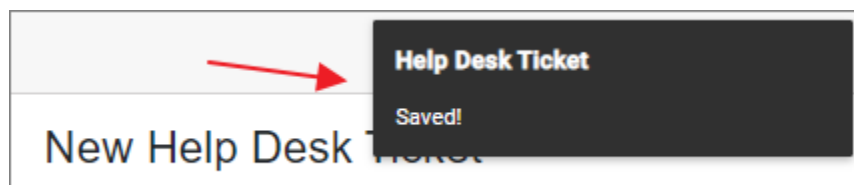
```
                <GridColumn Field=@nameof(HelpDeskTickets.TicketDescription)
                            HeaderText="Description" TextAlign="@TextAlign.Left"
                            Width="150">
                </GridColumn>
                <GridColumn Field=@nameof(HelpDeskTickets.TicketRequesterEmail)
                            HeaderText="Requester" TextAlign="@TextAlign.Left"
                            Width="150">
                </GridColumn>
            </GridColumns>
        </SfGrid>
    </div>
</div>
```

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

Code Listing 37: Data Grid Code

```
    public IQueryable<HelpDeskTickets> colHelpDeskTickets { get; set; }
    private HelpDeskTickets SelectedTicket = new HelpDeskTickets();

    protected override void OnInitialized()
    {
        // GetHelpDeskTickets returns IQueryable that the
        // SfGrid will use to only pull records for the
        // page that is currently selected.
        colHelpDeskTickets = @Service.GetHelpDeskTickets();
    }

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

The **OnInitialized** method is the first method to run when the page loads. This method populates the **colHelpDeskTickets** collection that is bound to the **DataSource** property of the Data Grid.

Figure 59: 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 60: Sorting

Sorting is achieved by clicking on the column headers.

Figure 61: 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 Data Grid allows us to create custom command buttons using the **CommandClicked** property that we currently have wired up to the **OnCommandClicked** method.

We will use the **OnCommandClicked** method to display a pop-up, using the Syncfusion Dialog control, that 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 38: 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 39: Help Desk Ticket Property

```
// Global property for the help desk ticket.
HelpDeskTicket objHelpDeskTicket =
new HelpDeskTicket() { TicketDate = DateTime.Now };
```

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 40: Open Confirmation Dialog

```
public async void OnCommandClicked(
    CommandClickEventArgs<HelpDeskTickets> args)
{
    // Get the selected help desk ticket.
    SelectedTicket =
        await @Service.GetHelpDeskTicketAsync(args.RowData.TicketGuid);

    if (args.CommandColumn.ButtonOption.Content == "Delete")
    {
        // 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 in the dialog.

Code Listing 41: 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.

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

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

Code Listing 43: Data Grid Property

```
    SfGrid<HelpDeskTickets> 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 44: Delete Record

```
    public async void ConfirmDeleteYes()
    {
        // The user selected Yes to delete the
        // selected help desk ticket.
        // Delete the record.
        var result =
            await @Service.DeleteHelpDeskTicketsAsync(
                SelectedTicket);

        // Close the dialog.
        this.DeleteRecordConfirmVisibility = false;

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

When we run the application, we can click the DELETE button to open the dialog.
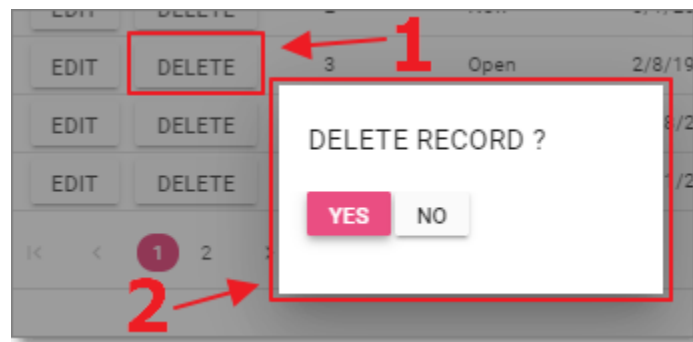
Figure 62: Delete Confirmation

Clicking the NO button will simply close the dialog. Clicking the YES button will delete the selected record and refresh the Data Grid.

EditTicket 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, create a new control called EditTicket.razor using the following code.

