11

Giving Your App a Speed Boost

Software that is fully functional is not always ready enough to ship. The app must open, let the user navigate between the pages, and update the data in the UI quickly enough to satisfy the user's expectations.

Blazor WebAssembly provides a set of features for optimizing the performance of the app, in addition to some tips that you can follow in the development phase that will increase the speed of your app significantly. This chapter covers some complex and specialized topics that are useful for creating apps where high performance and efficiency are the key. You may not encounter these situations often, but they are important to learn about so you are fully equipped and ready to handle any challenge that comes your way.

We will start this chapter by introducing the `Virtualize` component, in addition to some guidelines to help increase the efficiency of your Blazor components, especially when working on large-scale and data-intensive applications. We will also learn how to control and optimize the rendering process using the `ShouldRender` method, which can significantly improve app performance. Lastly, we will talk about Blazor WebAssembly's lazy loading feature, which reduces the initial size of the app and speeds up the load time.

This chapter will cover the following points:

- Increasing components' efficiency
- Rendering optimization with `ShouldRender`
- Decreasing the initial download size with assembly lazy loading

Technical requirements

The Blazor WebAssembly code and the API project used throughout this chapter are available on the book's GitHub repository in the *Chapter 11* folder:

```
https://github.com/PacktPublishing/Mastering-Blazor-WebAssembly/
tree/main/Chapter_11/
```

Make sure to have the API solution up and running beside the Blazor app.

Increasing components' efficiency

The first thing to do is to improve the speed of the component rendering process. Because, in Blazor WebAssembly, the rendering process happens fully in the browser, controlling the process of rendering can increase the overall speed of the app, especially in large, rich, and complicated UIs.

We will start this section by learning about the out-of-the-box `Virtualize` component. Then, we will learn some tips to keep in mind while developing Blazor components to maximize their efficiency.

Virtualize component

If you have a UI that renders a table with a large number of rows or a collection of UI components through a loop, the component will be slow and laggy because of the huge number of UI elements, especially while scrolling. One way to see this problem in action is to look at a chat or social media app. Imagine if the app loaded all 5,000 messages in a chatroom, or displayed all the posts in a newsfeed at once. That would create thousands of components in the UI, which would slow down the page and make it hard to use.

The `Virtualize` component introduced in Blazor in .NET 6.0 optimizes the rendering process for large lists of elements by only rendering the ones that are visible in the screen viewport. As the user scrolls, the `Virtualize` component dynamically updates the rendered elements based on the current viewport, removing the ones that are out of view. This way, the component and scrolling are efficient and smooth regardless of the number of elements in the UI.

The usage of the `Virtualize` component is straightforward; the following code snippet shows how to render the rows of a customer list that contain 10,000 objects the normal way:

```
<table class="table">
    <thead>
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Email</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var customer in _customers)
        {
            <tr>
                <td>@customer.FirstName</td>
                <td>@customer.LastName</td>
                <td>@customer.Email</td>
```

```
            </tr>
        }
    </tbody>
</table>
```

The preceding code will result in rendering all the rows in the UI, which makes the component laggy. To use the `Virtualize` component, all you need to do is to replace the `foreach` loop with the `Virtualize` component:

```
...
<Virtualize Items="_customers">
    <tr>
        <td>@context.FirstName</td>
        <td>@context.LastName</td>
        <td>@context.Email</td>
    </tr>
</Virtualize>
...
```

The preceding code shows that the `Virtualize` component will iterate internally over the list items and render them in a performant fashion. At first, from a user perspective, it will be the same: a table of all the customer records. However, the difference will be clear when the user starts to scroll through the data; it will be smooth and efficient.

The `Virtualize` component takes four important parameters that provide more customization and performance gains:

- **ItemSize**: You can use this parameter to set the height of each item; this will increase the performance of the `Virtualize` component even more as the component won't need to do any calculations to know when to render the items near the viewport.
- **OverscanCount**: This property specifies how many items to render in advance when the user is near the edge of the visible area. This way, more items are ready to be shown when the user scrolls further. The main benefit of this property is that it reduces the flashing effect when scrolling fast, as the items are already rendered before the user sees them.
- **ItemsProvider**: This is a delegate that the `Virtualize` component triggers to fetch more items when the user scrolls; in most cases, this can be a method that fetches the data from an API.
- **Placeholder**: For items that are being fetched from the data source provided in the `ItemsProvider` parameter, `Placeholder` specifies a template to render in the meantime, until the item is fetched. This approach is common in modern apps such as social media platforms, within which, while scrolling down, more posts are fetched and you see an empty box fading in and out to let you know that the app is retrieving more content.

