

Exploring Blazor

Creating Hosted, Server-side, and
Client-side Applications with C#

Taurius Litvinavicius

Apress®

Exploring Blazor

Creating Hosted, Server-side,
and Client-side Applications
with C#

Taurius Litvinavicius

Apress®

Exploring Blazor: Creating Hosted, Server-side, and Client-side Applications with C#

Taurius Litvinavicius
Jonava, Lithuania

ISBN-13 (pbk): 978-1-4842-5445-5
<https://doi.org/10.1007/978-1-4842-5446-2>

ISBN-13 (electronic): 978-1-4842-5446-2

Copyright © 2019 by Taurius Litvinavicius

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Siddhi Chavan
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5445-5. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

Table of Contents

About the Author	vii
About the Technical Reviewer	ix
Introduction	xi
Chapter 1: Introduction.....	1
What is Blazor?	1
What is WebAssembly?	2
Blazor Types	3
Blazor Server-side	4
Blazor Client-side	4
Blazor Hosted	5
Summary.....	6
Chapter 2: Razor Syntax and Basics of Blazor	7
Differences between Razor and Blazor	7
Syntax	8
Comments	8
Sections.....	9
Blazor Binds.....	11
Bind to Element	12
code.....	13
Page Events.....	16
Summary.....	17

TABLE OF CONTENTS

Chapter 3: Blazor server-side	19
Default template overview	19
Startup	19
Injections	23
Navigation	26
Pages	27
Components	29
Parameters	31
Finished example project	33
Summary	42
Chapter 4: Blazor Client-side	43
Default Template Overview	43
Program and Startup	43
Clean Up the Template	47
Navigation	49
Components	53
Using Key to Preserve Components	56
Example	58
Summary	65
Chapter 5: Blazor hosted	67
Default template overview	67
General structure	67
Clean up the template	72
Navigation	75
API calls	77
JSONfull way	81

HTTP client manipulations.....	84
Example.....	87
Summary.....	111
Chapter 6: General Blazor.....	113
Interacting with JavaScript.....	113
Execute JavaScript Function	114
UI Events.....	116
UI Arguments	117
Local Storage	118
Where to Store?.....	119
Store Text.....	120
Store Other Types	122
Pick and Save Files.....	125
Pick File	129
Save File.....	130
Summary.....	131
Chapter 7: Practice Tasks for Server-side	133
Task 1.....	133
Description	133
Resources.....	134
Solution	135
Task 2.....	143
Description	143
Solution	144
Summary.....	151

TABLE OF CONTENTS

Chapter 8: Practice Tasks for Client-side153

Task 1 153

 Description 153

 Solution 156

Task 2..... 169

 Description 170

 Solution 170

Summary..... 178

Chapter 9: Practice Task for Blazor Hosted179

Task 1 179

 Description 180

 Resources..... 180

 Solution 184

Summary..... 193

Index.....195

About the Author



Taurius Litvinavicius is a businessman and technology expert based in Lithuania who has worked with various organizations in building and implementing various projects in software development, sales, and other fields of business. He works on a platform called MashDrop, which is a modern way to monetize the influence of an influencer. As with most of his projects, this one uses cutting-edge technologies such as Blazor. He is responsible for technological improvements, development of new features, and general management.

Taurius is also the director at the Conficiens solutio consulting agency, where he supervises development and maintenance of various projects and activities.

About the Technical Reviewer



Carsten Thomsen is a back-end developer primarily, but working with smaller front-end bits as well. He has authored and reviewed a number of books, and created numerous Microsoft Learning courses, all to do with software development. He works as a freelancer/contractor in various countries in Europe, using Azure, Visual Studio, Azure DevOps, and GitHub as some of the tools he works with. Being an exceptional troubleshooter, asking the right questions,

including the less logical ones, in a most logical to least logical fashion, he also enjoys working with architecture, research, analysis, development, testing, and bug fixing. Carsten is a very good communicator with great mentoring and team lead skills, and great skills researching and presenting new material.

Introduction

For many years the web development community has been waiting for something new, something to escape that dreaded JavaScript monopoly. Finally, the prayers have been answered – first with the release of WebAssembly and now with the release of Blazor. This book will explore Blazor in its full depth, and alongside that, you will understand what role WebAssembly plays in this whole arrangement. In fact, this is where we will begin; we will learn what Blazor is, where it runs, and how to start using it. Being a businessman with software development skills, the author has a unique view toward technologies and may base his predictions of the future for the technology not only on it being convenient to code but also on having tremendous business value. Although the technology is still young, the author has already managed and taken part in the development of a large-scale platform – mashdrop.com – and from that experience can tell you firsthand about the ease of use and efficiency of using Blazor for the project.

The book will focus on practicality and practice; therefore, you can expect lots of sample code and some exercises to complete. In fact, we will have five exercises, covering all types of Blazor, and with that, we will explore some use cases. The author believes in experiential learning; that is why, from the early stages of the book, we will be exploring Blazor by looking at code samples or folder structures of projects. Since Blazor is not a stand-alone technology, such as a programming language, the best way to learn it is to interact with it, see what it looks like in the code, and uncover some similarities with technologies using the same programming language – in this case C#. You will see, you will do, and most importantly you will learn.

CHAPTER 1

Introduction

Before you start, you need to know and prepare a few things. This is not an introductory book to C# or .NET Core development, so you should have good knowledge of C# and be able to build applications with it. It does not matter if you develop back-end applications, Windows applications, or mobile applications; as long as you use C#, you will find something familiar in Blazor. You need to install some software on your system, starting with Visual Studio 2019, followed by the latest version of .NET Core 3.0.

What is Blazor?

Blazor is a web UI framework allowing you to use C# and .NET Core on the front end. It allows you to develop your front-end logic in a couple of different ways using the C# programming language, and that is something that we will explore later in this chapter.

Technical aspects aside, think of it this way; in any standard web development project, you would need to have two people, one for the JavaScript and the other for the back end. Sometimes you also need a designer to work with HTML elements and CSS and do other design-related tasks. The Blazor technology will not remove any dependency for a designer, but it will surely remove the dependency on JavaScript. However, JavaScript can still be used with the Blazor technology.

Blazor uses the Razor syntax (C# mixed with HTML), which will be covered in the next chapter, so any familiarity with the Razor syntax will give you an edge when developing. There are some differences though, as you will see shortly. Most importantly, Razor only happens once and Blazor will happen over and over again, meaning that your C# part in Razor (.cshtml file) will only execute when the page is loaded, but in Blazor (.razor file) the code will execute on the loaded page on various events, such as onclick, onchange, and others.

It uses WebSocket to communicate with the server as well as work on the server-side, or it uses the WebAssembly technology which allows for C# to be built on the client side. This is where the different types of Blazor technology come into play.

What is WebAssembly?

WebAssembly is a technology that allows you to compile languages like C++ or C# in the browser, thus allowing Blazor to exist. It first appeared as a minimum viable product in early 2017, and while the technology is still in its early years, it is being co-developed by companies like Microsoft, Google, Apple, and others. The technology already has the support of all major browsers (<https://webassembly.org/roadmap/>), and with its growth, we can expect the support to be there for a very long time. In general, Blazor simply sends a source code file to the browser and WebAssembly compiles it into a binary file. The technology is available in all major browsers – Edge, Chrome, Firefox, Opera, and Maxthon (MX) – and the equivalent mobile versions.

WebAssembly gives you a safe, sandboxed environment, so it appears similarly as running JavaScript. Nothing is accessible from outside the specific browser tab the user is using.

Blazor Types

Server-side (see Figure 1-1) Blazor will run all the logic, mainly using WebSocket to accomplish the task. While it does give you an ability to use C# for writing front-end code, this may not be the most efficient option. You eliminate the need for API calls with this option, as you will simply inject your libraries directly into the front-end part.

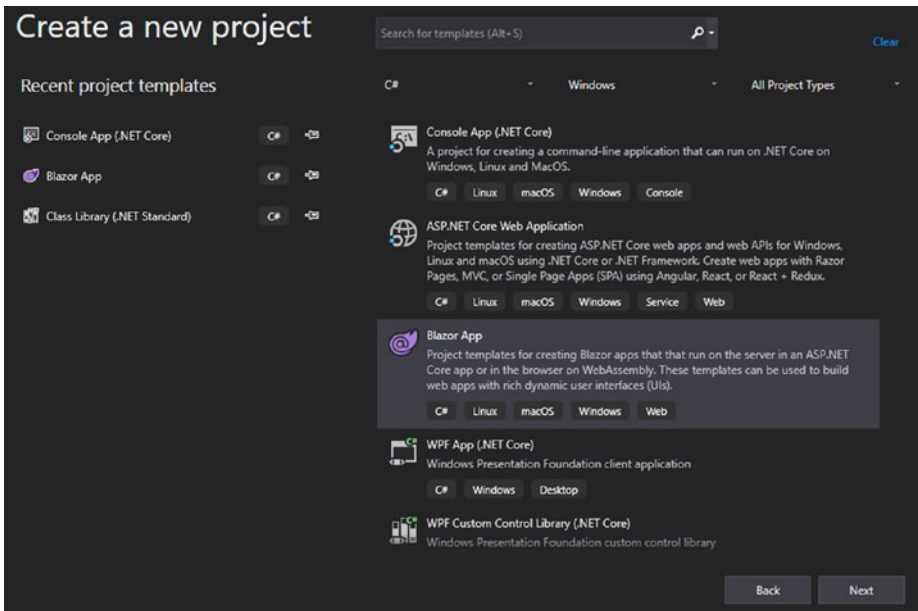


Figure 1-1. *Blazor templates*

All three types of Blazor have different templates in Visual Studio, and you should always use them for your Blazor projects no matter which type you choose. As shown in Figure 1-1, you will need to choose Blazor App project type and then choose the type of Blazor after you have picked your project location.

Blazor Server-side

While the server-side may come across as a convenience, you should still go with the client Blazor, that is, Blazor running in a browser. Server-side will use server resources, while the client will save you resources or at the very least will not waste them.