Code Listing 45: EditTicket.razor

```
@using System.Security.Claims;
@using SyncfusionHelpDesk.Data;
@using Syncfusion.Blazor.DropDowns
@inherits OwningComponentBase<SyncfusionHelpDeskService>
<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>
```

```razor
@if (SelectedTicket.HelpDeskTicketDetails != null)
{
    @if (SelectedTicket.HelpDeskTicketDetails.Count() > 0)
    {
        <table class="table">
            <thead>
                <tr>
                    <th>Date</th>
                    <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="@NewHelpDeskTicketDetail">
    </SfTextBox>
    <SfButton CssClass="e-small e-success"
               @onclick="AddHelpDeskTicketDetail">
        Add
    </SfButton>
}
<br />
@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

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

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

    string NewHelpDeskTicketDetail = "";

    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.
```

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

    private void AddHelpDeskTicketDetail()
    {
        // Create new HelpDeskTicketDetails record.
        HelpDeskTicketDetails NewHelpDeskTicketDetails =
            new HelpDeskTicketDetails();

        NewHelpDeskTicketDetails.HelpDeskTicketId =
            SelectedTicket.Id;

        NewHelpDeskTicketDetails.TicketDetailDate =
            DateTime.Now;

        NewHelpDeskTicketDetails.TicketDescription =
            NewHelpDeskTicketDetail;

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

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

Notice that this exposes a **SelectedTicket** parameter (of type **HelpDeskTickets**) 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 46: EditTicket Dialog

```
<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** property.

Also, add the following to the **@code** section to implement the functionality for the Save button on the dialog.

Code Listing 47: SaveTicket Method

```
    public async Task SaveTicket()
    {
        // Update the selected help desk ticket.
        var result =
            await @Service.UpdateTicketAsync(SelectedTicket);

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

        // Refresh the SfGrid
        // so the changes to the selected
        // help desk ticket are reflected.
        gridObj.Refresh();
    }
```

Finally, add the following code to the **OnCommandClicked** method (under the SelectedTicket line) to open the dialog when the EDIT button is clicked in a row in the Data Grid.

Code Listing 48: Open Dialog

```
        if (args.CommandColumn.ButtonOption.Content == "Edit")
        {
            // Open the Edit dialog.
            this.EditDialogVisibility = true;
            StateHasChanged();
        }
```

Figure 63: Edit Ticket Dialog

When we run the application, we can click the EDIT button 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 64: Edit Ticket

The help desk detail record will be added, but it will not be saved until SAVE is clicked.

Chapter 9 Sending Emails
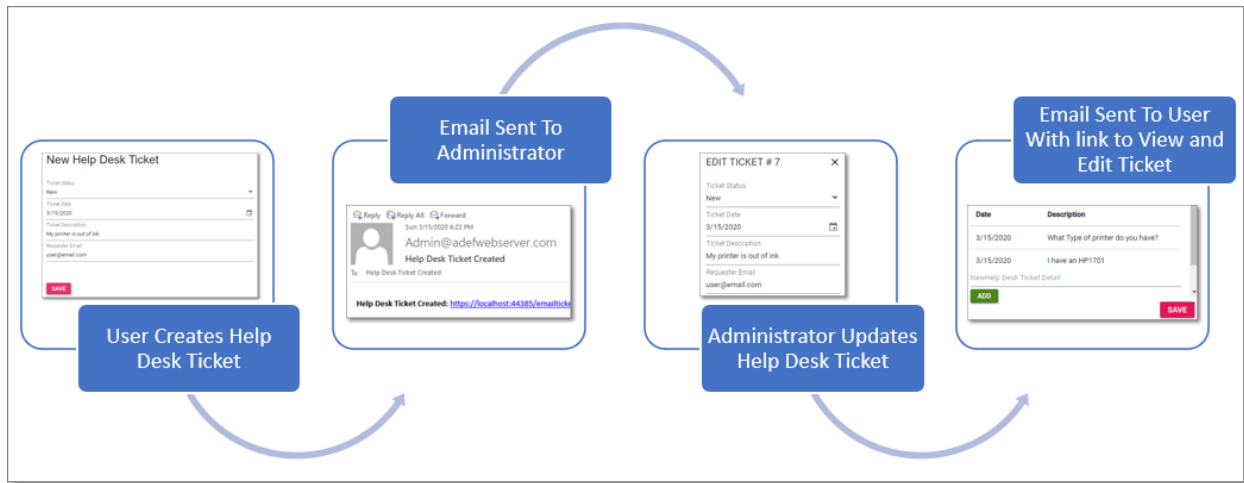
Figure 65: Email Process

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 at https://sendgrid.com/ 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 49: appsettings.json