Structuring components for performance

It's quite tricky to manage the speed and the memory allocation of the code, and it requires some advanced skills to get it right. When it comes to the Blazor UI components, there are sets of practices and patterns that you can keep in mind while developing them.

It's good to know that each component is a fully independent object by itself, and it gets rendered in isolation from its parent and child components. Based on a test managed by the ASP.NET Core team (`https://learn.microsoft.com/en-us/aspnet/core/blazor/performance?view=aspnetcore-7.0#avoid-thousands-of-component-instances`), an average component that receives three parameters takes around 0.06 ms to render. The amount of time taken to render the component is tiny, but for complicated UIs that contain huge numbers of components to render, poor performance may appear.

Here are three tips that can improve the performance of components in UIs that contain hundreds and thousands of components to render:

- **Placing inline child components in their parent**: Generally, it's a good practice to have isolated components for code readability and maintenance. However, this may not be the best option for scenarios where performance is critical. For example, if you have a page that displays your store's orders in the last month as cards, and there are 1,500 orders in the UI, you don't need to create a separate component for the order card and render it inside a loop. Instead, you can have the card UI directly in the parent component. This can be translated into code as follows:

```
/* Instead of having it as a component */
@foreach (var order in _orders)
{
    <OrderCard OrderDetails="order" />
}

/* Consider having the OrderCard UI elements directly
  inside the foreach loop */
@foreach (var order in _orders)
{
    <div class="card">
        <h3>Number: @order.Number</h3>
        <p>Total: >
    </div>
}
```

The preceding code will save around 1,500 component rendering operations and instances in our scenario.

- **Avoiding many parameters when possible**: If the component needs to be nested, and nesting it directly in the parent component is not possible, avoid having flat parameters for each property. Every parameter received by a component is added to the total time the rendering operation takes. Combining parameters within a single object is going to help reduce the rendering time when thousands of component instances are being rendered.

  The following code snippet shows an example of a customer card component that receives parameters that can be combined in a single object:

  ```
  <h3>@DisplayName</h3>
  @* Render more properties *@
  ...
  @code {

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

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

      /* More Customer properties */
  }
  ```

  Instead, you can combine the customer properties inside a class:

  ```
  <h3>@Customer.DisplayName</h3>
  @* Render more properties *@
  ...

  @code {

      [Parameter]
      public CustomerDetails Customer { get; set; }

  }
  ```

  With all the simplicity of the preceding example, when rendering thousands of instances, each added parameter can add a massive amount of time to the overall rendering process.

- **Using fixed cascading parameters**: In the *Moving data between components* section in *Chapter 2, Components in Blazor*, we learned about cascading values and cascading parameters. By default, the `IsFixed` property is `false` for each cascading value, which means every recipient of the cascading value will have a subscription to track the changes to that value. When `IsFixed` is set to `true`, the recipients will receive the initial value of the cascading value but won't have a subscription to track future updates of it. Consider setting `IsFixed` to `false` whenever possible, as this will improve the performance, especially when the cascading value is received by huge numbers of recipients.

Optimizing the JavaScript calls

JavaScript interop from Blazor can also be a bit costly because JavaScript communication happens asynchronously by default. This is in addition to the JSON serialization and deserialization operations involved when passing parameters or returning function results between C# and JavaScript.

To avoid a bit of additional overhead when using JavaScript with your Blazor app, consider making JavaScript calls synchronously whenever possible. Like the example we gave in the *Turning an existing JS package into a reusable Blazor component* section in *Chapter 6*, *Consuming JavaScript in Blazor*, if you are using a service such as Google Analytics in your project, JavaScript calls to communicate with the **Google Analytics SDK** can be made synchronously without the need to wait until the call is finished.

