Building Web User Interfaces Using Blazor

This section is about using Microsoft Blazor to build user interfaces for the web.

I will describe the different flavors of Blazor and their pros and cons. You will learn how to build Blazor components that can execute their code on the web server or in the web browser. When hosted with Blazor Server, it uses SignalR to communicate updates needed to the user interface in the browser. When hosted with Blazor WebAssembly, the components execute their code in the client and must make HTTP calls to interact with the server.

In this section, we will cover the following topics:

- Understanding Blazor
- Building components using Blazor Server
- Building components using Blazor WebAssembly

Understanding Blazor

In *Section 6*, *Introducing Practical Applications of C# and .NET*, I introduced you to Blazor (and SignalR that is used by Blazor Server). Blazor is supported on all modern browsers.

More Information: You can read the official list of supported platforms at the following link: https://docs.microsoft.com/en-us/aspnet/core/blazor/supported-platforms

Understanding Blazor hosting models

As a reminder, Blazor is a single app model with two main hosting models:

- **Blazor Server** runs on the server side, so the C# code that you write has full access to all resources that your business logic might need without needing to authenticate. It then uses SignalR to communicate user interface updates to the client side. The server must keep a live SignalR connection to each client and track the current state of every client, so Blazor Server does not scale well if you need to support lots of clients. It first shipped as part of .NET Core 3.0 in September 2019 and is included with .NET 5.0 and later.
- **Blazor WebAssembly** runs on the client side, so the C# code that you write only has access to resources in the browser and it must make HTTP calls (that might require authentication) before it can access resources on the server. It first shipped as an extension to .NET Core 3.1 in May 2020 and was versioned 3.2 because it is a Current release and therefore not covered by .NET Core 3.1's Long Term Support. The .NET Core 3.2 version used the Mono runtime and Mono libraries; the .NET 5 version uses the Mono runtime and the .NET 5 libraries. "*Blazor WebAssembly runs on a .NET IL interpreter without any JIT so it's not going to win any speed competitions. We have made some significant speed improvements though in .NET 5, and we expect to improve things further for .NET 6.*"—Daniel Roth

Although Blazor Server is supported on Internet Explorer 11, Blazor WebAssembly is not.

Blazor WebAssembly has optional support for **Progressive Web Apps** (**PWAs**), meaning a website visitor can use a browser menu to add the app to their desktop and run the app offline.

More Information: You can read more about hosting models in the official documentation: https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models

Understanding Blazor components

It is important to understand that Blazor is used to create **user interface components**. Components define how to render the user interface, react to user events, and can be composed and nested, and compiled into a NuGet Razor class library for packaging and distribution.

In the future, Blazor might not be limited to only creating user interface components using web technologies. Microsoft has an experimental technology known as **Blazor Mobile Bindings** that allows developers to use Blazor to build mobile user interface components. Instead of using HTML and CSS to build a web user interface, it uses XAML and Xamarin.Forms to build a cross-platform mobile user interface.

More Information: You can read more about Blazor Mobile Bindings at the following link: https://devblogs.microsoft.com/aspnet/mobile-blazor-bindings-experiment/

Microsoft is also experimenting with a hybrid model that enables building apps with a mixture of web and mobile.

More Information: You can read more about Blazor Hybrid apps at the following link: https://devblogs.microsoft.com/aspnet/hybrid-blazor-apps-in-mobile-blazor-bindings-july-update/

What is the deal with Blazor and Razor?

You might wonder why Blazor components use `.razor` as their file extension. Razor is a template markup syntax that allows the mixing of HTML and C#. Older technologies that support Razor use the `.cshtml` file extension to indicate the mix of C# and HTML.

Razor is used for:

- ASP.NET Core MVC **views** and **partial views** that use the `.cshtml` file extension. The business logic is separated into a controller class that treats the view as a template to push the view model to, that then outputs it to a web page.
- **Razor Pages** that use the `.cshtml` file extension. The business logic can be embedded or separated into a file that uses the `.cshtml.cs` file extension. The output is a web page.
- **Blazor components** that use the `.razor` file extension. The output is not a web page although layouts can be used to wrap a component so it outputs as a web page and the `@page` directive can be used to assign a route that defines the URL path to retrieve the component as a page.

Comparing Blazor project templates

One way to understand the choice between the Blazor Server and Blazor WebAssembly hosting models is to review the differences in their default project templates.

Reviewing the Blazor Server project template

Let us look at the default template for a Blazor Server project. Mostly you will see that it is the same as an ASP.NET Core Razor Pages template, with a few key additions:

1. In the folder named `PracticalApps`, create a folder named `NorthwindBlazorServer`.
2. In **Visual Studio Code**, open the `PracticalApps` workspace and add the `NorthwindBlazorServer` folder.
3. Navigate to **Terminal** | **New Terminal** and select `NorthwindBlazorServer`.
4. In **TERMINAL**, use the `blazorserver` template to create a new Blazor Server project, as shown in the following command:

```
dotnet new blazorserver
```

5. Select `NorthwindBlazorServer` as the active OmniSharp project.
6. In the `NorthwindBlazorServer` folder, open `NorthwindBlazorServer.csproj`, and note it is identical to an ASP.NET Core project that uses the Web SDK and targets .NET 5.0.
7. Open `Program.cs`, and note it is identical to an ASP.NET Core project.
8. Open `Startup.cs`, and note the `ConfigureServices` method, with its call to the `AddServerSideBlazor` method, as shown highlighted in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddRazorPages();
  services.AddServerSideBlazor();
  services.AddSingleton<WeatherForecastService>();
}
```

9. Note the `Configure` method, which is similar to an ASP.NET Core Razor Pages project except for the calls to the `MapBlazorHub` and `MapFallbackToPage` methods when configuring endpoints that configure an ASP.NET Core app to accept incoming SignalR connections for Blazor components, and other requests fall back to a Razor Page named `_Host.cshtml`, as shown in the following code:

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

10. In the `Pages` folder, open `_Host.cshtml`, as shown in the following markup:

```
@page "/"
@namespace NorthwindBlazorServer.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{
  Layout = null;
}
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport"
        content="width=device-width, initial-scale=1.0" />
  <title>NorthwindBlazorServer</title>
  <base href="~/" />
  <link rel="stylesheet"
        href="css/bootstrap/bootstrap.min.css" />
  <link href="css/site.css" rel="stylesheet" />
  <link href="_content/NorthwindBlazorServer/_framework/scoped.styles.css" rel="stylesheet" />
</head>
<body>
  <component type="typeof(App)"
             render-mode="ServerPrerendered" />
  <div >
    <environment include="Staging,Production">
      An error has occurred. This application may no longer respond until reloaded.
    </environment>
    <environment include="Development">
      An unhandled exception has occurred. See browser dev tools for details.
    </environment>
    <a href="" class="reload">Reload</a>
    <a class="dismiss">X</a>
  </div>
  <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

While reviewing the preceding markup, note the following:

- In the `<body>`, the Blazor component of type `App` that is prerendered on the server.

- The `<div >` for showing Blazor errors that will appear as a yellow bar at the bottom of the web page when an error occurs.
- The script block for `blazor.server.js` manages the SignalR connection back to the server.

11. In the `NorthwindBlazorServer` folder, open `App.razor`, and note it defines a `Router` for all components found in the current assembly, as shown in the following code:

```
<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData"
               DefaultLayout="@typeof(MainLayout)" />
  </Found>
  <NotFound>
    <LayoutView Layout="@typeof(MainLayout)">
      <p>Sorry, there's nothing at this address.</p>
    </LayoutView>
  </NotFound>
</Router>
```

While reviewing the preceding markup, note the following:

- If a matching route is found, then a `RouteView` is executed that sets the default layout for the component to `MainLayout` and passes any route data parameters to the component.
- If a matching route is not found, then a `LayoutView` is executed that outputs the internal markup (in this case a simple paragraph element with a message telling the visitor there is nothing at this address) inside the `MainLayout`.

12. In the `Shared` folder, open `MainLayout.razor`, and note it defines a `<div>` for a sidebar containing a navigation menu and a `<div>` for the main content, as shown in the following code:

```
@inherits LayoutComponentBase
<div class="page">
  <div class="sidebar">
    <NavMenu />
  </div>
  <div class="main">
    <div class="top-row px-4">
      <a href="https://docs.microsoft.com/aspnet/"
         target="_blank">About</a>
    </div>
    <div class="content px-4">
      @Body
    </div>
  </div>
</div>
```

13. In the `Shared` folder, open `MainLayout.razor.css`, and note it contains isolated CSS styles for the component.
14. In the `Shared` folder, open `NavMenu.razor`, and note it has three menu items for `Home`, `Counter`, and `Fetch data`. We will return to this later when we add our own component.
15. In the `Pages` folder, open `FetchData.razor`, and note it defines a component that fetches weather forecasts from an injected dependency weather service and then renders them in a table, as shown in the following code:

```
@page "/fetchdata"
@using NorthwindBlazorServer.Data
@inject WeatherForecastService ForecastService
<h1>Weather forecast</h1>
<p>This component demonstrates fetching data from a service.</p>
@if (forecasts == null)
{
  <p><em>Loading...</em></p>
}
else
{
  <table class="table">
    <thead>
      <tr>
        <th>Date</th>
        <th>Temp. (C)</th>
        <th>Temp. (F)</th>
        <th>Summary</th>
      </tr>
    </thead>
    <tbody>
      @foreach (var forecast in forecasts)
      {
        <tr>
          <td>@forecast.Date.ToShortDateString()</td>
          <td>@forecast.TemperatureC</td>
          <td>@forecast.TemperatureF</td>
          <td>@forecast.Summary</td>
        </tr>
      }
    </tbody>
  </table>
}
@code {
  private WeatherForecast[] forecasts;
  protected override async Task OnInitializedAsync()
  {
```

```
    forecasts = await ForecastService
      .GetForecastAsync(DateTime.Now);
  }
}
```

16. In the `Data` folder, open `WeatherForecastService.cs`, and note it is *not* a Web API controller class, it is just an ordinary class that returns random weather data, as shown in the following code:

```
using System;
using System.Linq;
using System.Threading.Tasks;
namespace NorthwindBlazorServer.Data
{
  public class WeatherForecastService
  {
    private static readonly string[] Summaries = new[]
    {
      "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    };
    public Task<WeatherForecast[]> GetForecastAsync(
      DateTime startDate)
    {
      var rng = new Random();
      return Task.FromResult(
        Enumerable.Range(1, 5)
        .Select(index => new WeatherForecast
      {
        Date = startDate.AddDays(index),
        TemperatureC = rng.Next(-20, 55),
        Summary = Summaries[rng.Next(Summaries.Length)]
      }).ToArray());
    }
  }
}
```

Understanding CSS isolation

Blazor components often need to provide their own CSS to apply styling. To ensure this does not conflict with site-level CSS, Blazor supports CSS isolation. If you have a component named `Index.razor`, simply create a CSS file named `Index.razor.css`.