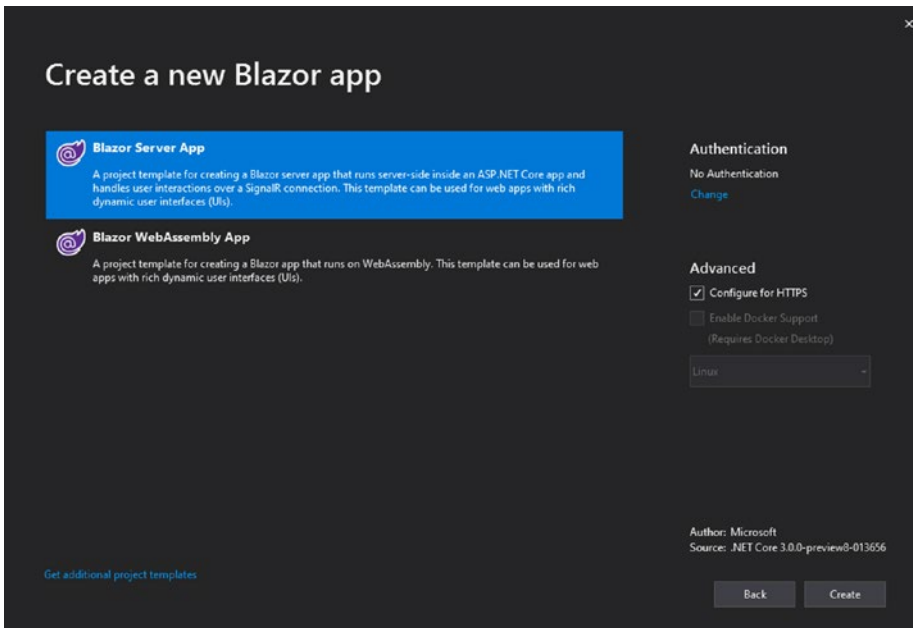


Figure 1-2. Server-side Blazor template selection

Once you get to picking the Blazor type in Visual Studio, you need to select “Blazor Server App” as shown in Figure 1-2.

Blazor Client-side

Client-side (see Figure 1-2) Blazor runs entirely on client-side in the browser. Your pages reside on the server, but it is all for client-side to handle. This is good for a presentation web site or web sites that

provide calculators and other such simple services. If you need database interactions or if you already have APIs and class libraries, this should not be your choice.

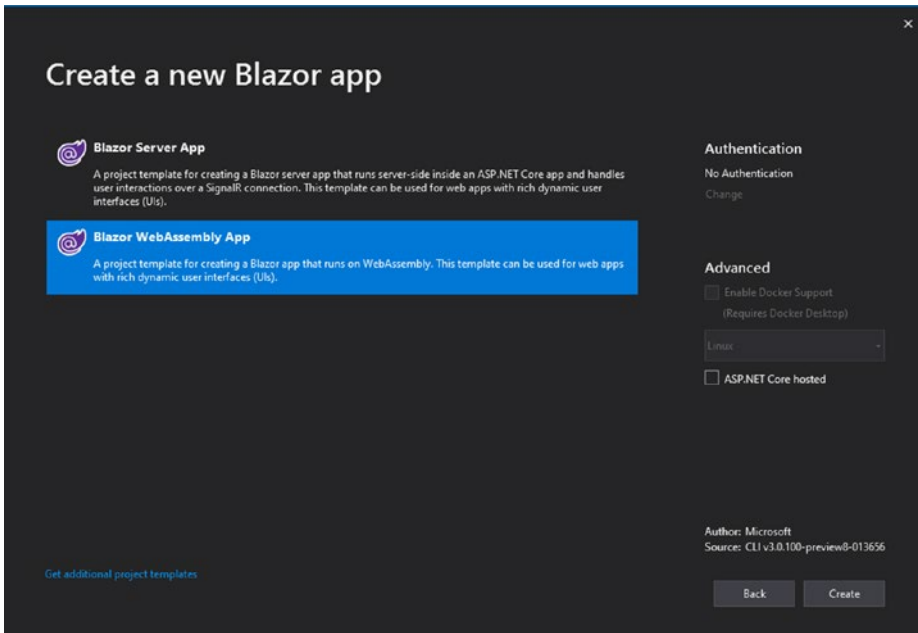


Figure 1-3. Client-side Blazor template selection

For the client-side project, you need to pick the template “Blazor WebAssembly App”. With that, the checkbox on the right side “ASP.NET Core hosted” needs to be unchecked.

Blazor Hosted

Blazor hosted (see Figure 1-3), this is probably the best type to go with as your logic will run on the browser saving those precious server resources. Basically, there are two parts, the client Blazor project and an API project. They are connected in a unique way, so you will not need to treat them as separate projects.

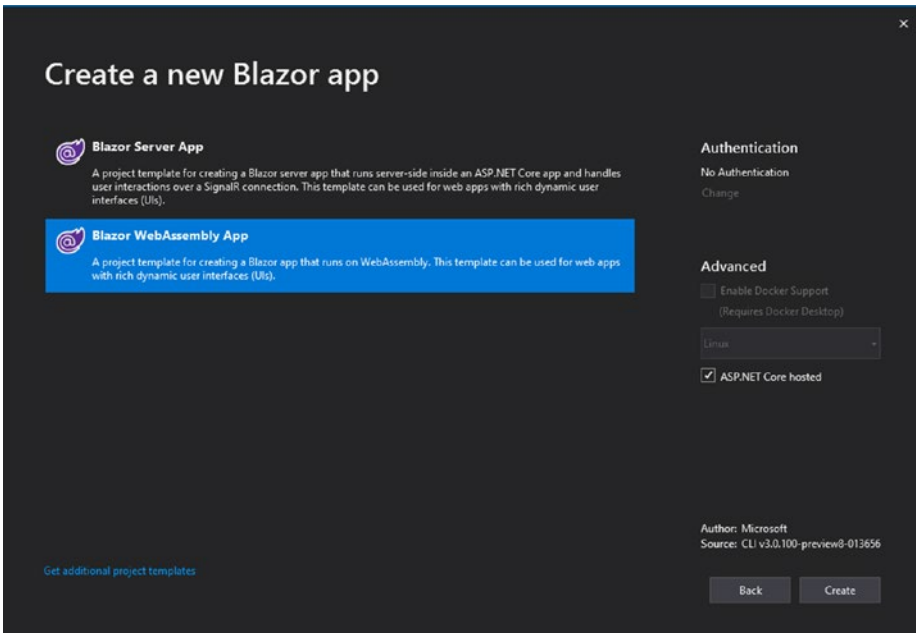


Figure 1-4. Blazor hosted template selection

If you wish to go with our final type of Blazor, you will need to select the template “Blazor WebAssembly App” (see Figure 1-4) just like you did for the client-side, but in this case you do need to check the checkbox on the right side “ASP.NET Core hosted”.

Summary

There is no best type of Blazor; as you have seen throughout this chapter, every option has its own use case. Everything depends on what your project needs right now and more importantly what it will need in the future. If you are not, simply go with the client-side, as it will be the most diverse option. In the next chapter, we will dive deeper into Blazor and explore the syntax and some other things. You will see that while the structure may be different, for the most part, coding happens in the same way for all types of Blazor.

CHAPTER 2

Razor Syntax and Basics of Blazor

This chapter will get you started with Blazor, as mentioned in the last chapter – all three types of Blazor have a lot in common and this is what this chapter is all about. Before we can go any further, we will need to look at the syntax and see how it works. Then we will get to the essentials of Blazor, such as bindings and method execution; all that will be used later in the book.

In this chapter, you will learn

- Syntax
- Element and variable bindings
- Method executions
- Use of general page events

Differences between Razor and Blazor

Simplistically speaking, the difference is that Razor will happen once on “page launch,” while the Blazor will work all the time. The loops and logic statements will get re-evaluated in Blazor, while with Razor it will only happen once.

Syntax

As mentioned previously, if you know the Razor syntax, you will know Blazor syntax. However, if you do not know the Razor syntax, this is the part for you. Blazor syntax goes into a markup file named `.razor`, which contains HTML, as well as C# code.

Comments

Even though we use HTML syntax in a Blazor file, we do not use html comments. Instead we use Razor syntax for that and get beautiful and efficient commenting system, with no comments left on a generated page.

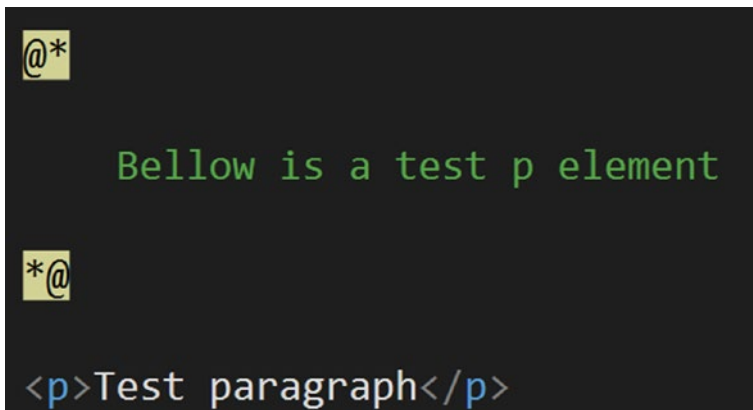
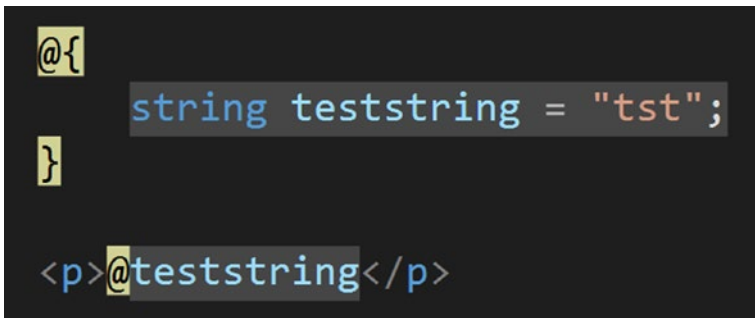


Figure 2-1. *Razor/Blazor comment syntax*

As shown in Figure 2-1, you simply start a comment section with `@*` and then end with `*@`. You can also use the standard HTML comments, but using the Razor/Blazor syntax will appear clearer on the code. The Razor/Blazor comment syntax characters get highlighted and the actual comment is displayed in a green font. The comments are compiled into the code; thus, they will not appear in the developer tools in the browser.