Another optimization tip is to avoid repetitive JavaScript calls or calls within loops whenever possible. For instance, your app supports exporting invoices as PDFs, and there is a JavaScript function that exports an invoice object into a PDF on the client side that looks like this:

```
function exportInvoiceAsPdf(invoice) {
    // Logic to export the invoice as PDF
}
```

And there's a Blazor component that exports all the invoices in the table as PDFs:

```
foreach(var invoice in _invoices)
{
    await JS.InvokeVoidAsync("exportInvoiceAsPdf",
      invoice);
}
```

Instead, consider creating a new JavaScript method that accepts an array of invoices, iterates over the invoices, and calls `exportInvoiceAsPdf`:

```
function exportInvoicesCollectionAsPdf(invoices) {
    invoices.forEach(invoice => {
        exportInvoiceAsPdf(invoice);
    });
}
```

Now, in C#, you can make one JavaScript call instead of multiple asynchronous calls. We can also make it synchronously if the C# logic is not dependent on the result of this call:

```
JS.InvokeVoidAsync("exportInvoicesCollectionAsPdf ", invoices);
```

This little change can have a big impact on the performance of the export invoices feature.

Using System.Text.Json over other JSON packages

Either while consuming web APIs or storing objects in the browser storage, JSON is widely used in your Blazor WebAssembly apps. The process of serializing and reserializing JSON objects is a bit expensive in terms of performance, especially when it comes to large objects. `System.Text.Json` has been built from scratch by the .NET team to be the native library for dealing with JSON in your .NET apps. The main goal of the library is to improve the process of serialization and deserialization, as it's generally fast and efficient. So, the overall advice here is to use `System.Text.Json` over other packages when performance is a high priority, even though there are some trade-offs, as other libraries may be richer with more features.

Leveraging the aforementioned features and tips can noticeably accelerate your app's speed and efficiency. However, keep in mind that optimization is not always needed. For simple UIs or apps that deal with small amounts of data, having a deep level of optimization will decrease the development time and make the code a bit complicated, and the results won't be noticed. So, good optimization will give great results only if it's implemented at the right time.

Next, we will look at how Blazor re-renders the components in the hierarchy, and how we can prevent useless re-renders to keep the UI performant using the `ShouldRender` method, especially if we have hundreds and thousands of components within a page.

Rendering optimization with ShouldRender

In order to introduce the `ShouldRender` method and explain how to use it, first, we need to understand how parameters and events can affect the rendering of components in Blazor.

Blazor components exist in a hierarchy, with a root component that has child components, each child component can have its own child components, and so on. The re-render happens in the following scenario:

- When a component receives an event or a parameter changes, it re-renders itself and passes a new set of parameter values to its child components
- Each child component decides whether to re-render or not based on the type and value of the parameter values it receives:
    - If the parameter values are primitive types (such as `string`, `int`, `DateTime`, and `bool`) and they have not changed, the child component does not re-render
    - If the parameter values are non-primitive types (such as complex models, event callbacks, or `RenderFragment` values) or they have changed, the child component re-renders
- This process continues recursively down the component hierarchy until all affected components are re-rendered or skipped

This process of re-rendering is expensive, and it's very common too, as all the components are in the parent-child relationship. Blazor's `ComponentBase` class has a virtual method called `ShouldRender` and returns a `bool` value. By default, `ShouldRender` returns `true`, and the method gets called after `OnParametersSet` and before `BuildRenderTree`, which we learned about in *Chapter 1, Understanding the Anatomy of a Blazor WebAssembly Project*. If `ShouldRender` returns `false`, the component won't be re-rendered, so taking advantage of the `ShouldRender` method can prevent many useless renderings, keeping the app performant.

Let's see an actual example to understand how to use `ShouldRender` in a real-world scenario. On the **Index** page of the `BooksStore` project, we have the `DataListView` component, which renders the collection of books returned from the API as a `BookCard` component. The `Index` component passes a `List<Book>` object to `DataListView`, while `DataListView` passes a `Book` object for each `BookCard` component.