More Information: You can read more about the reason for needing CSS isolation for Blazor components at the following link: https://github.com/dotnet/aspnetcore/issues/10170

Running the Blazor Server project template

Now that we have reviewed the project template and the important parts that are specific to Blazor Server, we can start the website and review its behavior:

1. In **TERMINAL**, enter a command to run the website, as shown in the following command line:

```
dotnet run
```

2. Start your browser and navigate to `https://localhost:5001/`, and click **Fetch data**, as shown in the following screenshot:

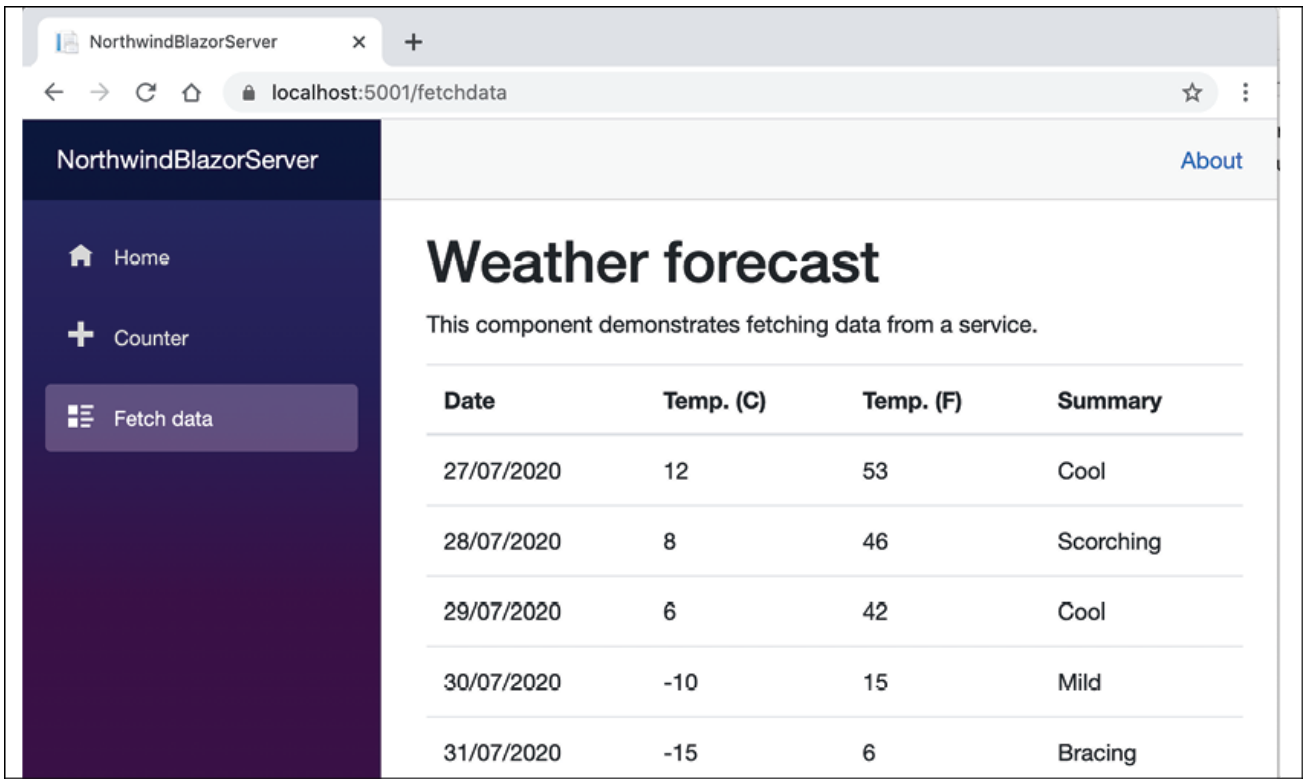


Figure 20.1: Fetching weather data

3. Change the route to `/apples`, and note the missing message, as shown in the following screenshot:

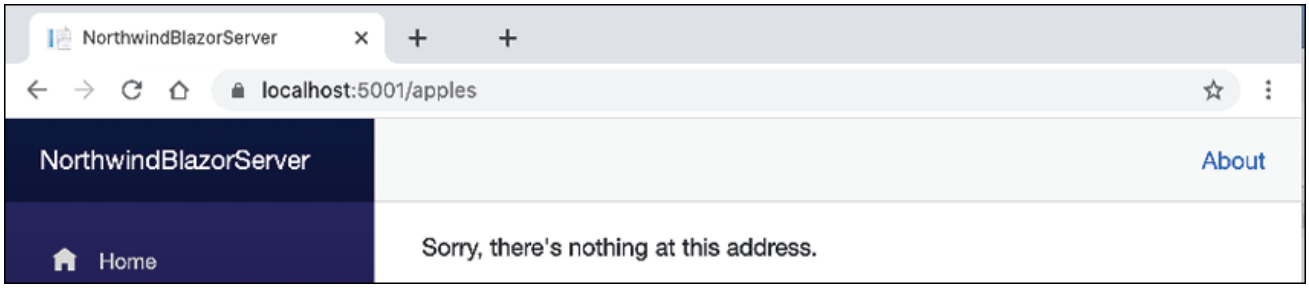


Figure 20.2: The missing component message

4. Close Chrome.
5. In **Visual Studio Code**, press `Ctrl` + `C` in **TERMINAL** to stop the web server.

Reviewing the Blazor WebAssembly project template

Now we will create a Blazor WebAssembly project. I will not show code that is the same as in a Blazor Server project:

1. In the folder named `PracticalApps`, create a folder named `NorthwindBlazorWasm`.
2. In **Visual Studio Code**, open the `PracticalApps` workspace and add the `NorthwindBlazorWasm` folder.
3. Navigate to **Terminal** | **New Terminal** and select `NorthwindBlazorWasm`.
4. In **TERMINAL**, use the `blazorwasm` template with the `--pwa` and `--hosted` flags to create a new Blazor WebAssembly project hosted in ASP.NET Core that supports the PWA feature of your operating system, as shown in the following command:

```
dotnet new blazorwasm --pwa --hosted
```

   While reviewing the generated project, note the following:

   - A solution and three project folders are generated: **Client**, **Server**, and **Shared**.
   - **Shared** is a class library that contains models for the weather service.
   - **Server** is an ASP.NET Core website for hosting the weather service that has the same implementation for returning random weather forecasts as before but is implemented as a proper Web API controller class. The project file has project references to **Shared** and **Client**, and a package reference to support WebAssembly on the server side.
   - **Client** is the Blazor WebAssembly project.
5. In the `Client` folder, open `NorthwindBlazorWasm.Client.csproj`, and note it uses the Blazor WebAssembly SDK and has three package references, as well as the service worker required for PWA support, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <ServiceWorkerAssetsManifest>service-worker-assets.js</ServiceWorkerAssetsManifest>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.AspNetCore.Components.WebAssembly"
      Version="5.0.0" />
    <PackageReference
      Include="Microsoft.AspNetCore.Components
        .WebAssembly.DevServer"
      Version="5.0.0" PrivateAssets="all" />
    <PackageReference
      Include="System.Net.Http.Json"
      Version="5.0.0" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\Shared\NorthwindBlazorWasm.Shared.csproj" />
  </ItemGroup>
  <ItemGroup>
    <ServiceWorker Include=
      "wwwroot\service-worker.js" PublishedContent=
      "wwwroot\service-worker.published.js" />
  </ItemGroup>
</Project>
```

6. In the `Client` folder, open `Program.cs`, and note the host builder is for `WebAssembly` instead of server-side ASP.NET Core, and it registers a dependency service for making HTTP requests, which is an extremely common requirement for Blazor WebAssembly apps, as shown in the following code:

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.Services.AddScoped(sp => new HttpClient
  { BaseAddress = new Uri(
    builder.HostEnvironment.BaseAddress) });
await builder.Build().RunAsync();
```

7. In the `wwwroot` folder, open `index.html`, and note the `manifest.json` and `service-worker.js` files to support offline work, and the `blazor.webassembly.js` script that downloads all the NuGet packages for Blazor WebAssembly, as shown highlighted in the following markup:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=no" />
  <title>NorthwindBlazorWasm</title>
  <base href="/" />
  <link href="css/bootstrap/bootstrap.min.css"
        rel="stylesheet" />
  <link href="css/app.css" rel="stylesheet" />
  <link href="_framework/scoped.styles.css"
        rel="stylesheet" />
  <link href="manifest.json" rel="manifest" />
  <link rel="apple-touch-icon" sizes="512x512"
        href="icon-512.png" />
</head>
<body>
  <div >Loading...</div>
  <div >
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">X</a>
  </div>
  <script src="_framework/blazor.webassembly.js"></script>
  <script>navigator.serviceWorker
    .register('service-worker.js');</script>
</body>
</html>
```

8. In the `Client` folder, note the following files are identical to Blazor Server: `App.razor`, `Shared\MainLayout.razor`, `Shared\NavMenu.razor`, `SurveyPrompt.razor`, `Pages\Counter.razor`, and `Pages\Index.razor`.
9. In the `Pages` folder, open `FetchData.razor`, and note the markup is similar to Blazor Server except for the injected dependency service for making HTTP requests, as shown highlighted in the following partial markup:

```
@page "/fetchdata"
@using NorthwindBlazorWasm.Shared
@inject HttpClient Http
<h1>Weather forecast</h1>
...
@code {
  private WeatherForecast[] forecasts;
  protected override async Task OnInitializedAsync()
  {
    forecasts = await
      Http.GetFromJsonAsync<WeatherForecast[]>(
        "WeatherForecast");
```

```
  }
}
```

10. Start the `Server` project, as shown in the following commands:

```
cd Server
dotnet run
```

11. Note the app has the same functionality as before, but the code is executing inside the browser instead of on the server.
12. Close Chrome.
13. In **Visual Studio Code**, press `Ctrl` + `C` in **TERMINAL** to stop the web server.

Building components using Blazor Server

In this section we will build a component to list, create, and edit customers in the `Northwind` database.

Defining and testing a simple component

We will add the new component to the existing Blazor Server project:

1. In the `NorthwindBlazorServer` project, add a new file to the `Pages` folder named `Customers.razor`.

   Good Practice: Component filenames must start with an uppercase letter or you will see compile errors!

2. Add statements to register `/customers` as its route, output a heading for the customers component, and define a code block, as shown in the following markup:

```
@page "/customers"
<h1>Customers</h1>
@code {

}
```

3. In the `Shared` folder, open `NavMenu.razor` and add a list item element for our new component labeled `Customers` that uses an icon of people, as shown in the following markup:

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

   More Information: You can see the available icons at the following link: https://iconify.design/icon-sets/oi/

4. Start the website project and navigate to it, and click **Customers**, as shown in the following screenshot:

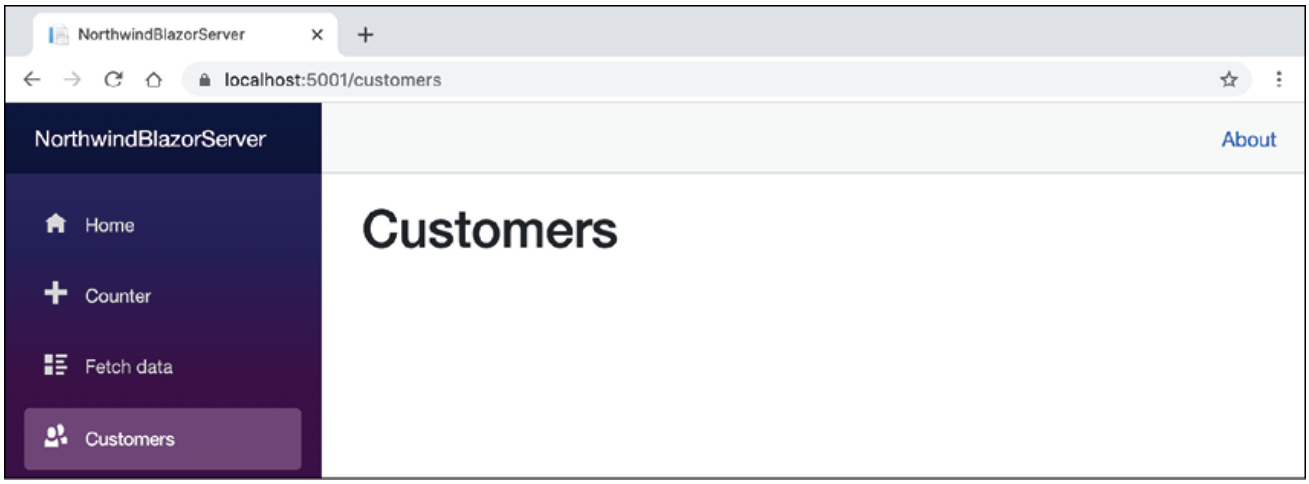


   Figure 20.3: The Customers component shown as a page

5. Close Chrome.
6. In **Visual Studio Code**, press `Ctrl` + `C` in **TERMINAL** to stop the web server.

Getting entities into a component

Now that you have seen the minimum implementation of a component, we can add some useful functionality to it. In this case, we will use the `Northwind` database context to fetch customers from the database:

1. Open `NorthwindBlazorServer.csproj`, and add statements to reference the `Northwind` database context project, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindContextLib\NorthwindContextLib.csproj" />
  </ItemGroup>
</Project>
```

2. In **TERMINAL**, restore packages and compile the project by entering the following command: `dotnet build`
3. Open `Startup.cs` and add the `System.IO`, `Microsoft.EntityFrameworkCore`, and `Packt.Shared` namespaces, as shown in the following code:

```
using Microsoft.EntityFrameworkCore;
using Packt.Shared;
using System.IO;
```

4. Add a statement to the `ConfigureServices` method to register the `Northwind` database context class to use SQLite as its database provider and specify its database connection string, as shown in the following code:

```
string databasePath = Path.Combine("..", "Northwind.db");
services.AddDbContext<Northwind>(options =>
  options.UseSqlite($"Data Source={databasePath}"));
```

5. Open `_Imports.razor` and import the `NorthwindBlazorServer.Data`, `Microsoft.EntityFrameworkCore`, and `Packt.Shared` namespaces, so that Blazor components that we build do not need to import the namespaces individually, as shown highlighted in the following markup:

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
@using NorthwindBlazorServer
@using NorthwindBlazorServer.Shared
@using NorthwindBlazorServer.Data
@using Microsoft.EntityFrameworkCore
@using Packt.Shared
```

6. In the `Pages` folder, open `Customers.razor`, inject the `Northwind` database context, and use it to output a table of all customers, as shown in the following code:

```
@page "/customers"
@inject Northwind db
<h1>Customers</h1>
@if (customers == null)
{
  <p><em>Loading...</em></p>
}
else
{
  <table class="table">
    <thead>
      <tr>
        <th>ID</th>
        <th>Company Name</th>
        <th>Address</th>
        <th>Phone</th>
        <th></th>
      </tr>
    </thead>
    <tbody>
      @foreach (var customer in customers)
      {
      <tr>
        <td>@customer.CustomerID</td>
        <td>@customer.CompanyName</td>
        <td>@customer.Address<br/>
        @customer.City<br/>
        @customer.PostalCode<br/>
        @customer.Country</td>
        <td>@customer.Phone</td>
        <td>
          <a class="btn btn-info"
             href="editcustomer/@customer.CustomerID">
               <i class="oi oi-pencil"></i></a>
          <a class="btn btn-danger"
             href="deletecustomer/@customer.CustomerID">
               <i class="oi oi-trash"></i></a>
        </td>
      </tr>
      }
    </tbody>
  </table>
}
@code {
  private IEnumerable<Customer> customers;
  protected override async Task OnInitializedAsync()
  {
    customers = await db.Customers.ToListAsync();
  }
}
```

7. In **TERMINAL**, enter the command `dotnet run` to start the website.
8. In Chrome, enter `https://localhost:5001/`, click **Customers**, and note the table of customers loads from the database and renders in the web page, as shown in the following screenshot:

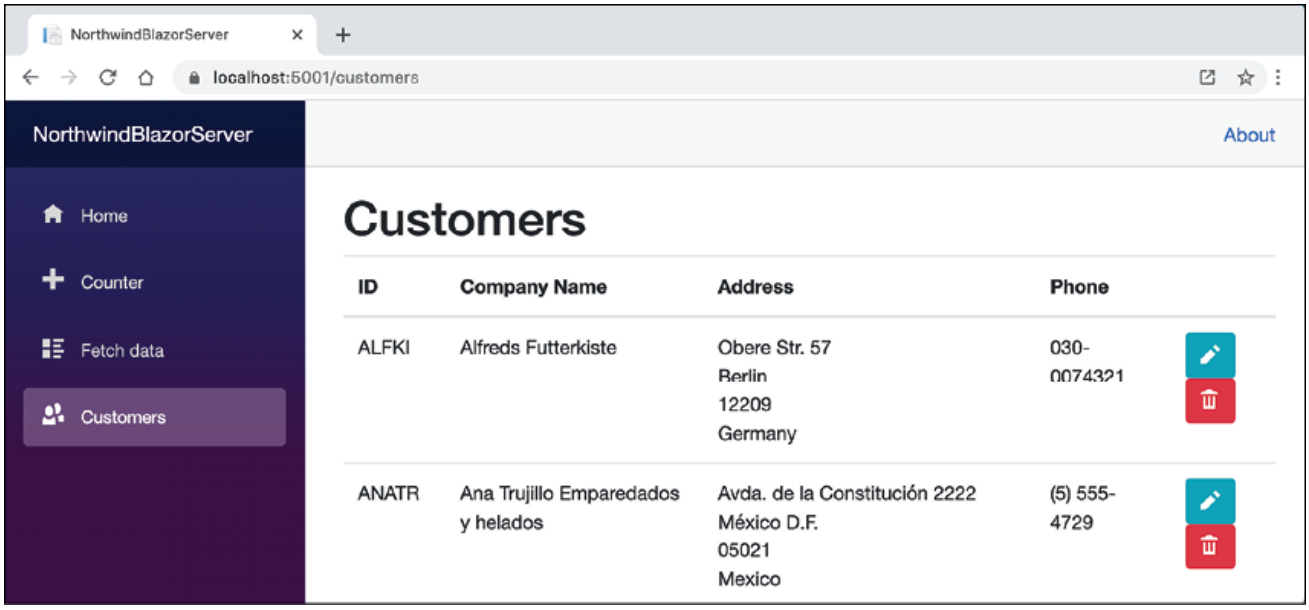


Figure 20.4: The list of customers

9. Close the browser.
10. In **Visual Studio Code**, press `Ctrl` + `C` in **TERMINAL** to stop the web server.

There are many built-in Blazor components, including ones to set elements like `<title>` in the `<head>` section of a web page, and plenty of third parties who will sell you components for common purposes.

More Information: You can read more about setting `<head>` elements at the following link: https://docs.microsoft.com/en-us/aspnet/core/blazor/fundamentals/additional-scenarios-influence-html-head-tag-elements

Abstracting a service for a Blazor component

Currently, the Blazor component directly calls the `Northwind` database context to fetch the customers. This works fine in Blazor Server since the component executes on the server. But this component would not work when hosted in Blazor WebAssembly.

We will now create a local dependency service to enable better reuse of the components:

1. In the `Data` folder, add a new file named `INorthwindService.cs` and modify its contents to define a contract for a local service that abstracts CRUD operations, as shown in the following code:

```
using System.Collections.Generic;
using System.Threading.Tasks;
namespace Packt.Shared
{
  public interface INorthwindService
  {
    Task<List<Customer>> GetCustomersAsync();
    Task<Customer> GetCustomerAsync(string id);
    Task<Customer> CreateCustomerAsync(Customer c);
    Task<Customer> UpdateCustomerAsync(Customer c);
    Task DeleteCustomerAsync(string id);
  }
}
```

2. In the `Data` folder, add a new file named `NorthwindService.cs`, and modify its contents to implement the `INorthwindService` interface by using the `Northwind` database context, as shown in the following code:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Packt.Shared;
namespace NorthwindBlazorServer.Data
{
  public class NorthwindService : INorthwindService
  {
    private readonly Northwind db;
    public NorthwindService(Northwind db)
    {
      this.db = db;
    }
    public Task<List<Customer>> GetCustomersAsync()
    {
      return db.Customers.ToListAsync();
    }
    public Task<Customer> GetCustomerAsync(string id)
```

```
    {
      return db.Customers.FirstOrDefaultAsync
        (c => c.CustomerID == id);
    }
    public Task<Customer> CreateCustomerAsync(Customer c)
    {
      db.Customers.Add(c);
      db.SaveChangesAsync();
      return Task.FromResult<Customer>(c);
    }
    public Task<Customer> UpdateCustomerAsync(Customer c)
    {
      db.Entry(c).State = EntityState.Modified;
      db.SaveChangesAsync();
      return Task.FromResult<Customer>(c);
    }
    public Task DeleteCustomerAsync(string id)
    {
      Customer customer = db.Customers.FirstOrDefaultAsync
        (c => c.CustomerID == id).Result;
      db.Customers.Remove(customer);
      return db.SaveChangesAsync();
    }
  }
}
```

3. Open `Startup.cs` and in the `ConfigureServices` method add a statement to register the `NorthwindService` as a transient service that implements the `INorthwindService` interface, as shown in the following code:

```
services.AddTransient
  <INorthwindService, NorthwindService>();
```

4. In the `Pages` folder, open `Customers.razor`, remove the directive to inject the `Northwind` database context, and add a directive to inject the registered Northwind service, as shown in the following code:

```
@inject INorthwindService service
```

5. Modify the `OnInitializedAsync` method to call the service, as shown in the following code:

```
customers = await service.GetCustomersAsync();
```

6. If you would like, run the `NorthwindBlazorServer` website project to test that it retains the same functionality as before.

Using Blazor forms

Microsoft provides ready-made components for building forms. We will use them to provide, create, and edit functionality for customers.

Defining forms using the EditForm component

Microsoft provides the **EditForm** component and several form elements like `InputText` to make it easier to use forms with Blazor.

EditForm can have a model set to bind it to an object with properties and event handlers for custom validation, as well as recognizing standard Microsoft validation attributes on the model class, as shown in the following code:

```
<EditForm Model="@customer" OnSubmit="ExtraValidation">
  <DataAnnotationsValidator />
  <ValidationSummary />
  <InputText  @bind-Value="customer.CompanyName" />
  <button type="submit">Submit</button>
</EditForm>
@code {
  private Customer customer = new Customer();
  private void ExtraValidation()
  {
    // perform validation
  }
}
```

As an alternative to a `ValidationSummary` component, you can use the `ValidationMessage` component to show a message next to an individual form element.

More Information: You can read more about forms and validation at the following link: https://docs.microsoft.com/en-us/aspnet/core/blazor/forms-validation

Navigating Blazor routes

Microsoft provides a dependency service named `NavigationManager` that understands Blazor routing and the `NavLink` component.

The `NavigateTo` method is used to go to the specified URL.

More Information: You can read more about using `NavigationManager` with Blazor routes at the following link: https://docs.microsoft.com/en-us/aspnet/core/blazor/fundamentals/routing#uri-and-navigation-state-helpers

Building and using a customer form component

Now we can create a custom component to create or edit a customer:

1. In the `Pages` folder, create a new file named `CustomerDetail.razor` and modify its contents to define a form to edit the properties of a customer, as shown in the following code:

```
<EditForm Model="@Customer" OnValidSubmit="@OnValidSubmit">
  <DataAnnotationsValidator />
  <div class="form-group">
    <div>
      <label>Customer ID</label>
      <div>
        <InputText @bind-Value="@Customer.CustomerID" />
        <ValidationMessage
          For="@(() => Customer.CustomerID)" />
      </div>
    </div>
  </div>
  <div class="form-group ">
    <div>
      <label>Company Name</label>
      <div>
        <InputText @bind-Value="@Customer.CompanyName" />
        <ValidationMessage
          For="@(() => Customer.CompanyName)" />
      </div>
    </div>
  </div>
  <div class="form-group ">
    <div>
      <label>Address</label>
      <div>
        <InputText @bind-Value="@Customer.Address" />
        <ValidationMessage
          For="@(() => Customer.Address)" />
      </div>
    </div>
  </div>
  <div class="form-group ">
    <div>
      <label>Country</label>
      <div>
        <InputText @bind-Value="@Customer.Country" />
        <ValidationMessage
          For="@(() => Customer.Country)" />
      </div>
    </div>
  </div>
  <button type="submit" class="btn btn-@ButtonStyle">
    @ButtonText
  </button>
</EditForm>
@code {
  [Parameter]
  public Customer Customer { get; set; }
  [Parameter]
  public string ButtonText { get; set; } = "Save Changes";
  [Parameter]
  public string ButtonStyle { get; set; } = "info";
  [Parameter]
  public EventCallback OnValidSubmit { get; set; }
}
```

2. In the `Pages` folder, create a new file named `CreateCustomer.razor` and modify its contents to use the customer detail component to create a new customer, as shown in the following code:

```
@page "/createcustomer"
@inject INorthwindService service
@inject NavigationManager navigation
<h3>Create Customer</h3>
<CustomerDetail ButtonText="Create Customer"
                Customer="@customer"
                OnValidSubmit="@Create" />
@code {
  private Customer customer = new Customer();
  private async Task Create()
  {
    await service.CreateCustomerAsync(customer);
    navigation.NavigateTo("customers");
  }
}
```

3. In the `Pages` folder, open the file named `Customers.razor` and after the `<h1>` element, add a `<div>` element with a button to navigate to the create customer component, as shown in the following markup:

```
<div class="form-group">
  <a class="btn btn-info" href="createcustomer">
    <i class="oi oi-plus"></i> Create New</a>
</div>
```

4. In the `Pages` folder, create a new file named `EditCustomer.razor` and modify its contents to use the customer detail component to edit and save changes to an existing customer, as shown in the following code:

```
@page "/editcustomer/{customerid}"
@inject INorthwindService service
@inject NavigationManager navigation
<h3>Edit Customer</h3>
<CustomerDetail ButtonText="Update"
                Customer="@customer"
                OnValidSubmit="@Update" />
@code {
  [Parameter]
  public string CustomerID { get; set; }
  private Customer customer = new Customer();
  protected async override Task OnParametersSetAsync()
  {
    customer = await service.GetCustomerAsync(CustomerID);
  }
  private async Task Update()
  {
    await service.UpdateCustomerAsync(customer);
    navigation.NavigateTo("customers");
  }
}
```

5. In the `Pages` folder, create a new file named `DeleteCustomer.razor` and modify its contents to use the customer detail component to show the customer that is about to be deleted, as shown in the following code:

```
@page "/deletecustomer/{customerid}"
@inject INorthwindService service
@inject NavigationManager navigation
<h3>Delete Customer</h3>
<div class="alert alert-danger">
  Warning! This action cannot be undone!
</div>
<CustomerDetail ButtonText="Delete Customer"
                ButtonStyle="danger"
                Customer="@customer"
                OnValidSubmit="@Delete" />
@code {
  [Parameter]
  public string CustomerID { get; set; }
  private Customer customer = new Customer();
  protected async override Task OnParametersSetAsync()
  {
    customer = await service.GetCustomerAsync(CustomerID);
  }
  private async Task Delete()
  {
    await service.DeleteCustomerAsync(CustomerID);
    navigation.NavigateTo("customers");
  }
}
```

6. Start the website project and navigate to `https://localhost:5001/`.
7. Navigate to **Customers** and click the **+ Create New** button.
8. Enter an invalid **Customer ID** like `ABCDEF`, leave the text box, and note the validation message, as shown in the following screenshot:

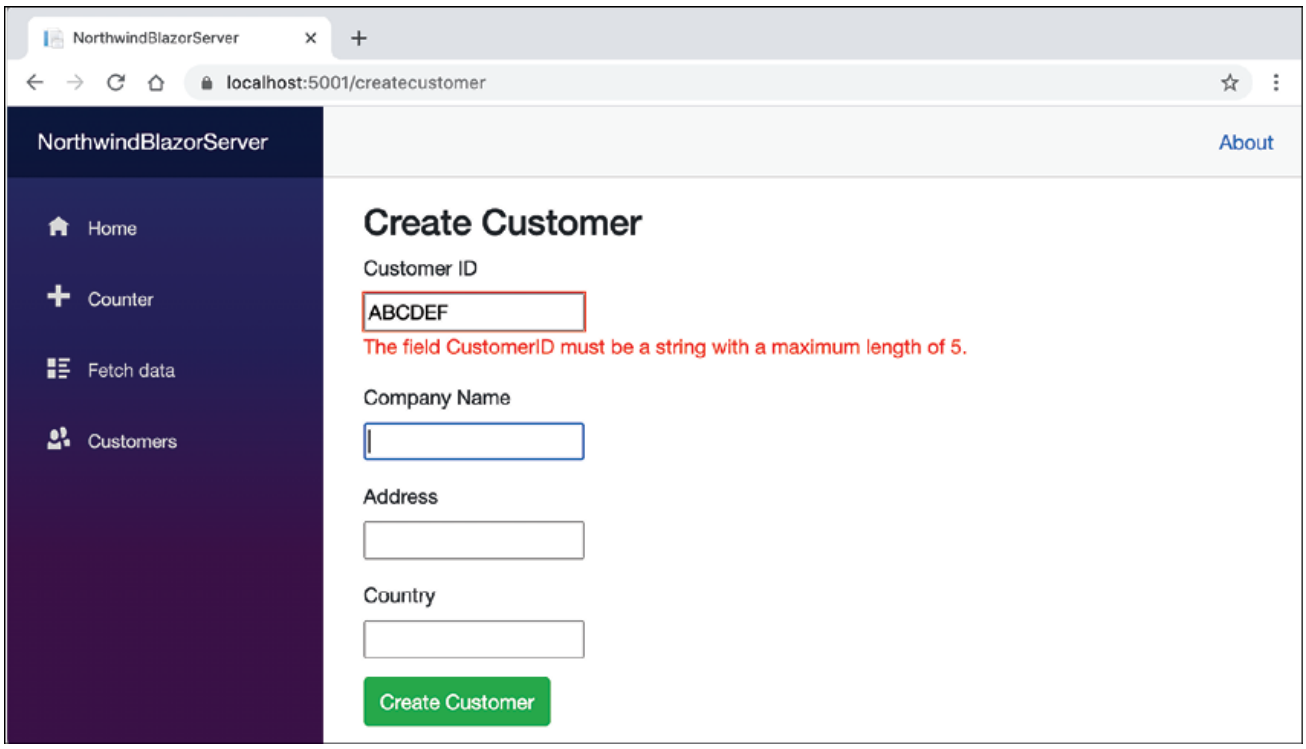


Figure 20.5: Creating a new customer and entering an invalid Customer ID

9. Change the **Customer ID** to ABCDE, enter values for the other textboxes, and click the **Create Customer** button, as shown in the following screenshot:

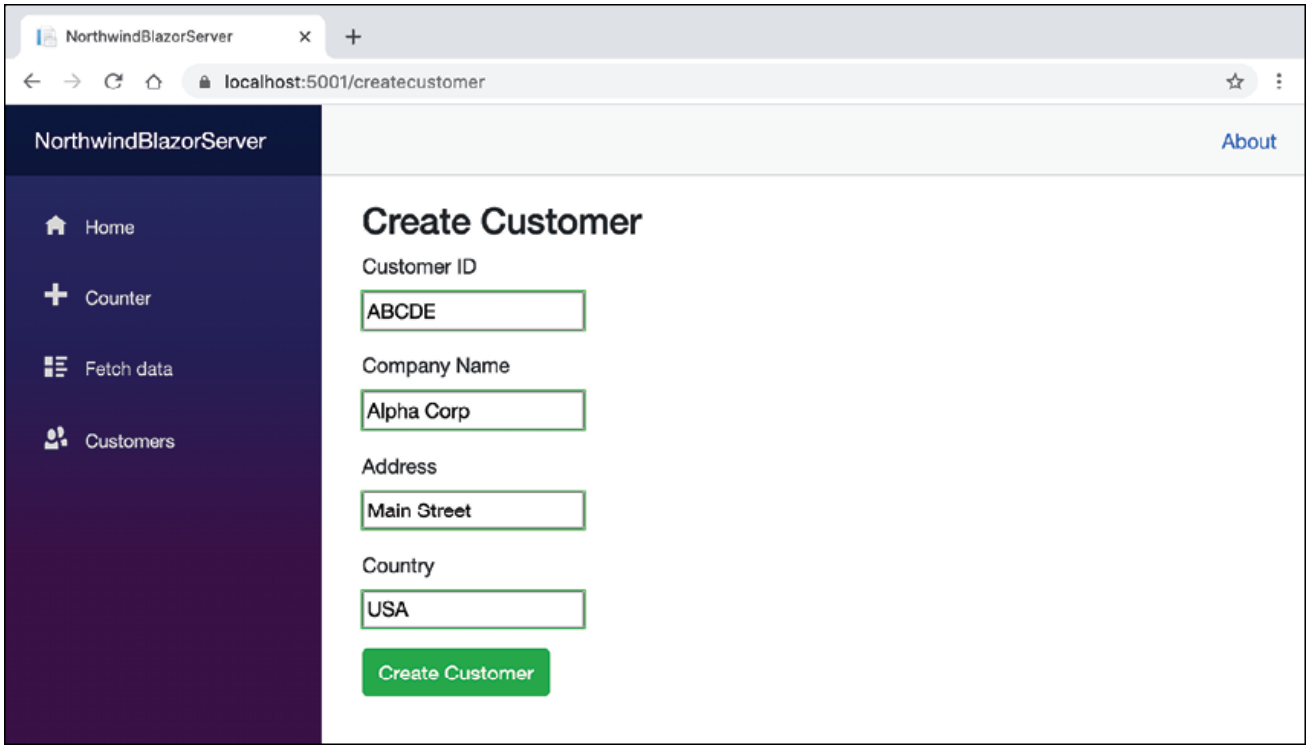


Figure 20.6: New customer information that validates successfully

10. When the list of customers appears, scroll down to the bottom of the page to see the new customer, as shown in the following screenshot:

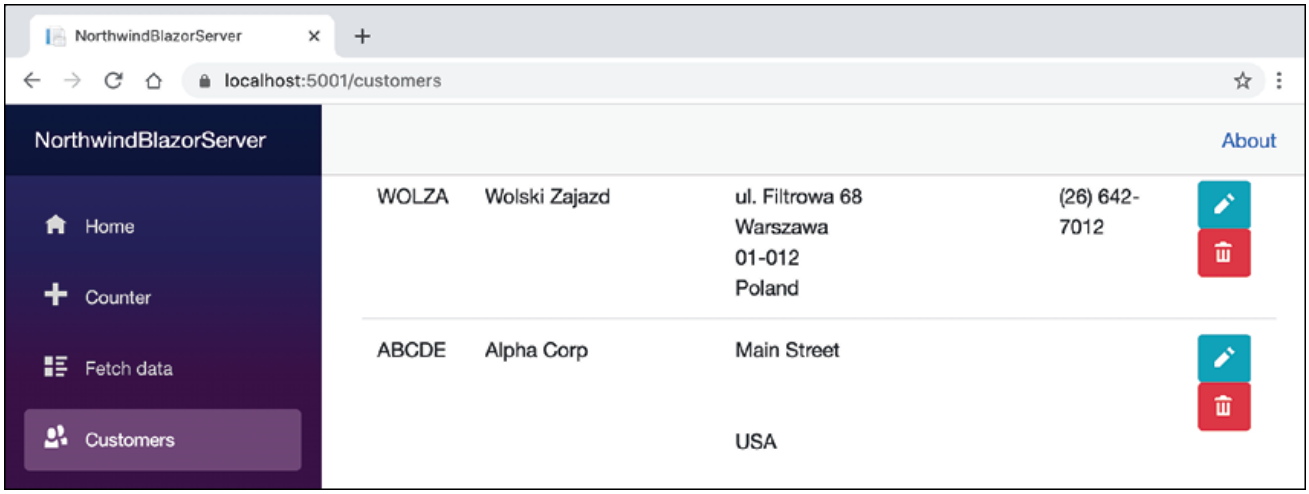


Figure 20.7: Viewing the new customer

11. On the **ABCDE** customer row, click the **Edit** icon button, change the address, click **Update**, and note that the customer record has been updated.
12. On the **ABCDE** customer row, click the **Delete** icon button, note the warning, click the **Delete Customer** button, and note that the customer record has been deleted, as shown in the following screenshot:

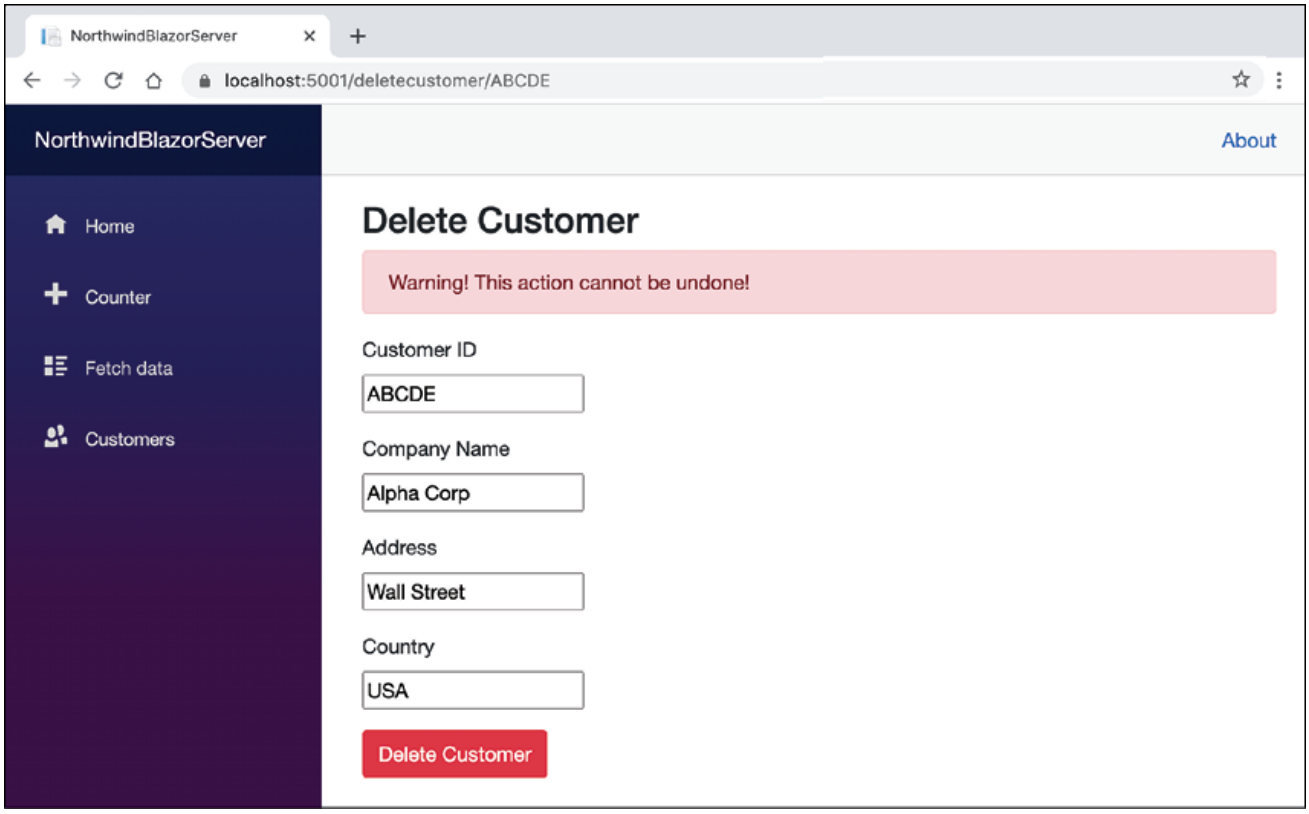


Figure 20.8: Deleting a customer

13. Close Chrome.
14. In **Visual Studio Code**, press `Ctrl` + `C` in **TERMINAL** to stop the web server.

Building components using Blazor WebAssembly

Now we will build the same functionality using Blazor WebAssembly so that you can clearly see the key differences.