Sections

Razor syntax is basically C# and html code in one file, and while they do interact, you still need some things to be clearer than the others and you need some higher contrast between the two languages. That is where all the different sections come in; as you will see, they are C# dominant and they are used to highlight the C# parts of the code.



```
@{  
    string teststring = "tst";  
}  
  
<p>@teststring</p>
```

Figure 2-2. *Basic Blazor Sections*

In Figure 2-2, a variable is being declared and then it is displayed directly in a paragraph using C# code. So, for a single variable and construction of classes, you can simply use the @ sign and write everything on a single line, but if you want to do more than one line for a variable declaration, you need to create a section using @{ ... }.

At this point, this may look very simple, so let's dive into a few more examples:

```
@{
    int testint = 0;
}

@if (testint != 0)
{
    <p>Is not equal to zero</p>
}
```

Figure 2-3. *if statement syntax*

In Figure 2-3, the `testint` variable is declared and set to the value 0, followed by an if statement checking if the value of `testint` is not 0. Since the statement criteria is not satisfied, whatever is inside the if statement is not displayed. If the `testint` variable is set to any other value than 0, say 1, the HTML paragraph tag and value would be displayed. The C# code in the `@{ }` section is highlighted, and it requires no `@` sign for each line. The if statement part starts with `@` sign and creates a section similar to the previous example. This means the HTML code in the if statement section is not highlighted in any way.

```
@for (int i = 0; i < 5; i++)
{
    <p>@i</p>
}
```

Figure 2-4. *<Caption>syntax coloring*

In Figure 2-4, a for loop has been created, looping five times. Each loop creates a new paragraph tag containing the value for the current iteration, *i*. The for loop part is highlighted in a slight shade of gray, while the @ signs are highlighted in a yellow color. The HTML part inside the loop is not highlighted but the C# code is; that is how you can tell the difference between HTML markup and C# code.

Listing 2-1. Code section

```
<p>test</p>

@code {
    int a;
    double b = 2.5;

    void testmethod() {

    }
}
```

Finally, there's the code section (see Listing 2-1), where all the methods should be declared. The binding variables should also be added to the code section, which we will explain in a later chapter (Blazor binds).

Blazor Binds

Blazor allows you to bind an HTML input value to a variable and vice versa. Therefore, for the most part, we can call all bindings two-way. If you bind a text box (input type text) to a string variable, the displayed value will be the value of that string. Different elements will work differently and there are many use cases for this.

Bind to Element

Binding to an element is very simple, but not all elements can be bound, although most can.

Elements where values can be bound to variable

- Input (except for file type)
- Textarea
- Select

The listed elements are most common elements that can be bound, but others may work too.

Listing 2-2. variables

```
@code {  
    string teststring = "test value"  
    bool testbool = true;  
}
```

In Listing 2-2, two simple variables have been declared and initial values assigned, and Listing 2-3 shows how to bind them to different elements.

Listing 2-3. bindings

```
<input type="checkbox" @bind="@testbool">  
<input @bind="@teststring">  
<textarea @bind="@teststring"></textarea>
```

In Listing 2-3, a Boolean value is bound to a checkbox, checked/unchecked, and the same is true for a radio button. The string value can be bound to any text value – input, textarea, and others. When the input value changes, the variable value changes, and when the variable value changes, the value displayed in the input tag will change too.

code

The code section is where the C# methods for the front end are added. The code sections are meant to contain your code for client-side, variables, and methods. It is much like `<script>` tag in a standard HTML page, but there's more to it as shown in Listing 2-4.

Listing 2-4. Variable display

```
<p>@testvar</p>

@code {
    string testvar = "test variable";
}
```

In Listing 2-4, a variable is declared and then it is added to a paragraph tag. This is quite special, as shown in Listing 2-5.

Listing 2-5. On click event

```
<p>@testvar</p>
<p><button onclick="@testmethod">change</button></p>

@code {
    string testvar = "test variable";

    void testmethod() {
        testvar = "tst";
    }
}
```

In Listing 2-5, the same variable as in Listing 2-4 is declared and then displayed in the paragraph tag. There's also a C# method which, in client-side Blazor, will run on the front end. The method is called by declaring it in the onclick event attribute for the button tag, but parenthesis should not be used. The method simply changes the value for the variable, and in turn

what is displayed in the paragraph tag is also changed. So, that is a one-way binding, and in Listing 2-6, a two-way binding is shown.

Listing 2-6. Bind, on click and display

```
<p>@testvar</p>
<p><input @bind="@testvar"></p>
<p><button @onclick="@(() => testmethod())">change</button></p>

@code {
    string testvar = "nothing to display";
    void testmethod()
    {
        testvar = "test value";
    }
}
```

In Listing 2-6, an input tag is bound to the testvar variable, so whenever the input tag value changes, the variable will also change and therefore the display in the paragraph tag also changes. Do note that the input tag must lose focus for it to take effect.

So that is how to call a method accepting no parameters. While it is not recommended, Listing 2-7 shows how to pass and accept parameters. Your method could also be a Task method, and you would be able to await it in that lambda expression.

Listing 2-7. Method with parameters

```
<p>@testvar</p>
<p><input @bind="@testvar"></p>
<p><button @onclick="@(() => testmethod("test var"))">change
</button></p>

@code {
    string testvar = "nothing to display";
```



```

void testmethod(string testparam)
{
    testvar = testparam;
}
}

```

In Listing 2-7, the method accepts the testparam parameter. So, use parenthesis on the method call, and pass the value to call the method, rather than declaring it for the event almost like a variable. Use a lambda expression, and then use the method normally. This can be very useful if different values are needed on list output. To use a Task method, a lambda expression should be used as well.

Listing 2-8. Asynchronous task

```

<p>@testvar</p>
<p><input bind="@testvar" /></p>
<p><button onclick="@ (async () => await testmethod())">change
</button></p>

@code {
    string testvar = "nothing to display";

    async Task testmethod()
    {
        testvar = "test value";
    }
}

```

In Listing 2-8, it is quite easy to do, but it is recommended to use the await keyword and use a Task with an async method.

Page Events

Whenever the user loads a page, some events are triggered, which is common in all event-based applications, and while some technologies provide many events, Blazor, however, only provides a handful of them as shown in Listing 2-9.

Listing 2-9. Event overrides

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

@code {
    protected override Task OnInitAsync()
    {
        return base.OnInitAsync();
    }

    protected override void OnInit()
    {
        base.OnInit();
    }

    protected override Task OnAfterRenderAsync()
    {
        return base.OnAfterRenderAsync();
    }

    protected override void OnAfterRender()
    {
        base.OnAfterRender();
    }
}
```

```

protected override Task OnParametersSetAsync()
{
    return base.OnParametersSetAsync();
}

protected override void OnParametersSet()
{
    base.OnParametersSet();
}

protected override bool ShouldRender()
{
    return base.ShouldRender();
}
}

```

The first method, `OnInitAsync`, is useful for assigning initial variable values and retrieving other data before the page is loaded. The `OnInit` method is a non-task-based method equivalent to the `OnInitAsync`. The two event methods `OnAfterRender` and `OnAfterRenderAsync` can be used when needing to do some additional work after the UI elements have rendered. The final two methods are triggered when a parameter changes, which may be useful if you have search parameters or you have layout changes, such as dark and light themes.

Summary

You now know some basics of Blazor, as well as the most important parts of Blazor – bindings and method executions. In the next chapters, we will dive deeper into Blazor and explore differences between the different types. With that, we will not forget the basics, and alongside everything else, you will see something from this chapter occurring in almost every example of code in this book.

CHAPTER 3

Blazor server-side

This chapter is an introduction to the Blazor server-side technology which allows you to run your front-end logic in the back end using the C# programming language.

In this chapter, you will learn

- How Blazor server-side runs
- Handling injections
- Navigation and related matters

Default template overview

Blazor is a complex technology, containing lots of elements, but Microsoft has taken care of the complexity by supplying three different templates – server-side, client-side, and hosted. In this section, you will need to use server-side, and that is what we will explore here.

Startup

Any Blazor application is a standard web application, or in fact, it is a console application. .NET Core web applications have the usual main Program.cs class, containing the main method, and the Startup.cs class, which contains all the startup logic. In Listing 3-1, the default created Program.cs is shown.

Listing 3-1. Program.cs contents

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

namespace WebApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

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

Keep the default Program.cs as shown in Listing 3-1. Once the program is initiated, the method creates a host and sets the startup type to Startup.cs, and that is where the Blazor part begins.

Listing 3-2. Startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
```

```
using Microsoft.Extensions.Hosting;
using WebApplication1.Data;

namespace WebApplication1
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

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

            services.AddSingleton<Data.TestService>();
        }

        public void Configure(IApplicationBuilder app,
            IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
        }
    }
}
```

```

        else
        {
            app.UseExceptionHandler("/Home/Error");

            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

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

```

Most of the content of `Startup.cs` shown in Listing 3-2 is quite generic, the same content found in any .NET Core web application template. The first important part is the `ConfigureServices` method, where every line of code is used to configure Blazor. Use the `AddRazorPages` to allow Razor pages, which is required by the initial `.cshtml` file, covered in the “Navigation” section. The `AddServerSideBlazor` method handles all things server-side Blazor. The singletons are used to register services, which is covered later. In the `Configure` method, the important bit is the `UseEndpoints` lambda, where the endpoint configuration for handling Blazor is located. The fallback page is the `_Host.cshtml` file.

Injectoins

Injection samples can be seen in the default template, but we will add a custom one to see exactly what needs to be done. Figure 3-1 shows the Data folder with two services and an entity object with public properties. The TestService.cs code file is the custom service.