Initially, when the app runs, a welcome modal pops up on the screen. This triggers a change in the state of the **Index** page, and based on what we mentioned earlier in this section about the issue of re-rendering, the **Index** page will supply a new copy of the `List<Book>` object to `DataListView`, which will also supply a new copy of the `Book` object to each `BookCard`. So, whenever the welcome modal pops up or is closed, a change that's not related to the logic of the app causes `DataListView` and each `BookCard` to be re-rendered. The process is fast and unnoticeable because we have only 3 books, but if the API is returning 500 books, for example, every time you close the popup or click on the paging buttons that change the state of the **Index** page, you will notice that the app will lag. This is where `ShouldRender` comes into play. We know that once the books are retrieved from the API and each `BookCard` is rendered, the changes that affect the **Index** page shouldn't affect the rendering of the `BookCard` components.

Before we resolve the issue, let's see practically how each `BookCard` is being re-rendered every time the state of the **Index** page is changed.

Open the `BookCard` component in the `Shared` folder, override the `OnAfterRender` life cycle method, and add the following code:

```
...
protected override void OnAfterRender(bool firstRender)
{
    Console.WriteLine($"BookCard rendered for book
      '{Book.Title}'");
}
...
```

The `OnAfterRender` method will be triggered whenever the `BookCard` component gets rendered. Now, run the API and the Blazor project. When the page is rendered for the first time, you will see the following in the console window of **Developer Tools** in the browser:

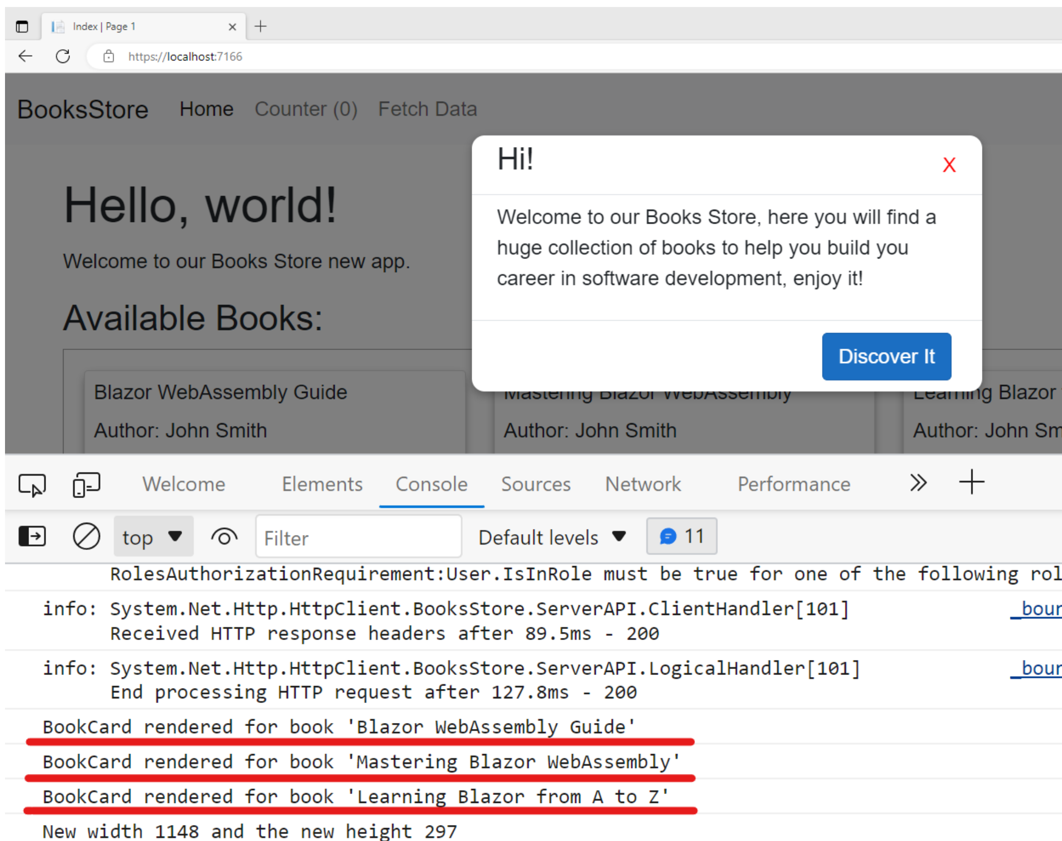

Figure 11.1 – BookCard OnAfterRender method gets triggered

That's expected, as it's the first render, but if you click the **X** button in the modal or the **Discover It** button, the state of the **Index** page will change, and you will notice the same messages will be printed again in the console window. That's because each `BookCard` gets rendered again.