Since we abstracted the local dependency service in the `INorthwindService` interface, we will be able to reuse all the components and that interface, as well as the entity model classes, and just rewrite the implementation of the `NorthwindService` class and create a customer controller for its implementation to call for Blazor WebAssembly, as shown in the following diagram:

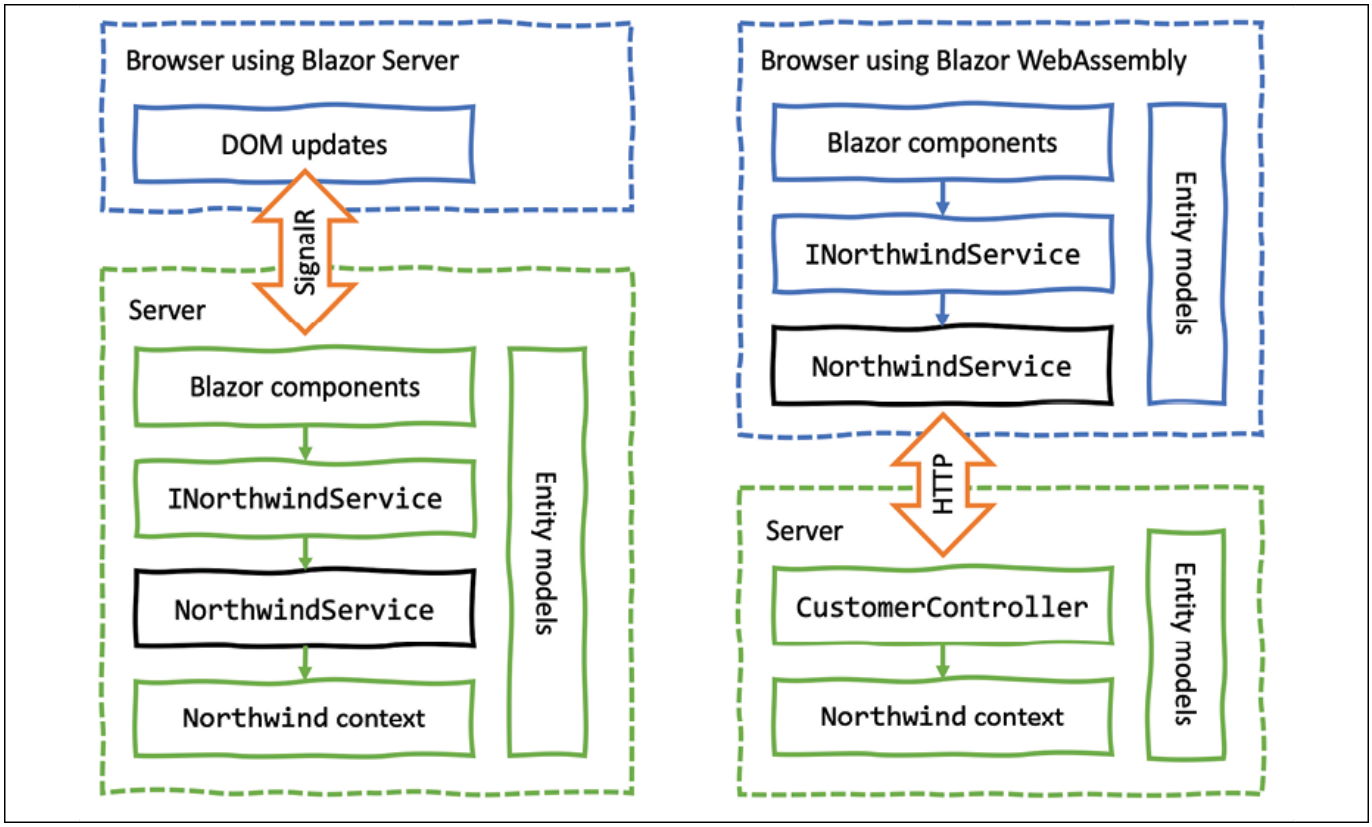


Figure 20.9: Comparing Blazor implementations using Server and WebAssembly

Configuring the server for Blazor WebAssembly

First, we need to build a service that the client app can call using HTTP:

Warning! All relative path references for projects and the database are two levels up, for example, `"..\..\"`.

1. In the **Server** project, open `NorthwindBlazorWasm.Server.csproj`, and add statements to reference the `Northwind` database context project, as shown in the following markup:

```
<ItemGroup>
  <ProjectReference Include=
    "..\..\NorthwindContextLib\NorthwindContextLib.csproj" />
</ItemGroup>
```

2. In **TERMINAL**, in the `Server` folder, restore packages and compile the project, as shown in the following command:

```
dotnet build
```

3. In the **Server** project, open `Startup.cs`, and add statements to import some namespaces, as shown in the following code:

```
using Packt.Shared;
using Microsoft.EntityFrameworkCore;
using System.IO;
```

4. In the `ConfigureServices` method, add statements to register the `Northwind` database context, as shown in the following code:

```
string databasePath = Path.Combine(
  "..", "..", "Northwind.db");
services.AddDbContext<Northwind>(options =>
  options.UseSqlite($"Data Source={databasePath}"));
```

5. In the **Server** project, in the `Controllers` folder, create a file named `CustomersController.cs`, and add statements to define a Web API controller class with similar CRUD methods as before, as shown in the following code:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Packt.Shared;
namespace NorthwindBlazorWasm.Server.Controllers
{
  [ApiController]
  [Route("api/[controller]")]
  public class CustomersController : ControllerBase
  {
    private readonly Northwind db;
    public CustomersController(Northwind db)
    {
      this.db = db;
    }
    [HttpGet]
    public async Task<List<Customer>> GetCustomersAsync()
    {
      return await db.Customers.ToListAsync();
    }
    [HttpGet("{id}")]
    public async Task<Customer> GetCustomerAsync(string id)
    {
      return await db.Customers.FirstOrDefaultAsync
        (c => c.CustomerID == id);
    }
    [HttpPost]
    public async Task<Customer>CreateCustomerAsync
      (Customer customerToAdd)
    {
      Customer existing = await db.Customers
        .FirstOrDefaultAsync
        (c => c.CustomerID == customerToAdd.CustomerID);
      if (existing == null)
      {
        db.Customers.Add(customerToAdd);
        int affected = await db.SaveChangesAsync();
        if (affected == 1)
        {
          return customerToAdd;
        }
      }
      return existing;
    }
    [HttpPut]
    public async Task<Customer> UpdateCustomerAsync
      (Customer c)
    {
      db.Entry(c).State = EntityState.Modified;
      int affected = await db.SaveChangesAsync();
      if (affected == 1)
      {
        return c;
      }
      return null;
    }
    [HttpDelete("{id}")]
    public async Task<int> DeleteCustomerAsync(string id)
    {
      Customer c = await db.Customers.FirstOrDefaultAsync
        (c => c.CustomerID == id);
      if (c != null)
      {
        db.Customers.Remove(c);
        int affected = await db.SaveChangesAsync();
        return affected;
      }
      return 0;
    }
  }
}
```

Configuring the client for Blazor WebAssembly

Second, we can reuse the components from the Blazor Server project. Since the components will be identical, we can copy them and only need to make changes to the local implementation of the abstracted Northwind service:

1. In the **Client** project, open `NorthwindBlazorWasm.Client.csproj`, and add statements to reference the Northwind entities library project, as shown in the following markup:

```
<ItemGroup>
  <ProjectReference Include=
"..\..\NorthwindEntitiesLib\NorthwindEntitiesLib.csproj" />
</ItemGroup>
```

2. In **TERMINAL**, in the `Client` folder, restore packages and compile the project, as shown in the following commands:

```
cd ..
cd Client
dotnet build
```

3. In the **Client** project, open `_Imports.razor` and import the `Packt.Shared` namespace to make the Northwind entity model types available in all Blazor components, as shown in the following code:

```
@using Pact.Shared
```

4. In the **Client** project, in the `Shared` folder, open `NavMenu.razor` and add a `NavLink` element for customers, as shown in the following markup:

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

5. Copy the following five components from the `NorthwindBlazorServer` project `Pages` folder to the `NorthwindBlazorWasm Client` project `Pages` folder:
   - `CreateCustomer.razor`
   - `CustomerDetail.razor`
   - `Customers.razor`
   - `DeleteCustomer.razor`
   - `EditCustomer.razor`
6. In the **Client** project, create a `Data` folder.
7. Copy the `INorthwindService.cs` file from the `NorthwindBlazorServer` project `Data` folder into the `Client` project `Data` folder.
8. In the `Data` folder, add a new file named `NorthwindService.cs`, and modify its contents to implement the `INorthwindService` interface by using an `HttpClient` to call the customers Web API service, as shown in the following code:

```
using System.Collections.Generic;
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;
using Packt.Shared;
namespace NorthwindBlazorWasm.Client.Data
{
  public class NorthwindService : INorthwindService
  {
    private readonly HttpClient http;
    public NorthwindService(HttpClient http)
    {
      this.http = http;
    }
    public Task<List<Customer>> GetCustomersAsync()
    {
      return http.GetFromJsonAsync
        <List<Customer>>("api/customers");
    }
    public Task<Customer> GetCustomerAsync(string id)
    {
      return http.GetFromJsonAsync
        <Customer>($"api/customers/{id}");
    }
    public async Task<Customer> CreateCustomerAsync
      (Customer c)
    {
      HttpResponseMessage response = await
        http.PostAsJsonAsync<Customer>
        ("api/customers", c);
      return await response.Content
        .ReadFromJsonAsync<Customer>();
    }
    public async Task<Customer> UpdateCustomerAsync
      (Customer c)
    {
      HttpResponseMessage response = await
        http.PutAsJsonAsync<Customer>
        ("api/customers", c);
      return await response.Content
        .ReadFromJsonAsync<Customer>();
    }
    public async Task DeleteCustomerAsync(string id)
    {
      HttpResponseMessage response = await
        http.DeleteAsync($"api/customers/{id}");
    }
  }
}
```

9. Open `Program.cs` and import the `Packt.Shared` and `NorthwindBlazorWasm.Client.Data` namespaces.
10. In the `ConfigureServices` method, add a statement to register the Northwind dependency service, as shown in the following code:

```
builder.Services.AddTransient
  <INorthwindService, NorthwindService>();
```

11. In **TERMINAL**, in the `Server` folder, compile the project, as shown in the following commands:

```
cd ..
cd Server
dotnet run
```

12. Start Chrome, show **Developer Tools**, and select the **Network** tab.
13. In the address bar, enter the following: `https://localhost:5001/`.
14. Select the **Console** tab and note that Blazor WebAssembly has loaded .NET 5 assemblies into the browser cache, as shown in the following screenshot:

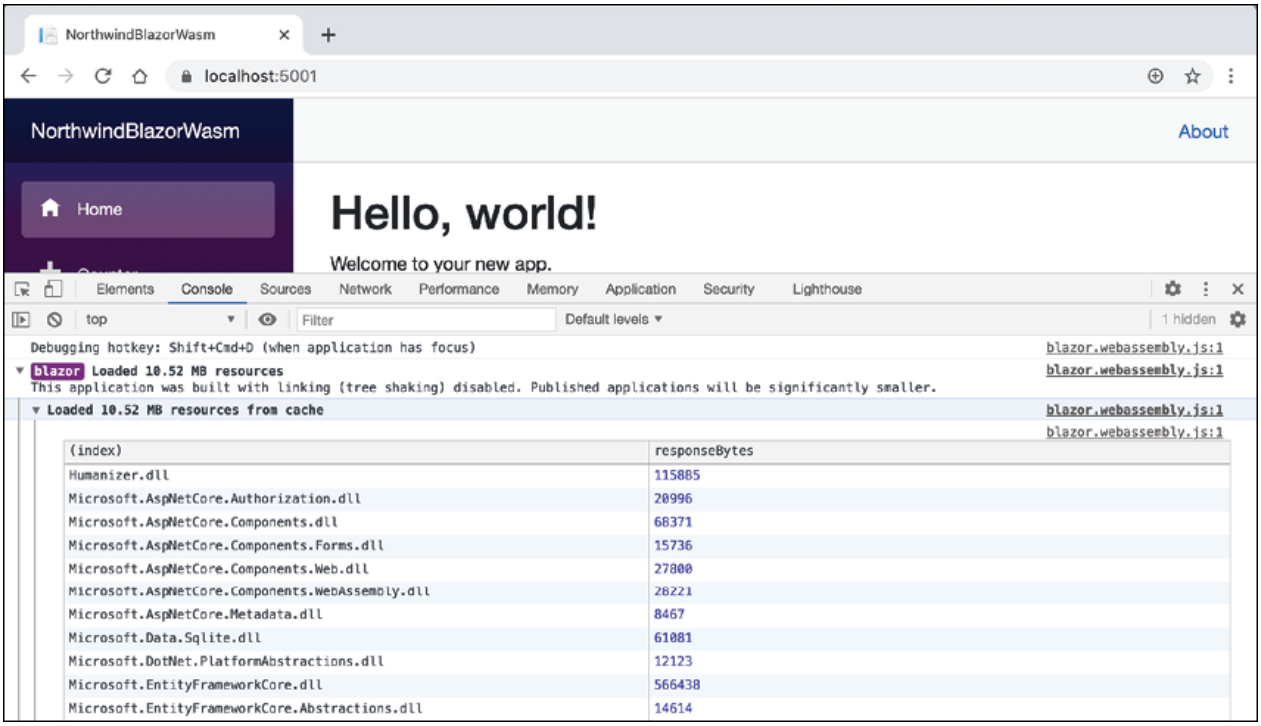


Figure 20.10: Blazor WebAssembly loading .NET 5 assemblies into the browser cache

15. Select the **Network** tab.
16. Click **Customers** and note the HTTP `GET` request with the JSON response containing all the customers, as shown in the following screenshot:

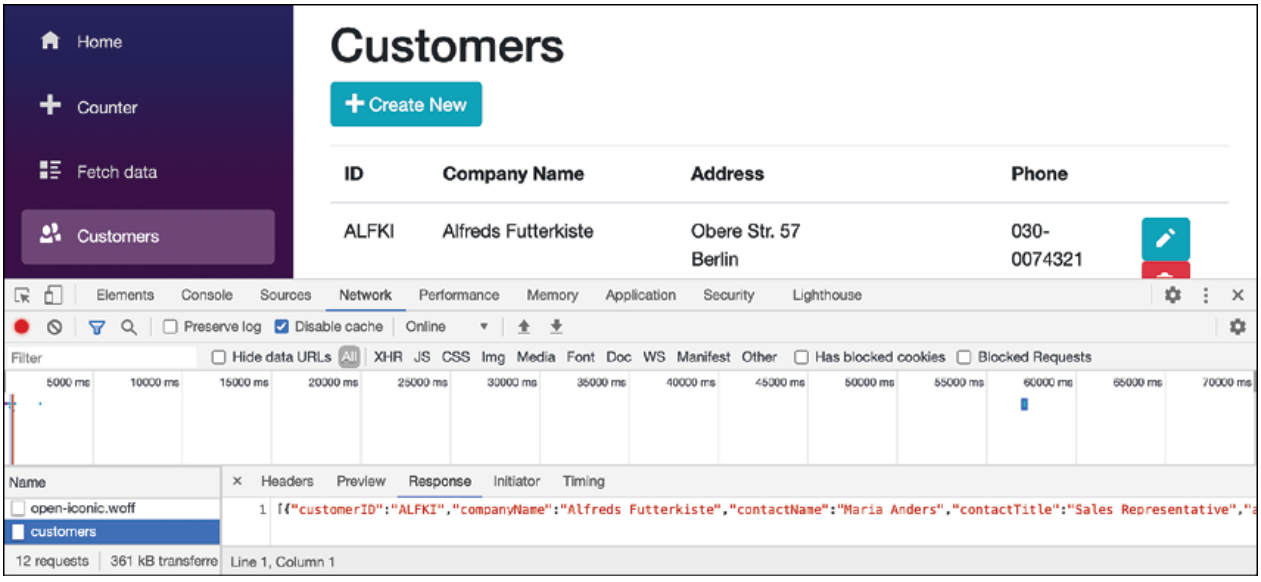


Figure 20.11: The HTTP GET request with JSON response for customers

17. Click the **+ Create New** button, complete the form to add a new customer as before, and note the HTTP `POST` request made, as shown in the following screenshot:

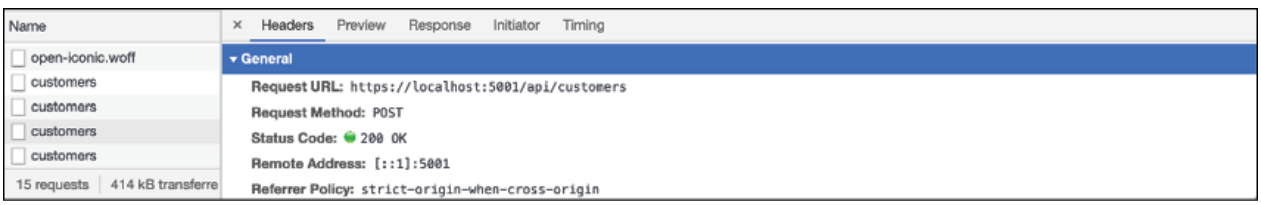


Figure 20.12: The HTTP POST request for creating a new customer

Exploring Progressive Web App support

Progressive Web App (**PWA**) support in Blazor WebAssembly projects means that the web app gains the following benefits:

- It acts as a normal web page until the visitor explicitly decides to progress to a full app experience.
- After the app is installed, launch it from the OS's start menu or desktop.
- It visually appears in its own app window instead of a browser tab.
- It works offline (if the developer has put in the effort to make this work well).
- It automatically updates.

Let us see PWA support in action:

1. In Chrome, in the address bar on the right, click the circled plus button with the tooltip **Install NorthwindBlazorWasm** and then click the **Install** button, as shown in the following screenshot:

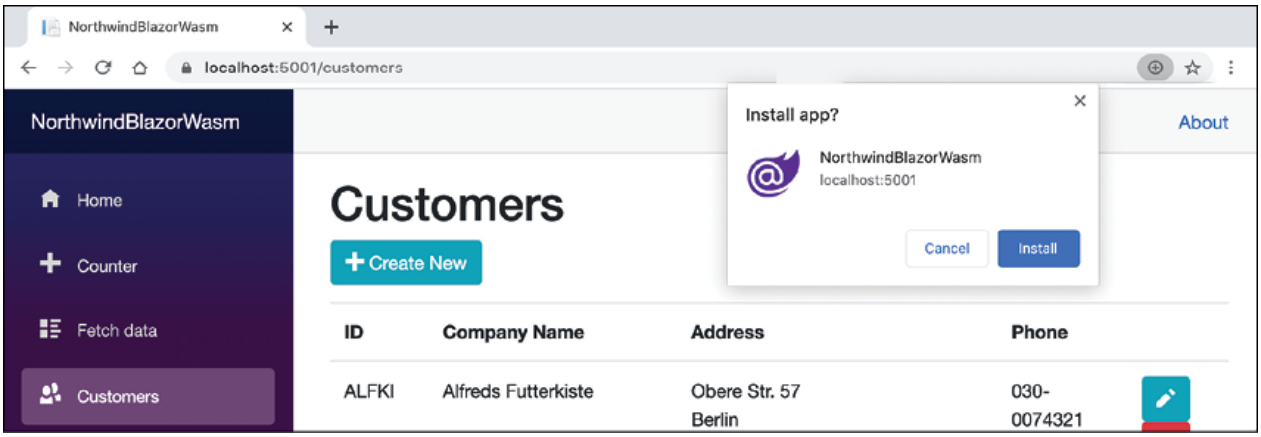


Figure 20.13: Installing NorthwindBlazorWasm as an app

2. Close Chrome.
3. Launch the **NorthwindBlazorWasm** app from your macOS Launchpad or Windows Start menu, and note it has a full app experience.
4. On the right of the title bar, click the three dots menu, and note that you can uninstall the app, but do not do so yet, as shown in the following screenshot:

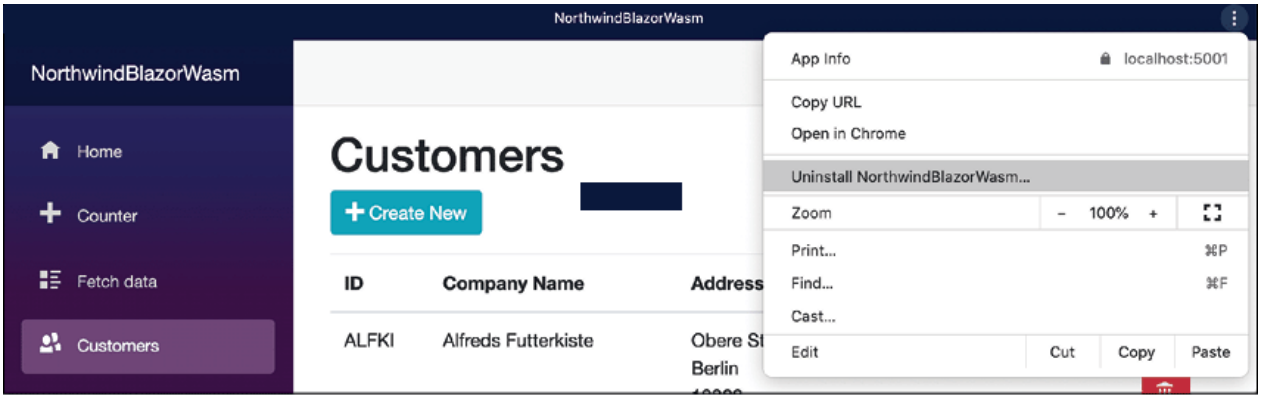


Figure 20.14: How to uninstall NorthwindBlazorWasm

5. Navigate to **View** | **Developer** | **Developer Tools** or, on Windows, press F12.