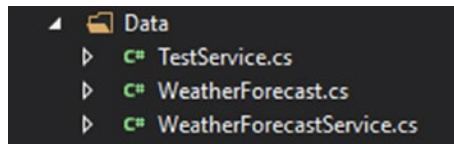


Figure 3-1. Data folder with custom service

Listing 3-3. TestService.cs

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

namespace WebApplication1.Data
{
    public class TestService
    {
        public Task<double> TestCalculationAsync(double a,
            double b)
        {
            return Task.FromResult(a * b);
        }
    }
}
```


In Listing 3-3, the `TestService.cs` file is shown, and it is a simple class, containing a static method. One important thing to do so that the service can be used in the context of server-side Blazor is to register it as service. If a service is not registered, an error is displayed in the browser window – `InvalidOperationException: Cannot provide a value for property 'tstservice' on type 'WebApplication1.Pages.Index'.` There is no registered service of type `'WebApplication1.Data.TestService'`.

Listing 3-4. Page with injected data service

```
@page "/"

@inject Data.TestService tstservice

<p><input @bind="a" /></p>
<p><input @bind="b" /></p>
<p><button onclick="@ (new Action(() =>
tstservice.TestCalculationAsync(a, b)))">test 1</button></p>
<p><button onclick="@ (new Action(() =>
testcalculation()))">test 2</button></p>

<p>@todisplay</p>
@code {
    double a;
    double b;
    double todisplay;

    async Task testcalculation()
    {
        todisplay = await tstservice.TestCalculationAsync(a, b);
    }
}
```

The `Index.razor` file, shown in Listing 3-4, has been customized with two implementations of the service, one synchronous and the other asynchronous, both invoked by clicking or tapping a button. To use the service, it must be injected, and the injection works the same way as simple construction. It's possible to access methods and other members of the class. The injection starts with keyword `@inject`, followed by the namespace and class name, using the usual dot notation, and finally the object name is assigned. All of that is done in a single line of code.

Listing 3-5. Service registry

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

    services.AddSingleton<Data.TestService>();
}
```

In `Startup.cs`, shown in Listing 3-5, the `ConfigureServices` method is used to register the custom services by using the `AddSingleton` method of the `IServiceCollection`.

The `Index.razor` page (Listing 3-4) contains two buttons, and they are bound to two double variables, `a` and `b`. There's also one paragraph displaying the resulting double value. Methods from injections can be called directly in the button's `onclick` event; however, if you need to assign the return value, it is better to create a method in the Listing section and execute your injection method there. The button with content "Test 1" executes the method directly, while the button with "Test 2" executes the local method in the code, which in turn assigns the return value to the `todisplay` variable, and it gets displayed.

Navigation

The navigation is simple and straightforward; as you will soon see, it is a matter of only one for the most part. Blazor allows you to open a page in the layout, but it also allows you to use a page as a component, or in other words as an HTML element.

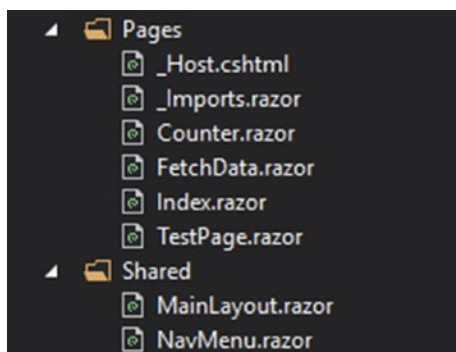


Figure 3-2. *HTML and Razor pages*

As shown in Figure 3-2, there are two folders, Pages and Shared, for storing HTML and Razor pages. The Shared folder should be used for layout elements and the Pages folder for pages. The MainLayout.razor file contains the main layout, containing one little piece where all the other pages get displayed, and NavMenu.razor is a component that goes into MainLayout.razor. Getting to pages, we only need to look at _Host.cshtml, _Imports.razor, and then Index.razor. The Counter.razor and FetchData.razor are sample pages embedded in the template, and TestPage.Razor was created for the purpose of this book. We will start by looking at the MainLayout.razor and see how the pages get displayed.

Listing 3-6. Layout component

```
@inherits LayoutComponentBase

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

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

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

In the content of `MainLayout.razor`, shown in Listing 3-6, provided in the template, you only need to notice two things: inheritance of `LayoutComponentBase` and the `@Body` which comes from `LayoutComponentBase`. Once you have those two, you only need to know that wherever the `@body` is, that is where your pages will be displayed. Now, let us look at how you get to your main layout.

Pages

Now that we know where the pages will appear, we need to find out how they will get there. As mentioned previously, it is only a matter of one line to establish a route to a page.

Listing 3-7. Html entry point page

```

@page "/"
@namespace WebApplication1.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
    initial-scale=1.0" />
    <title>WebApplication1</title>
    <base href="~/>
    <link rel="stylesheet" href="css/bootstrap/bootstrap.
    min.css" />
    <link href="css/site.css" rel="stylesheet" />
</head>
<body>
    <app>@(await Html.RenderComponentAsync<App>(RenderMode.
    ServerPrerendered))</app>

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

```

Server-side Blazor does not run C# code in the browser; therefore, it requires an entry point from Razor to Blazor. You will find that in `_Host.cshtml`, shown in Listing 3-7. This page is to be used as it is and not to be changed. Now, if you do need JavaScript, this is where you reference it, same with the stylesheets. `RenderComponentAsync` method initiates the component `App.razor`, and in there you will find something that handles all the page routes.

Listing 3-8. Default contents of App.razor

```

<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>

```

The `NotFound` component (see Listing 3-8) will come in use when someone tries to access a route that does not exist, and it should not be modified. However, you may change the contents of `NotFound` and add your custom error pages in there. Now that we are done with defaults, we will take a look at something more custom.

Components

Components can be placed anywhere in the HTML code just like you would do it with any HTML element; it just does not have any default features – events or values. A component is the same kind of file as page is; in fact, a page can be used as a component and vice versa. You will find them particularly useful when displaying list outputs, and we will study them throughout the book.

Listing 3-9. Component parameter

```

<h3>TestComponent</h3>
<p>@testparameter</p>

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

```

As you can see in the content of the `TestComponent.razor` page in Listing 3-9, it is quite straightforward like any other page. The one difference is the parameter, and that is quite a difference; it works as any parameter on HTML element would, as if it was “style”, “class”, or “href”, except in this case it is custom. Here we have a simple string, which gets displayed in the paragraph. Be aware that even though this looks like any HTML element, it does not function like one. There are no default parameters like “style” or “class”; therefore, if you do need to use them, the best way to do it is to wrap the contents of component in a `<div>` tag and pass the parameters to that element.

Listing 3-10. Test page for component parameter

```

@page "/TestPage"

<p>New test page</p>

<p><WebApplication1.Components.TestComponent
testparameter="this is the test parameter for test
component"></WebApplication1.Components.TestComponent></p>

```

Every page must have a route as shown in Listing 3-10 (first line), unless you use the page as a component. The declaration of the route is very simple; you need to use `@page` followed by the route. This particular page also has a component on it, declared as you would declare a class, using the full namespace using dot notation. The component has a set parameter, `testparameter`.

Parameters

Handling parameters is very important. You can use parameters to determine different states and pass authentication tokens, discount coupons, and more. This can be very straightforward or a bit complicated, you will have to choose one of two ways according to your needs.

Listing 3-11. Parameter display

```
@page "/TestPage1/{param1}/{param2}"

<p>Parameter 1</p>
<p>@param1</p>

<p>Parameter 2</p>
<p>@param2</p>

@code {
    [Parameter]
    public string param1 { get; set; }

    [Parameter]
    public string param2 { get; set; }
}
```

The first option is to list the parameters in the route for the page, as shown in Listing 3-11. You access them by declaring a variable in the `@code` section and using the attribute `[Parameter]`; the name of the variable must match the name in the route. However, the downside of this option is that you have to have all your variables in a correct order, so if you have two mandatory parameters, it will work fine. But what if you have three parameters and none of them are mandatory, you can have one, two, or all three? The answer is the second, more complex option. Also, you can have several different routes with different parameters declared or one route with no parameter; if you only have one route and parameter is missing, the route will not be found.

Listing 3-12. Page with navigation manager

```
@page "/TestPage2"
@Inject NavigationManager navmanager
@using System.Web;

<p>Parameter 1</p>
<p>@param1</p>

<p>Parameter 2</p>
<p>@param2</p>

@code {
    string param1;
    string param2;

    protected override Task OnInitAsync()
    {
        var qparams = HttpUtility.ParseQueryString(new
            Uri(navmanager.GetAbsoluteUri()).Query);

        param1 = qparams["param1"];
        param2 = qparams["param2"];

        return base.OnInitAsync();
    }
}
```

The TestPage2.razor page shown in Listing 3-12 contains two variables in the @code section, param1 and param2. On initialization of the page, we first need to get a query from the current url. To do that, you will need to inject NavigationManager and use the method GetAbsoluteUri that goes into the constructor for a new uri from which you access the property Query. The Query is passed to ParseQueryString method, which can be found in System.Web.HttpUtility, and it returns a key value pair with your query string parameters.

Finished example project

Now, we will take a look at another Blazor project that will bring all navigation tricks together. The example is quite simple, a basic presentation web site containing Home page, What we do page, and a contact page.

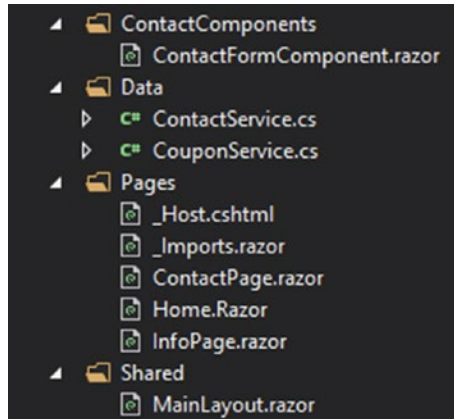


Figure 3-3. *Finished example project*

Before we begin, we will remove all the unnecessary defaults, and Figure 3-3 shows what it will look like once everything is removed and additional files are added.

Listing 3-13. Layout page

```
<div style="width:100%" >
<label></label>
<a href="" >Home</a>
<a href="InfoPage" >What we do</a>
<a href="ContactPage" >Contact us</a>
</div>
<div style="width:100%" >
@Body
</div>
```

Our layout, shown in Listing 3-13, is very basic, but instead of using the side bar as you can see in the template (Listing 3-6), we only have a top bar for navigation between pages.

Listing 3-14. Home page