The `BookCard` component won't change unless the `Book` parameter with its properties is changed; otherwise, the rendering result will stay the same during the component's lifetime. So, to prevent unnecessary re-rendering, we will do the following:

1. Declare two private fields in the component's `@code` section, one called `_oldBookId` of type `string` and another called `_shouldRerender` of type `bool`:

```
...
private string _oldBookId = string.Empty;
private bool _shouldRerender = false;
...
```

2. In the `OnParametersSet` method, which is called when the component receives new parameter values from its parent component, we compare the current `Book.Id` with `_oldBookId`. If they are different, it means that the book has changed, and the component needs to be re-rendered. We update `_oldBookId` and set `_shouldRerender` to `true`, and if the IDs match, we can skip the re-rendering, so we set `_shouldRerender` to `false` in the `else` block:

```
...
protected override void OnParametersSet()
{
    if (Book == null)
        throw new ArgumentNullException(nameof(Book));
    if (Book.Id.Equals(_oldBookId))
    {
        _oldBookId = Book.Id;
        _shouldRerender = true;
    }
    else
    {
        _shouldRerender = false;
    }
}
...
```

3. Finally, we can override the `ShouldRender` method and let it return the `_shouldRerender` value, which will be `false` most of the time, so the component won't be re-rendered:

```
...
protected override bool ShouldRender()
{
    return _shouldRerender;
}
...
```

Now, if you run the project again and monitor the console window, you will notice that the messages being written inside `OnAfterRender` will be printed only at the first rendering; then, re-rendering will be skipped.

> **Note**
>
> You shouldn't always override `ShouldRender` and prevent re-rendering. This approach should only be taken when your component UI doesn't change after the first render, regardless of the parameter values. Another example where `ShouldRender` is needed is while adding or manipulating items within a list that you don't want to be rendered until the full list is ready.

In the next chapter, we will learn how Blazor renders the components internally with `RenderTree` so we can optimize even more whenever we need to.

Next, we will cover a powerful Blazor WebAssembly feature that can shrink the initial load size of the app, which makes it load faster when it opens. This will give us fully optimized software not only during runtime but also during initialization.

Decreasing the initial download size with assembly lazy loading

In single-page applications in general and Blazor WebAssembly specifically, the full app packages, scripts, and stylesheets are downloaded for the first time to the browser so the app is fully functional on the client side. The average Blazor WebAssembly app has a size of 2 MB when it is published. That's not very big, but when the app gets more complicated and more packages and assemblies are involved, the size can increase significantly.

In this section, we will learn about **Blazor WebAssembly's lazy loading** feature, which was released in .NET 6.0 to help reduce the initial download size to make first-time loading faster. Additionally, Blazor provides some out-of-the-box features, such as runtime relinking, compression, and trimming, that decrease the app size, but we will cover those features in the *Blazor WebAssembly app prerelease final checks* section *Chapter 14*, *Publishing Blazor WebAssembly Apps*.

By default, when you run the app for the first time, all the assemblies that are used are downloaded. For huge apps with large assemblies, that could decrease the startup time until all the assemblies are ready. In .NET 6.0, Blazor introduced the lazy loading feature, which waits for some defined assemblies to load until they are required instead of loading them at startup time.

To use the lazy loading feature effectively, you should organize your solution with Razor Class Library projects when possible. For example, suppose you have an e-commerce app with a statistics page that shows charts of your sales. You can put the chart components in a separate Razor Class Library project and mark it for lazy loading. This way, the chart assembly will only be loaded when you navigate to the statistics page, not when you launch the app. This can improve the app's startup performance and save bandwidth, especially if the statistics page is rarely accessed or restricted to certain users.

Let's see a practical example of lazy loading an assembly in our `BooksStore` project. In addition to the main Blazor WebAssembly project, we have a Razor Class Library project called `BooksStore.Blazor.Components`. This project contains reusable components mostly used in the **BookForm** page, the page the admin uses to add new books to the system. We can mark the `BooksStore.Blazor.Components` package to be lazily loaded when the admin navigates to `/book/form` instead of loading it at first.