6. Select the **Network** tab, in the **Throttling** dropdown, select **Offline**, then in the app navigate to **Customers**, and note the failure to load any customers and the error message at the bottom of the app window, as shown in the following screenshot:

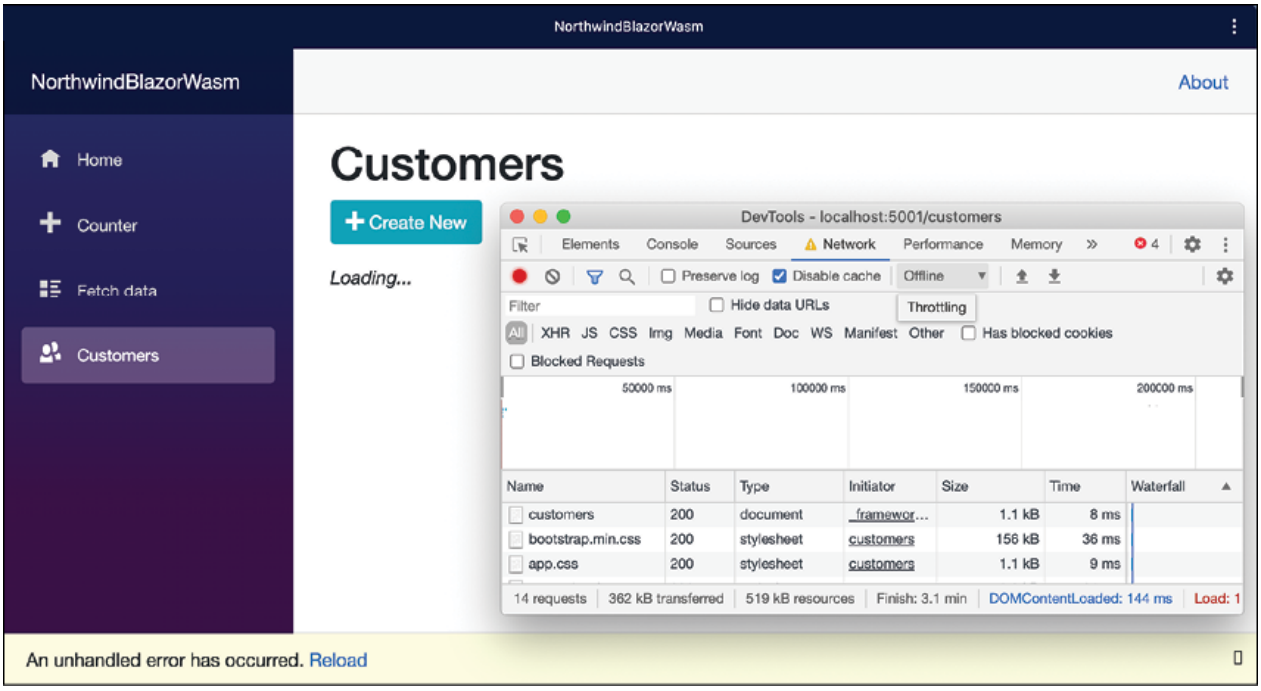


Figure 20.15: Failure to load any customers when the network is offline

7. In **Developer Tools**, set **Throttling** back to **Online**.
8. Click the **Reload** link in the yellow error bar at the bottom of the app and note that functionality returns.
9. Close the app.

We could improve the experience by caching HTTP GET responses from the Web API service locally, and storing new customers and modified or deleted customers locally, and then synchronizing with the server later by making the HTTP requests once network connectivity is restored, but that takes a lot of effort to implement well.

More Information: You can read more about implementing offline support for Blazor WebAssembly projects at the following link: https://docs.microsoft.com/en-us/aspnet/core/blazor/progressive-web-app#offline-support

Another way to improve Blazor WebAssembly projects is to use lazy loading of assemblies.

More Information: You can read about lazy loading assemblies at the following link: https://docs.microsoft.com/en-us/aspnet/core/blazor/webassembly-lazy-load-assemblies?view=aspnetcore-5.0

Explore topics

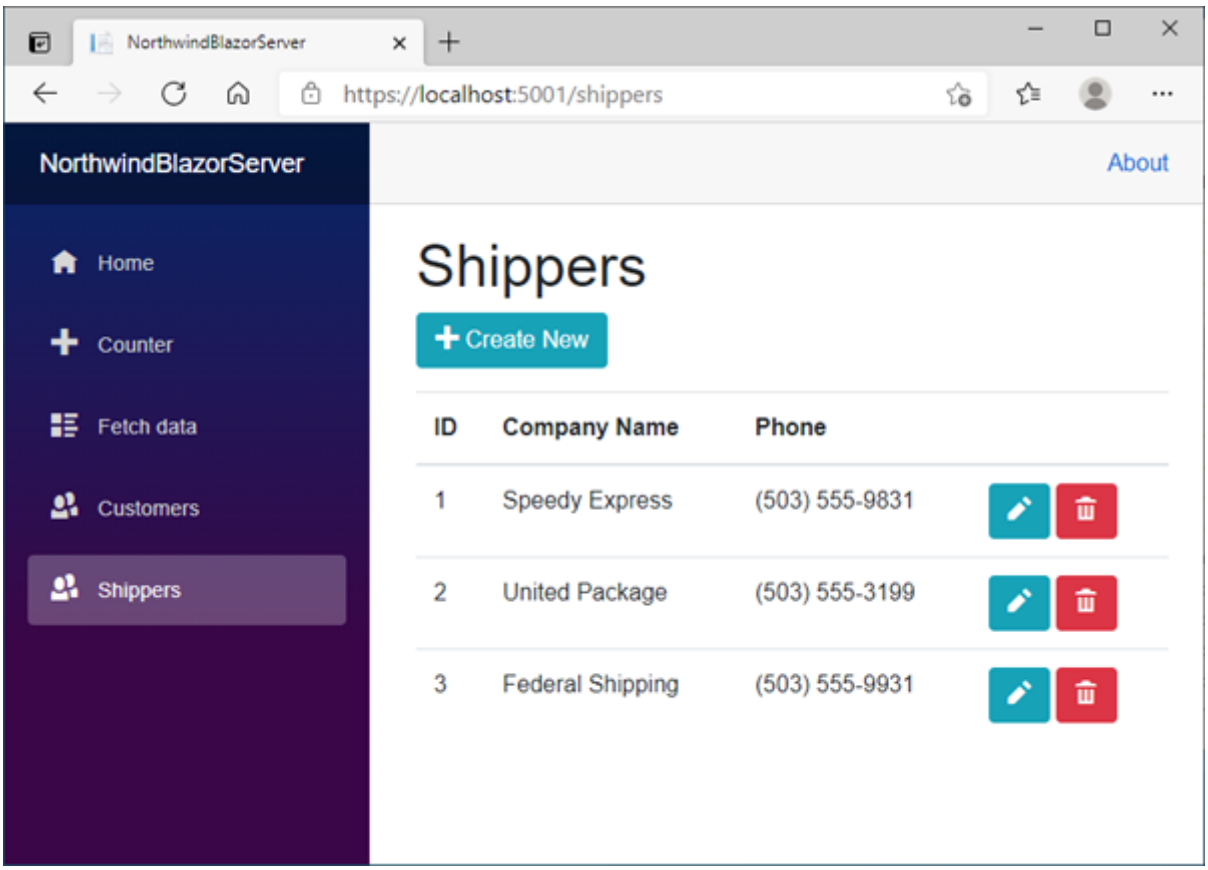
Use the following links to read more about this section's topics:

- **Awesome Blazor**: A collection of awesome Blazor resources: https://github.com/AdrienTorris/awesome-blazor
- **Blazor University**: Learn the new .NET SPA framework from Microsoft: https://blazor-university.com
- **Blazor - app building workshop**: In this workshop, we will build a complete Blazor app and learn about the various Blazor framework features along the way: https://github.com/dotnet-presentations/blazor-workshop/
- **Carl Franklin's Blazor Train**: https://www.youtube.com/playlist?list=PL8h4jt35t1wjvwFnvcB2LlYL4jLRzRmoz
- **Routing in Blazor Apps**: Comparing the routing of popular web frameworks like React and Angular with Blazor: https://devblogs.microsoft.com/premier-developer/routing-in-blazor-apps/
- **Welcome to PACMAN written in C# and running on Blazor WebAssembly**: https://github.com/SteveDunn/PacManBlazor

Exercises

Exercise 10.1.

- add a Shippers View to your Blazor Server application (NorthwindBlazorServer) that lets the client perform CRUD operations on the Shippers table.



Exercise 10.2.

- repeat Exercise 20.1, this time adding Shippers View to your Blazor WebAssembly application (NorthwindBlazorWasm)
- as before, the Shippers View should let client perform CRUD operations on the Shippers table