```
@page "/"
<div style="width:33%">
    <h1>Painting</h1>
</div>
<div style="width:33%">
    <h1>Sketching</h1>
</div>
<div style="width:33%">
    <h1>Paints and other products</h1>
</div>
```

For the home page, shown in Listing 3-14, we only have some simple HTML to fill the space and the default route. Once we have the home page, we can do something a little more interesting, finish the info page.

Listing 3-15. Program.cs contents

```
using System.Collections.Generic;

namespace WebApplication1
{
    public class Program
    {
        public static Dictionary<string, int> couponDictionary
            = new Dictionary<string, int>();
        public static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
```

```

        {
            var rnd = new Random();
            couponDictionary.Add(Guid.NewGuid().ToString(),
                                rnd.Next(10, 70));
        }
        CreateHostBuilder(args).Build().Run();
    }
}
}

```

For testing, we will generate ten random coupons by simply establishing a dictionary in Program.cs as shown in Listing 3-15 and assigning them Guid values as their id and a random value for the discount amount. You can break after the loop to get a test value for a single coupon.

Listing 3-16. Coupon service

```

using System.Threading.Tasks;

namespace WebApplication1.Data
{
    public class CouponService
    {
        public Task<int> CheckCoupon(string coupon)
        {
            try
            {
                int discountvalue = Program.
                    couponDictionary[coupon];
                return Task.FromResult(discountvalue);
            }
        }
    }
}

```

```

        catch
        {
            return Task.FromResult(0);
        }
    }
}

```

In Listing 3-16, our service simply contains one method that returns an integer, which is the value of your discount coupon. In case the coupon is not found, the error is handled and 0 is returned.

Listing 3-17. Service registry

```

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

```

As always, you need to register your service in the Startup.cs, as shown in Listing 3-17; if you do not do this, you will get an error.

Listing 3-18. Information page

```

@page "/InfoPage/{coupon}"
@page "/InfoPage"
@Inject Data.CouponService cpnservice

<div style="width:33%">
    <h1>Painting</h1>
    <p>We do it quickly</p>
</div>

```

```

<div style="width:33%">
    <h1>Sketching</h1>
    <p>We do it accurately</p>
</div>
<div style="width:33%">
    <h1>Paints and other products</h1>
    <p>Highest quality</p>
</div>

@if (discountvalue != 0)
{
    <div>
        <p>You are eligible for @discountvalue % discount</p>
    </div>
}

@code {
    double discountvalue = 0;

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

    protected override async Task OnInitializedAsync()
    {
        discountvalue = await cpnservice.CheckCoupon(coupon);
    }
}

```

The InfoPage shown in Listing 3-18 contains two routes because the coupon parameter is optional. However, since we only have one simple parameter, we can use the default way (Listing 3-11) to implement it. With that, we have some basic HTML just to see the difference between pages. After that, we get to the coupon check.

First the coupon service is injected under name `cpnservice`, and then we set up a variable for the discount value and an if statement to check if it is equal to 0 or not. The discount box is displayed and the value is shown in the paragraph if the value is different to 0. We only evaluate the variable once and that is when the page loads. It is done by simply executing `CheckCoupon` and assigning the return value to the `discountvalue` variable.

After we are done with the `InfoPage`, we will get to the contact page which will contain general information and a contact form component. First, we need to establish the component which will also require a service for sending the message to the email.

Listing 3-19. contact us page

```
using System;
using System.Net.Mail;
using System.Threading.Tasks;
namespace WebApplication1.Data
{
    public class ContactService
    {
        public Task<bool> SendMessage(string name, string
            email, string messagebody)
        {
            try
            {
                MailMessage message = new MailMessage(email,
                    "yoursupportemail", "question by " + name,
                    messagebody);
                message.IsBodyHtml = true;
                SmtpClient client = new
                    SmtpClient("emailclient", 465);
                client.EnableSsl = true;
                client.Timeout = 30;
```

```

        client.Credentials = new System.
        Net.NetworkCredential("youremail",
        "youremailpass");

        client.Send(message);
        return Task.FromResult(true);
    }
    catch (Exception ex)
    {
        return Task.FromResult(false);
    }
}
}
}

```

As shown in Listing 3-19, this service also gives you an example on how you would send your email. We only have one simple method in the service which takes in name and puts it in the title of the email, then takes an email, and sets it as destination address, and finally the message is taken and inserted in the body of the email. The return of the service method is very simple; you either have success (true) or failure (false) of sending the email.

Listing 3-20. Service registry

```

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

```


Before you go any further, you need to register your service, as shown in Listing 3-20. After you do that, we can move on to the contact form component.

Listing 3-21. Contact component

```
@inject Data.ContactService cservice

<p>Name</p>
<p><input bind="@name"></p>

<p>Email</p>
<p><input bind="@email"></p>

<p>Message</p>
<p><textarea bind="@message"></textarea></p>

<p><button onclick="@ (new Action (() =>
SendMessageAsync()))">Send</button></p>

@if (displayboxopened)
{
    <p>@displayboxmessage</p>
}

@code {
    public string name;
    public string email;
    public string message;

    public bool displayboxopened = false;
    public string displayboxmessage;
```

```

public async void SendMessageAsync()
{
    if (await cnservice.SendMessage(name,email,message))
    {
        displayboxmessage = "Message sent succesfully";
        displayboxopened = true;
        await Task.Delay(7000);
        displayboxopened = false;
    }
    else
    {
        displayboxmessage = "Sending failed, try again";
        displayboxopened = true;
        await Task.Delay(7000);
        displayboxopened = false;
    }
}
}

```

The component shown in Listing 3-21 is filled with lots of useful things and neat tricks; we will get to those in due time. First, we need to notice the injection of the service, and since this is a component, there is no route. There are three input boxes and three variables, and as you can see, the textarea tag can also be bound just like the input tag. With that, we have another set of variables, one for checking if a display box should be displayed (`displayboxopened`) and the contents of it – that is either success message or failure warning. Of course, this is quick way, but it is a crude way; if you want the box to pop up in a more fashionable manner, you should insert or remove different classes of CSS. The component only has one method, which is executed on the click of the button. It simply sends the message and returns a Boolean value which is evaluated in the if statement.

The interesting thing here is the disappearing notification box; as you can see, we are easily doing that simply by adding a delay and then changing the state of the box back to hidden.

Listing 3-22. Contact us page

```
@page "/ContactPage"
<div style="width:50%">
    <p>We are located in</p>
    <p>Address, City, Country</p>
</div>

<div style="width:50%">
    <WebApplication1.ContactComponents.ContactFormComponent>
    </WebApplication1.ContactComponents.ContactFormComponent>
</div>
```

Finally, we have our contact page as shown in Listing 3-22 with a simple route declaration. With that, you will find some location information, and on the side of the page, we have our contact form component inserted.

As you can see, we use no JavaScript at all. This will save you a lot of time, and in such a case, you would only need to hire a designer to do the CSS for HTML; everything else can be done by you in C#.

Summary

When it comes to specific types of Blazor, you will eventually notice how they are quite similar in general. The main difference is how they start and how they function; one thing to remember about the server-side is that it runs on the server-side. In the next chapter, we will cover the client-side Blazor, a type of Blazor that runs on the browser.

CHAPTER 4

Blazor Client-side

In the previous chapter, we have covered a type of Blazor that runs your front-end logic on the server; in this case, we will cover a type that runs directly in the browser.

In this chapter, we will learn

- Startup of the app
- How to clean up the template
- Components
- Reusing and removing components

Default Template Overview

We will now look at the default template and understand how the client-side Blazor works. You will find some interesting differences between this one and the server-side template, as well as you will learn how to customize things when needed.

Program and Startup

As usual, the web application is a console application, and it starts with a `Program.cs` and the main method in it. The difference here is that this runs on the browser, not on the server, so the host builder will be completely different. For the most part, Blazor server and client options are exactly the same; the major difference is in startup of things.

Listing 4-1. Program.cs

```

using Microsoft.AspNetCore.Blazor.Hosting;

namespace WebApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IWebAssemblyHostBuilder
        CreateHostBuilder(string[] args) =>
            BlazorWebAssemblyHost.CreateDefaultBuilder()
                .UseBlazorStartup<Startup>();
    }
}

```

You can see that the program (Listing 4-1) does create a host builder, but it is a web assembly host builder. This whole thing works by default; therefore, it is not necessary to look at it too much. One thing to note, if you want to do something in the main method (e.g., get environment variable), you need to do it after the `CreateHostBuilder`. If you do anything before, the application will not load. The more interesting part is the `Startup.cs`.

Listing 4-2. Index.html – entry page

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">

```

```

<title>WebApplication1</title>
<base href="/">
<link href="css/bootstrap/bootstrap.min.css"
rel="stylesheet">
<link href="css/site.css" rel="stylesheet">
</head>
<body>
    <app>Loading...</app>

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

```

Listing 4-2 shows the index.html contents, the file found in wwwroot folder. These contents are generated for the template and should not be modified with the exceptions mentioned later.

Listing 4-3. Startup.cs

```

using Microsoft.AspNetCore.Components.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace WebApplication1
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection
services)
        {
        }
    }
}

```

```

        public void Configure(IComponentsApplicationBuilder app)
        {
            app.AddComponent<App>("app");
        }
    }
}

```

The startup is very straightforward and very empty; however, you can still use it for configuration. The important thing, again, is not to forget that this all happens on client-side. The main part of this is the `AddComponent<App>` (Listing 4-3), which basically adds our `App.razor` to the `<app></app>` (Listing 4-2) element in the html (Listing 4-2). In the template, this would replace the text “Loading...”; however, in real-world application, you might want to put an image with a loader gif there. It is likely that bigger apps will take a few seconds to load; therefore, it may not be too pretty to just have text there.

Listing 4-4. App.razor

```

<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>

```

As you can see, the `App.razor` file (Listing 4-4) contains a router which handles all the routing. You will also find a `NotFound` placement and remember that at this point you are already using Blazor; therefore,

you could add your error message as a custom component that you have created. The actual pages are generated where the `@body` is placed, and that requires an inheritance from `LayoutComponentBase`.