Let's see how the assemblies are being loaded first before we start achieving the desired outcome. Run the Blazor project and open a new private window in your browser, as all the DLLs and other app files are cached in the normal browser instance that you use for testing. Open the **Network** tab in **Developer**

Tools in the browser, then navigate to the **Index** page of the app (`https://localhost:7166`). You will notice all the JavaScript files and DLLs are loaded, including `BooksStore.Blazor.Components`, as shown here:

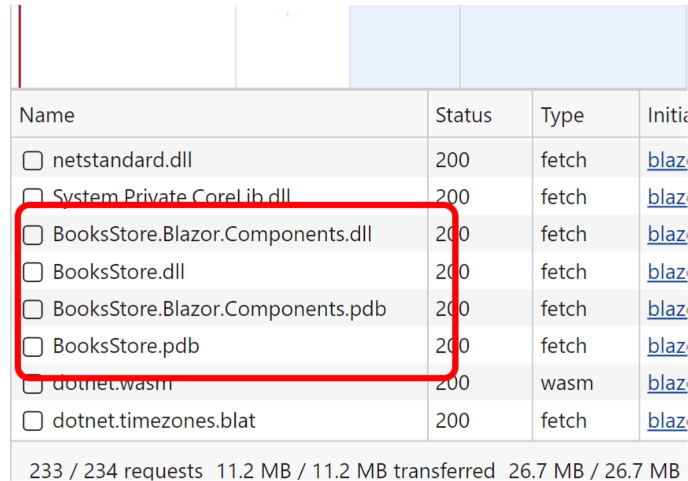

Figure 11.2 – The files and assemblies loaded for the first time

Now, let's get started with implementing assembly lazy loading for `BooksStore.Blazor.Components` so that this package is installed when we navigate to the `/book/form` page:

1. Set `BooksStore.Blazor.Components` to be loaded when navigating to the `/book/form` page.
2. Open the `BooksStore.csproj` file in the Blazor project and add the following snippet, which will mark `BooksStore.Blazor.Components.dll` to be lazily loaded:

```
...
<ItemGroup>
    <BlazorWebAssemblyLazyLoad
      Include="BooksStore.Blazor.Components.dll" />
 </ItemGroup>
...
```

   By just adding this, `BooksStore.Blazor.Components` won't be loaded at the first load of the app anymore. So, next, we need to modify `App.razor` so it will load `BooksStore.Blazor.Components` when navigating to the `/book/form` page.

3. In `App.razor`, add the following namespaces and inject `LazyAssemblyLoader` and `ILogger<App>`. `LazyAssemblyLoader` is the service that's responsible for loading the desired assemblies:

```
...
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore
  .Components.WebAssembly.Services
@using Microsoft.Extensions.Logging
@using System.Reflection;
@inject LazyAssemblyLoader AssemblyLoader
@inject ILogger<App> Logger
...
```

4. Add a `@code` section and define a `List<Assembly>` instance:

```
...
@code
{
    private List<Assembly> _lazyLoadedAssemblies =
      new();
}
```

5. Define a method that takes a `NavigationContext` parameter. This parameter contains information about the current navigation. The method will be assigned to an `EventCallback` named `OnNavigateAsync` in the `Router` component we learned about in the *Understanding routers and pages* section in *Chapter 4*, *Navigation and Routing*. The method will check whether the user has navigated to `/book/form`. If so, it will load the assembly from `BooksStore.Blazor.Components.dll` and add it to `_lazyLoadedAssembliesList`:

```
...
private async Task OnNavigateAsync(NavigationContext
  args)
{
    try
    {
        if (args.Path.Equals("book/form",
         StringComparison.InvariantCultureIgnoreCase))
        {
            var assemblies = await AssemblyLoader
              .LoadAssembliesAsync(
                    new[]  { "BooksStore
                      .Blazor.Components.dll" });
                _lazyLoadedAssemblies
                  .AddRange(assemblies);
```

```
            }
        }
        catch (Exception ex)
        {
            Logger.LogError("Error: {Message}",
              ex.Message);
        }
    }
    ...
```

6. In the markup section, assign the previous method to `OnNavigateAsync` `EventCallback` of the `Router` component, and assign the `_lazyLoadedAssemblies` list to the `AdditionalAssemblies` parameter of the `Router` component too, as shown here:

```
...
<Router AppAssembly="@typeof(App).
Assembly"    AdditionalAssemblies="_
lazyLoadedAssemblies"    OnNavigateAsync="OnNavigateAsync">
...
```

`OnNavigateAsync` will be triggered when the user navigates between pages, so, based on the written logic, when the user navigates to `/book/form`, `BooksStore.Blazor.Components.dll` should be loaded.