```
"SENDGRID_APIKEY": "enter your key from app.sendgrid.com",
"SenderEmail": "enter your email address",
```
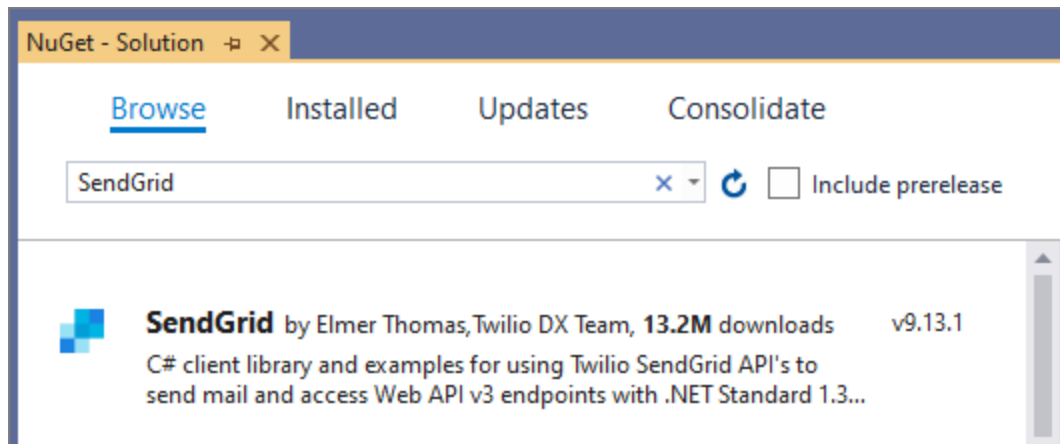
Figure 66: SendGrid NuGet Package

Install the SendGrid NuGet package.

Email sender class
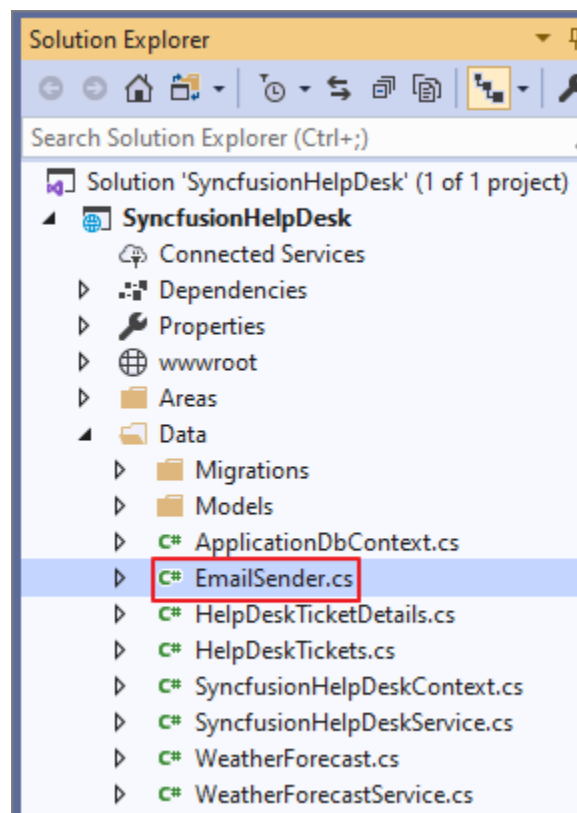
Figure 67: EmailSender.cs

We will create a class that will read the settings from the appsettings.json file and send emails.

Create a new class in the Data folder called EmailSender.cs using the following code.

Code Listing 50: EmailSender.cs