Clean Up the Template

Blazor projects provide a nice template to get a basic overview on how Blazor technology works. However, in most cases you will not need a lot of the default stuff. Therefore, you need to understand how to clean it up, so that every major part stays as template suggests, but the non-essentials disappear.

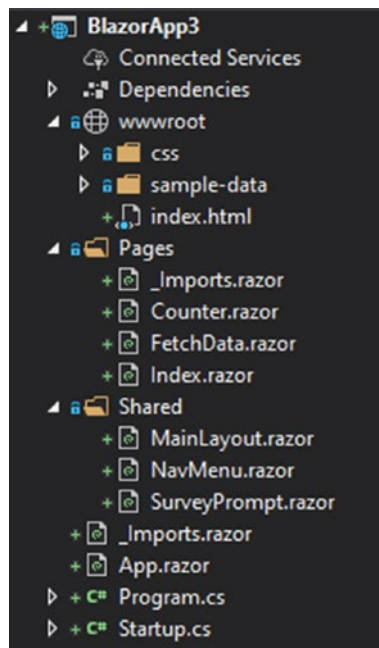


Figure 4-1. *Blazor client-side default template contents*

Figure 4-1 shows the contents of the default project template. Everything is as generated for the template, and nothing has been modified yet.

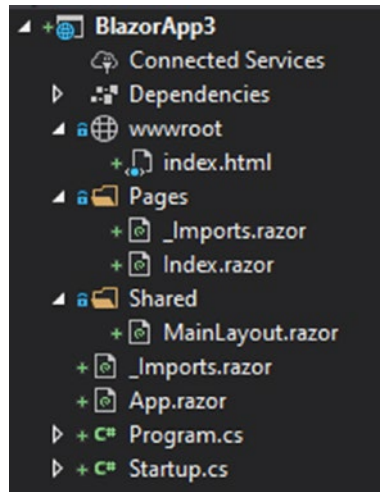


Figure 4-2. Client-side template after cleaning

If you want to clean up your template and leave only the essentials, you will need to start with files. In short, you will need to go from Figure 4-1 to Figure 4-2, and you will achieve that by removing `css` folder in `wwwroot`, `sample-data` folder in `wwwroot`, both sample `.razor` pages in the `Pages` folder, and everything in `Shared` folder except for `MainLayout.razor`.

Listing 4-5. Cleaned layout template

```
@inherits LayoutComponentBase

<div>

    @Body

</div>
```

In terms of deleting code, you only need to clean `Index.razor` and leave only `@page` for the route and clean up the `MainLayout` (Listing 4-5) as shown earlier. Of course, you will want to use your custom navigation structure in the main layout, but this is a good place to start.

Navigation

The navigation works the same way as it does on the server-side; you have to use the `@page` to set the route for a page and you can pass parameters in two different ways. We will take a few more samples on how you can structure your navigation.

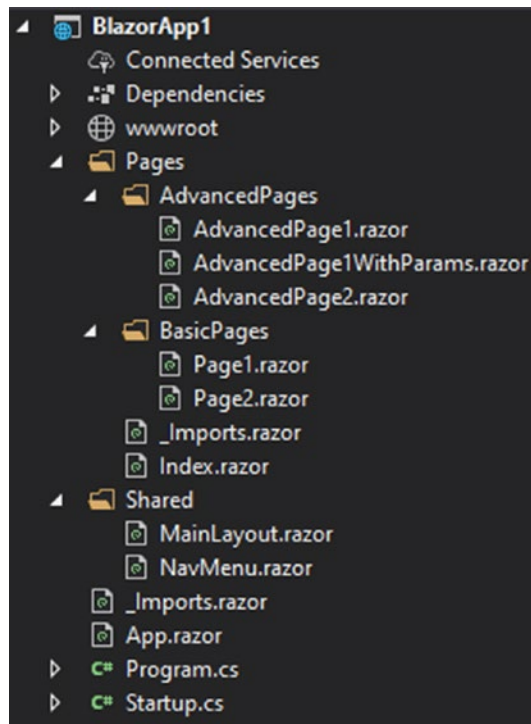


Figure 4-3. Project structure for the navigation example

For this example, we have a total of five-page files (Figure 4-3). We will see how to work with routes and how to navigate from C# code directly. You can already see that it is possible to put pages in folders; this is especially helpful when you have large amounts of pages. In this case, we can clearly see where our basic pages will be found and where our advanced pages will be.

Listing 4-6. Basic page

```
@page "/page1"  
  
<p>Basic page 1</p>
```

Listing 4-7. Page with navigation manager

```
@page "/page2"  
@inject NavigationManager navmanager  
  
<p>Basic page 2</p>  
<p><button @onclick="@TestNavigate">Go to page 1</button></p>  
  
@code {  
    void TestNavigate()  
    {  
        navmanager.NavigateTo("/page1");  
    }  
}
```

Our first basic page (Listing 4-6) is a very simple one; it only has a route and a statement saying that it is page 1. You have seen such a page many times before, and you will find it many times in the future; the more interesting one is the second basic page (Listing 4-7). This one also has a route, but this page will allow us to navigate using C#. To do that, we need to use `NavigationManager`, which is also useful for parameter management – something that was covered in the previous chapter. To navigate, you simply use `NavigateTo` method which takes the route for your page. In this example, on the button click we would navigate to the first basic page.

Listing 4-8. Advanced page

```
@page "/advancedpage1"

<p>This is the advanced page 1</p>
```

Listing 4-9. Advanced page with parameters

```
@page "/advancedpage1/{param1}"

<p>This is the advanced page 1</p>
<p>parameter: @param1</p>

@code {
    [Parameter]
    public string param1 { get; set; }
}
```

With the first advanced page (Listing 4-8), we want to explore pages with similar routes. The first page has only a simple route and the name for the page displayed. For the second page (Listing 4-9), we have the same route with the exception of a parameter which is displayed in the page. You can use it when you have a page which is displayed completely differently when a parameter is supplied and when there is no parameter. If the page contents are almost identical, the next option may better suit your needs.

Listing 4-10. Page with two routes

```
@page "/advancedpage2/{param1}"
@page "/advancedpage2"

<p>This is the advanced page 2</p>
@if (param1 != null)
{
    <p>parameter: @param1</p>
}
```

```
@code {
    [Parameter]
    public string param1 { get; set; }
}
```

As you can see, in this page (Listing 4-10) we have two routes – two simple routes, one basic and the other with a parameter. In this case, we simply check if parameter was supplied and display it.

Listing 4-11. Navigation page

```
<div>
    <ul class="nav flex-column">
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="/page1">
                Basic page 1
            </NavLink>
        </li>
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="/page2">
                Basic page 2
            </NavLink>
        </li>
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="/advancedpage1">
                Advanced page 1
            </NavLink>
        </li>
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="/advancedpage1/
                testparam">
```

```

        Advanced page 1 with parameters
    </NavLink>
</li>
<li class="nav-item px-3">
    <NavLink class="nav-link" href="advancedpage2">
        Advanced page 2
    </NavLink>
</li>
<li class="nav-item px-3">
    <NavLink class="nav-link" href="advancedpage2/
    testparam">
        Advanced page 2 with parameters
    </NavLink>
</li>
</ul>
</div>

```

For the navigation (Listing 4-11), we have it all set up in the NavMenu component. You can also notice how some of these routes take parameters in them for our examples.

Components

Components are just like pages; in fact, they are pages or pages are components. Any .razor file can be used as a component, but if it has a route declaration, it can be navigated to as page.

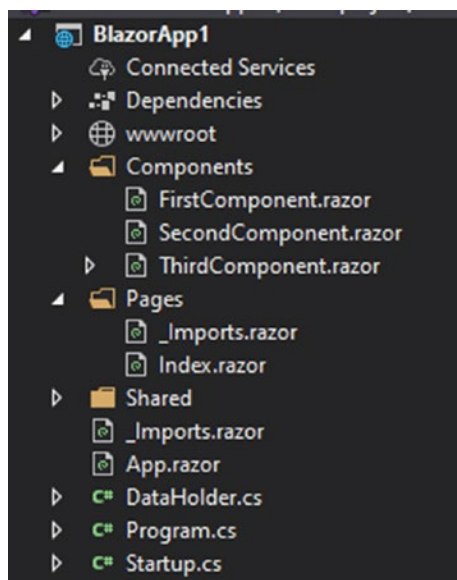


Figure 4-4. Blazor client-side project with example component files

To understand components better, we have a little project (Figure 4-4), with three different components. The first two will explore sharing of variables between parent and child and the issues that come with that. The third component will be covered in the next part.

Listing 4-12. Simple parameter pass-through

```
<input @onchange="@TestChanged" />

@code {
    [Parameter]
    public Action<UIChangeEventArgs> TestChanged { get; set; }
}
```

The first option (Listing 4-12) is to create a rather complex system of events and event argument checks. However, at this point this is the most efficient and the simplest to use option. In the component we first have a

custom event, which basically just binds to the generic onchange event in the input element. What we are really looking for is the new value, and that is what we will retrieve from parent.

Listing 4-13. Index.razor contents

```
@page "/"

<p><BlazorApp1.Components.FirstComponent TestChanged="@(( args)
=> settest(args))"></BlazorApp1.Components.FirstComponent></p>
<p>@test</p>

@code {
    string test;

    void settest(UIChangeEventArgs args)
    {
        test = (string)args.Value;
    }
}
```

In the Index.razor (Listing 4-13), we declare the component and access the TestChanged event as we would do any other generic event on generic variables. The method executed on TestChanged event simply takes the change arguments and assigns the new value to an already declared test variable.

Listing 4-14. Binds with static variables

```
<input @bind="@DataHolder.testvariable" />
<p>@DataHolder.testvariable</p>
```

What might also be a good idea is to simply have a static variable (Listing 4-14) in some other class, but it does not actually work. In this case, we have our DataHolder.cs class, with a static string testvariable. If you look at the code in the component, that would work perfectly, but it

would only work in this component. As you will see, the parent can also bind to the variable, but only separately.