Now, let's make sure that what we have done is working properly. Let's run the project again, open it in a private window one more time, and monitor the **Network** tab. The first thing we should notice is that `BooksStore.Blazor.Components` is no longer loaded when we open the app for the first time:

Name	Status	Type	Initiator	Size
mscorlib.dll	200	fetch	blazor.webasse...	23.1 kB
netstandard.dll	200	fetch	blazor.webasse...	33.4 kB
System.Private.CoreLib.dll	200	fetch	blazor.webasse...	1.3 MB
BooksStore.dll	200	fetch	blazor.webasse...	35.4 kB
BooksStore.pdb	200	fetch	blazor.webasse...	49.3 kB
dotnet.wasm	200	wasm	blazor.webasse...	1.0 MB
dotnet.timezones.blat	200	fetch	blazor.webasse...	75.3 kB

Figure 11.3 – BooksStore.Blazor.Components not loaded at first launch

Then, log in with the admin account so the **Add Book** button appears in the menu. Click on the **Add Book** button, which will take you to `/book/form`. You will notice in the **Network** tab that the `BooksStore.Blazor.Components.dll` file is loaded the moment you navigate to that page:

Name	Status	Type	Initiator	Si
favicon.ico	200	x-icon	Other	
favicon.ico	200	x-icon	Other	
BooksStore.Blazor.Components.pdb	200	fetch	blazor.webasse...	
BooksStore.Blazor.Components.dll	200	fetch	blazor.webasse...	
en_US.aff	200	xhr	simplemde.min...	
en_US.dic	200	xhr	simplemde.min...	
font-awesome.min.css	200	styleshe...	simplemde.min...	

Figure 11.4 – BooksStore.Blazor.Components is loaded when navigating to /book/form

Note

You can mark multiple assemblies to be lazily loaded, and you can load multiple assemblies at the same time. The `LoadAssembliesAsync` method accepts a collection of assembly names to load, but for the purpose of our example, we need only to lazily load one assembly.

Assembly lazy loading is a powerful feature and has a clear impact on the loading time and size of an app, especially when you have an enterprise-level application with many large assemblies. One more reminder: defining a good separation in your solution's projects is a crucial factor in being able to utilize the assembly lazy loading feature. For example, components that are not always accessed, are quite large, and reference multiple packages should be placed into a different project so you can mark them to be lazily loaded.

Summary

This chapter discussed the importance of optimizing the performance of a software application beyond ensuring that it is fully functional. The focus was on Blazor WebAssembly and the various features it provides to help improve app performance. The chapter covered three main areas: increasing component efficiency, rendering optimization with `ShouldRender`, and decreasing app size with lazy loading. The `Virtualize` component was introduced, along with guidelines for improving Blazor components' efficiency, especially in data-intensive applications. The `ShouldRender` method was also discussed. We highlighted its importance in controlling and optimizing the rendering process to improve app performance. Finally, the chapter explored Blazor's lazy loading feature, which reduces the app's initial size and speeds up the load time.

After completing this chapter, you should be able to do the following:

- Understand the importance of performance in Blazor WebAssembly apps
- Use the `Virtualize` component to render a large number of UI components efficiently

- Use guidelines to keep your component optimized and performant in rich and intensive UIs
- Understand rendering in the component hierarchy and avoid useless rendering with `ShouldRender`
- Reduce the app size and improve the load time with Blazor lazy loading

In the next chapter, we will explore Blazor's `RenderTree` and examine how it updates the UI. Through practical examples, readers will gain a clear understanding of how Blazor works underneath the hood.

Further reading

- ASP.NET Core Blazor performance best practices: `https://learn.microsoft.com/en-us/aspnet/core/blazor/performance?view=aspnetcore-7.0`