```csharp
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using SendGrid;
using SendGrid.Helpers.Mail;

namespace SyncfusionHelpDesk
{
    public class EmailSender
    {
        private readonly IConfiguration configuration;
        private readonly IHttpContextAccessor httpContextAccessor;

        public EmailSender(
            IConfiguration Configuration,
            IHttpContextAccessor HttpContextAccessor)
        {
            configuration = Configuration;
            httpContextAccessor = HttpContextAccessor;
        }

        public async Task SendEmail(
            string EmailType,
            string EmailAddress,
            string TicketGuid)
        {
            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 =
                    $"{EmailType}: {GetHelpDeskTicketUrl(TicketGuid)}";
                string strHtmlContent =
                    $"<b>{EmailType}:</b> ";
                strHtmlContent = strHtmlContent +
                    $"<a href='{ GetHelpDeskTicketUrl(TicketGuid) }'>";
                strHtmlContent = strHtmlContent +
                    $"{GetHelpDeskTicketUrl(TicketGuid)}</a>";

                if (EmailType == "Help Desk Ticket Created")
                {
```

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

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

                if (EmailType == "Help Desk Ticket Updated")
                {
                    msg = new SendGridMessage()
                    {
                        From = FromEmail,
                        Subject = EmailType,
                        PlainTextContent = strPlainTextContent,
                        HtmlContent = strHtmlContent
                    };

                    // Updated emails go to administrator or ticket creator.
                    // Send to EmailAddress passed to method.
                    msg.AddTo(new EmailAddress(EmailAddress, EmailType));
                }

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

        // Utility

        #region public string GetHelpDeskTicketUrl(string TicketGuid)
        public 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
    }
}
```

To allow this class to be injected in our code as a service, add the following line to the ConfigureServices section of the Startup.cs file.

Code Listing 51: EmailSender in Startup.cs

```
            services.AddScoped<EmailSender>();
```

Send emails—new help desk ticket

To send an email to the administrator when a new help desk ticket is created, open the Index.razor file and add the following line to inject the email service.

Code Listing 52: EmailSender Service

```
@inject EmailSender _EmailSender
```

Next, add the following code to the end of the **HandleValidSubmit** method.

Code Listing 53: HandleValidSubmit method

```
    // Send email.
    await _EmailSender.SendEmail(
        "Help Desk Ticket Created",
        "", // No need to pass an email because it goes to administrator.
        NewHelpDeskTickets.TicketGuid
        );
```

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 file and add the following line to inject the email service.

Code Listing 54: EmailSender Service

```
@inject EmailSender _EmailSender
```

Next, add the following code to the end of the **SaveTicket** method.

Code Listing 55: Send Email

```
        // Send email to requester.
        await _EmailSender.SendEmail(
            "Help Desk Ticket Updated",
            SelectedTicket.TicketRequesterEmail,
            SelectedTicket.TicketGuid
            );
```

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.

Create a new control called EmailTicketEdit.razor with the following code.

Code Listing 56: 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 57: EmailTicketEdit Code

```
@using Microsoft.Extensions.Configuration
@using System.Security.Claims;
@using SyncfusionHelpDesk.Data;
@inject EmailSender _EmailSender
@inject IConfiguration _configuration
@inherits OwningComponentBase<SyncfusionHelpDeskService>
<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="true"
          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 {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

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

    ClaimsPrincipal CurrentUser = new ClaimsPrincipal();
    private HelpDeskTickets SelectedTicket = new HelpDeskTickets();
    private bool EditDialogVisibility = true;

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

    protected override async Task
        OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            // Get the help desk ticket associated with
            // the GUID that was passed to the control.
            SelectedTicket =
                await @Service.GetHelpDeskTicketAsync(TicketGuid);

            StateHasChanged();
        }
    }

    public async Task SaveTicket()
    {
        // Save the help desk ticket.
        var result = await @Service.UpdateTicketAsync(SelectedTicket);

        // Close the Dialog.
        EditDialogVisibility = false;

        // Send emails.
        if (CurrentUser.Identity.IsAuthenticated)
        {
            if (CurrentUser.IsInRole("Administrators"))
            {
                // User an administrator.
                // Send email to requester.
                await _EmailSender.SendEmail(
                "Help Desk Ticket Updated",
                SelectedTicket.TicketRequesterEmail,
                SelectedTicket.TicketGuid
                );

                return;
            }
```

```
        }

        // User is not an administrator.
        // Send email to administrator.
        string AdministratorEmail = _configuration["SenderEmail"];

        await _EmailSender.SendEmail(
        "Help Desk Ticket Updated",
        AdministratorEmail,
        SelectedTicket.TicketGuid
        );
    }
}
```

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 68: Email Link

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

Clicking that link will take the administrator directly to the help desk ticket.

Thanks for reading

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