Listing 4-15. Index page

```
<p><BlazorApp1.Components.SecondComponent></BlazorApp1.
Components.SecondComponent></p>

<p>General variable</p>
<p>@DataHolder.testvariable</p>
<p><input @bind="@DataHolder.testvariable" /></p>
```

In the Index.razor (Listing 4-15), we add our component, and then we display the variable and also bind it with an input field. Unfortunately, the updates only occur on the page where change happened. So, if you change it in this page, it will display here, but it will not do the same in the component or vice versa.

Using Key to Preserve Components

Another important feature of components is the preservation. This works great when your interface needs to add or remove an item. When you loop through elements and display them, every time the loop occurs, your elements will be destroyed and re-created. To preserve them, you need to use something called key, and the best way to understand it is to see it.

Listing 4-16. Component with key

```
<p>@keyforcomponent</p>
@code {
    [Parameter]
    public Guid keyforcomponent { get; set; }
}
```

```

<p><button @onclick="@AddElement">Add</button></p>
@foreach (var item in ElementList)
{
    <p><button @onclick="@(() => RemoveElement(item))">Remove
    </button></p>
    <p><BlazorApp1.Components.ThirdComponent @key="item"
keyforcomponent="item"></BlazorApp1.Components.
ThirdComponent></p>
}
@code {
    List<Guid> ElementList = new List<Guid>();

    void AddElement()
    {
        ElementList.Add(Guid.NewGuid());
    }

    void RemoveElement(Guid id)
    {
        ElementList.Remove(id);
    }
}

```

Our component (Listing 4-16) in this case simply takes the key or the id as we call it and displays it. Do note that we only display the id for showcasing purposes and it is not mandatory to do so. In our `Index.razor`, we have a list variable established which will contain Guid types. We also have `AddElement` and `RemoveElement` methods. They will both trigger the foreach loop, which simply loops through the list and creates our component for each cycle. The interesting part here is the passing of `@key` property, which by default you can find in every component. This unique

key is what will preserve that element not to be re-evaluated. This is very useful if you want to display list of products or list of items for some form where you might need to add or remove items, but leave the current ones with the values inserted or displayed.

Example

Now we will take a look at a simple example that will explore navigation parameters and component use, as well as the general Blazor interaction. We only have a simple sign-up form and a preview page.

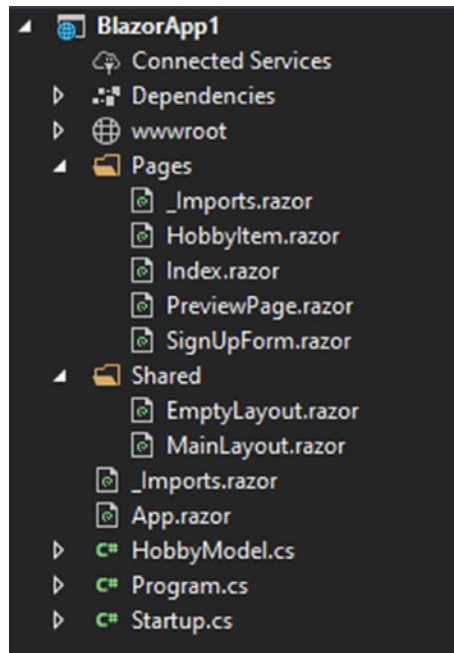


Figure 4-5. Example project structure

As you may notice, we will be using two types of layouts (Figure 4-5) in this project, and you will see exactly how to accomplish that.

Listing 4-17. Component with parameters and callbacks

```

<div style="width:300px;">
  <div style="width:50px;">
    @title
  </div>
  <div style="width:250px;">
    @description
  </div>
  <div style="width:250px;">
    <button @onclick="@delete">Delete</button>
    <p>@tst</p>
  </div>
</div>

@code {
  [Parameter]
  public string id { get; set; }
  [Parameter]
  public string title { get; set; }
  [Parameter]
  public string description { get; set; }

  [Parameter]
  public EventCallback<string> OnDelete { get; set; }

  async void delete()
  {
    await OnDelete.InvokeAsync(id);
  }
}

```

Listing 4-18. Hobby model

```
namespace BlazorApp1
{
    public class HobbyModel
    {
        public string id { get; set; }
        public string title { get; set; }
        public string description { get; set; }
    }
}
```

First, we need to create a component (Listing 4-17) for a hobby, which will be used as a list, and the user will be able to add or remove them. Of course, you do not have to use a component, but it is more convenient to do that in the long run. For the hobby, we also have a model (Listing 4-18), which we will only use in the parent page for the component. The hobby is created in the parent, added to the list, and then displayed; therefore, we do not have any input fields – we simply display what was inserted and allow for deletion, which is something we will explore later.

Listing 4-19. Page with navigation manager

```
@page "/signuppage"
@inject NavigationManager navmanager
@using Newtonsoft.Json

<p>First name</p>
<p><input @bind="@firstname"></p>

<p>Second name</p>
<p><input @bind="@surname"></p>

<p>Bio</p>
<p><textarea @bind="bio"></p>
```

```

<p>Hobbies</p>
<p>Title</p>
<p><input @bind="@newhobby.title"></p>
<p>Description</p>
<p><textarea @bind="@newhobby.description" /></p>
<p><button @onclick="@AddHobby" >Add hobby</button></p>
<p>My hobbies</p>
@foreach (var item in hobbies)
{
    <HobbyItem @key="@item.id" id="@item.id" title="
@item.title" OnDelete="DeleteHobby" description="@item.
description"></HobbyItem>
}
<p><button @onclick="@Submit">Submit</button></p>
@code {
    string firstname { get; set; }
    string surname { get; set; }
    string bio { get; set; }

    HobbyModel newhobby = new HobbyModel() { id = Guid.
NewGuid().ToString() };
    List<HobbyModel> hobbies { get; set; } = new
List<HobbyModel>();

    void AddHobby()
    {
        hobbies.Add(newhobby);
        newhobby = new HobbyModel() { id = Guid.NewGuid().
ToString() };
    }
}

```

```

void DeleteHobby(string id)
{
    hobbies.Remove(hobbies.Where(x => x.id == id).
        ToArray()[0]);
}

void Submit()
{
    var json = JsonConvert.SerializeObject(hobbies);
    navmanager.NavigateTo("/previewpage/" + firstname + "/"
        + surname + "/" + bio + "/" + json);
}
}

```

We start the Sign up page (Listing 4-19) by declaring a route to it, as well as adding some dependencies that will be used and explored a bit later. The user has three fields to be field: first name, second name, and bio. As you can see, the variables for those are declared in the code section, and they are bound to input elements and textarea for the bio. This is the most straightforward and the most practical approach to take. The more interesting part is the newhobby variables, which have their properties bound to the appropriate input fields. You may also notice how the list of hobbies is not only declared but assigned too. This way, we do not have to check for null values. A hobby is added in the AddHobby method, which simply adds the current hobby object to the list and creates a new one.

So far, everything seems quite simple, but the difficult part is yet to come. Hobby components are displayed in every cycle of the loop, where we use keys to preserve existing elements although in this case that is not required. More importantly, we pass the id and keep it in the component. The component (Listing 4-17) has an event callback declared with a return type of string. The delete button in the component invokes the callback by passing the id of that component. This is how you create a completely

custom event for your component. The parent page (Listing 4-19) has a method `DeleteHobby` which is executed when the callback is invoked, or in other words when the event occurs. Then, the `DeleteHobby` method takes the string argument (id for the hobby) and with some Linq magic removes the element with that id from the list.

Finally, we want to get to the preview and that is where things get complex. If you are passing parameters to a component, everything is quite simple; unfortunately, it is not the same if you do it for a page.

Listing 4-20. Page with parameters

```
@page "/previewpage/{firstname}/{surname}/{bio}/{hobbies}"
@using Newtonsoft.Json
@layout EmptyLayout
<p>First name: @firstname</p>
<p>Second name: @surname</p>
<p>bio: @bio</p>

<p>Hobbies:</p>
@if (hobbieslist != null)
{
    @foreach (var item in hobbieslist)
    {
        <p>id: @item.id</p>
        <p>title: @item.title</p>
        <p>description: @item.description</p>
    }
}

<p><button>Submit</button></p>

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



```

[Parameter]
public string surname { get; set; }
[Parameter]
public string bio { get; set; }

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

List<HobbyItem> hobbieslist;

protected override void OnParametersSet()
{
    hobbieslist = ((Newtonsoft.Json.Linq.JArray)
        JsonConvert.DeserializeObject(hobbies)).
        ToObject<List<HobbyItem>>();
}
}

```

Our preview page (Listing 4-20) contains four parameters, and you may already notice where the simplicity disappears. In the query string, we can only pass string content; therefore, more complex object needs to be serialized. This is done in our sign up page (Listing 4-19), where submit method first serializes the list to JSON string and only then adds it to the query string for navigation. We retrieve it as a string parameter (Listing 4-20) hobby and only then deserialize it to our object in a very complex way. A good way to get rid of that complexity would be to simply have a static variable somewhere in the code and read it once the preview page is initialized.

As you have seen in this example, using components is always a good decision, even when we need to deal with callbacks to bind data two ways. The parameters are useful and easy to pass, but do not forget that only works for component. Once you get to the pages, you have to be very careful with parameters. They should only be used when data needs to be passed from outside sources.

Summary

As you may have noticed, you will mostly use client-side Blazor for your front end alongside a back end. There is, however, a way to avoid having two different projects; that way is Blazor hosted and that is what we will cover in the next chapter.

CHAPTER 5

Blazor hosted

The previous chapters covered a type of Blazor that runs on the server-side and the type that runs in the browser. This chapter will cover the hosted type of Blazor, which essentially is client-side Blazor combined with web api project.

Default template overview

Like the other types of Blazor, this one too has its own template. It is however as lot more complicated both in its size and its structure.

General structure

The structure of Blazor hosted project is rather complex; fortunately for us, everything is taken care of in the template. Essentially, you have a shared assembly for API project, client-side project, and a shared .NET standard library to hold shared data models. Also important to note, the client-side project is served from the API project, not vice versa.

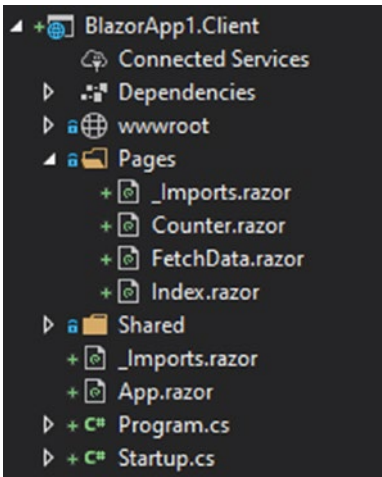


Figure 5-1. Client-side Blazor project in the solution

Figure 5-1 shows the first part of the solution, in this case called client, which is basically client-side Blazor template.

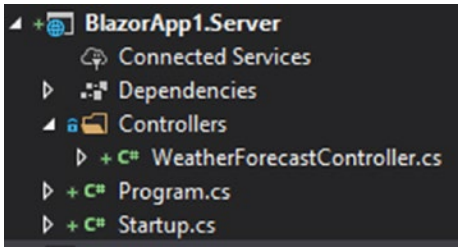


Figure 5-2. Server-side Blazor project in the solution

Figure 5-2 shows the web api project in the solution. It will be used as the back-end part of the client-side project.

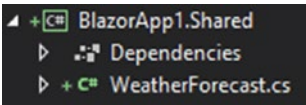


Figure 5-3. Shared library .net standard project in the solution

As you can see (Figure 5-3), the hosted project contains the same template for client part as the client-side template. The server part has a simple controller for the default template example. Finally, the Shared library has one sample model. Do note the library always has to have a class to work at all.

Listing 5-1. Startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.ResponseCompression;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Newtonsoft.Json.Serialization;
using System.Linq;

namespace BlazorApp1.Server
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection
            services)
        {
            services.AddMvc().AddNewtonsoftJson();
            services.AddResponseCompression(opts =>
            {
                opts.MimeTypes = ResponseCompressionDefaults.
                    MimeTypes.Concat(
                        new[] { "application/octet-stream" });
            });
        }
    }
}
```

```
public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env)
{
    app.UseResponseCompression();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBlazorDebugging();
    }

    app.UseStaticFiles();
    app.UseClientSideBlazorFiles<Client.Startup>();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
        endpoints.MapFallbackToClientSideBlazor<Client.
            Startup>("index.html");
    });
}
}
```

Listing 5-1 shows the Startup.cs file in the default template project; it is shown as it is generated.

Listing 5-2. Startup.cs for client

```

using Microsoft.AspNetCore.Components.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace BlazorApp1.Client
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
        }

        public void Configure(IComponentsApplicationBuilder app)
        {
            app.AddComponent<App>("app");
        }
    }
}

```

As mentioned previously, client-side comes from the server-side. So, we need to start with the server-side Startup.cs. First, in the ConfigureServices (see Listing 5-2), you will notice a lot of API-related things, and that is because essentially it is an API. The Blazor part starts in the Configure method, where you will find UseClientSideBlazorFiles; this is what launches the client-side startup and we will get back to that later. Another important part of this is the endpoints – MapDefaultControllerRoute basically takes care of the API routes and then MapFallbackToClientSideBlazor will deal with the page routes. We also have Blazor debugging added with UseBlazorDebugging, which is not mandatory. It would be best practice not to change anything in here, but if you do change Startup.cs to another name, make sure you change it in both places. The hosted client-side part, as you can see, is exactly the same as you would find in the client-side template.

Clean up the template

The templates in Visual Studio are very useful and will save you lots of time. Despite all that, templates contain examples and other contents that you do not need. So now you will see how to safely clean it up.

To understand this better, we have two projects: one with default contents and the other containing only what you will need.

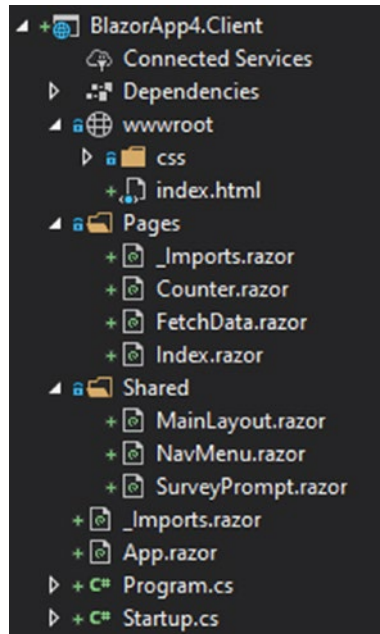


Figure 5-4. Client-side Blazor project in the solution

Figure 5-5 shows server-side project.

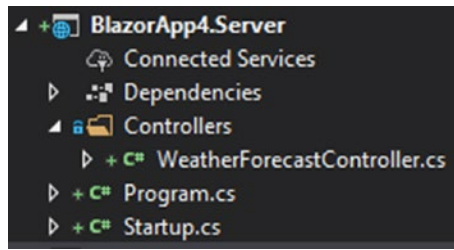


Figure 5-5. *Server-side Blazor project in the solution*

Figure 5-6 shows shared library .net standard project.

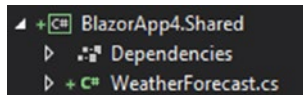


Figure 5-6. *Shared library .net standard project in the solution*

Figure 5-7 shows cleaned template.

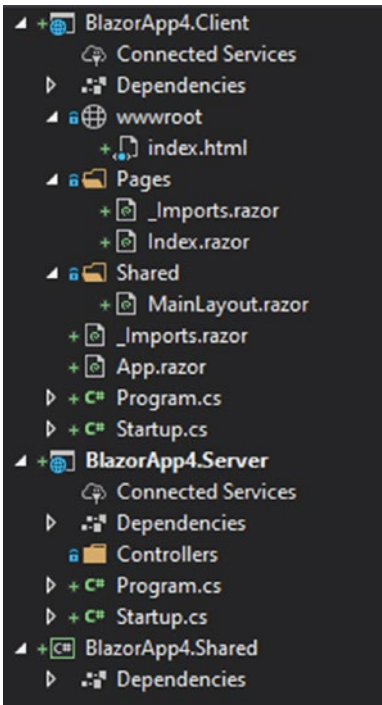


Figure 5-7. *Cleaned template*

As you may have already noticed, initial template (Figures 5-4, 5-5, and 5-6) is filled with lots of unnecessary files and code. As far as files are concerned, simply look at the cleaned template (Figure 5-7) and delete all the unnecessary ones to get to that result.

After you are done with cleaning the files, the code, for the most part, is up to you to decide. Index.razor does need to be cleaned as it contains Survey prompt component (SurveyPromt.razor) which at that point will no longer exist. Other than that, you should completely clean the MainLayout.razor, as you will more than likely need to customize that, and it will be a lot quicker to do it from scratch.

Navigation

Navigation in hosted Blazor works the same way as it does for server-side or client-side, after all – host is part client-side Blazor, part api project. However, there is a big difference, or rather something to watch for. Both api and pages parts will run under the same domain; therefore, the routes are shared and conflicts may arise. Obviously, to avoid that, you simply do not set the same route for two things. But that may be easier said than done, so let us look at some possible structures.

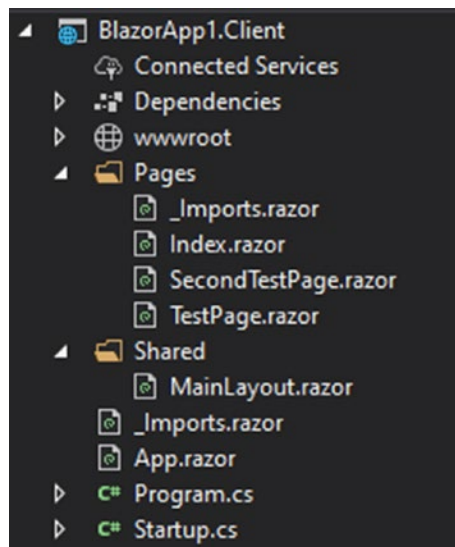


Figure 5-8. *Client part of the project*

Figure 5-8 shows the client-side Blazor part, with two test pages. Also, the template has been cleaned and prepared for work.

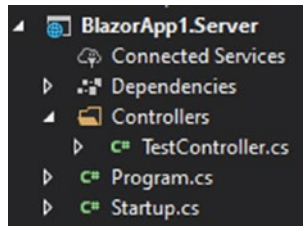


Figure 5-9. API part of the project

For this example, we have cleaned up the template (see Figure 5-9) set up two pages in the client part. For the server part, we simply have one controller.

Listing 5-3. Page with conflicted route

```
@page "/conflictedroute"
```

```
<p>test page</p>
```

Listing 5-4. Test controller

```
using Microsoft.AspNetCore.Mvc;

namespace BlazorApp1.Server.Controllers
{
    public class TestController : Controller
    {
        [Route("conflictedroute")]
        public string conflictedroutemethod()
        {
            return "test";
        }
    }
}
```

First, you can see the conflict between the route of `TestPage.razor` and the route of the `Test` controller (see Listings 5-3, 5-4). This will not cause an error, but instead the controller route will take priority.

Listing 5-5. Second test page

```
@page "/secondtestpage"

<p>Second test page</p>
```

The best way to avoid this is to create a naming system. A truly logical way to name pages is by adding the word “page” (see Listing 5-5) on the end of route.

API calls

While in JavaScript you only had one way to make a request and of course hundreds of third-party implementations, in c# there are at least three major ways to do it. Before we begin, we will prepare a few routes on the back end for testing those ways.

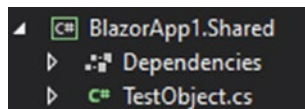


Figure 5-10. Shared library for the example project

Listing 5-6. Test object model

```
namespace BlazorApp1.Shared
{
    public class TestObject
    {
```

```

        public int a { get; set; }
        public string b { get; set; }
        public double c { get; set; }
    }
}

```

First, we create a shared model for data (see Figure 5-10 and Listing 5-6) that we will post and retrieve; you will see how that works client-side. As you can see in the code, we only have three properties with three different basic data types.

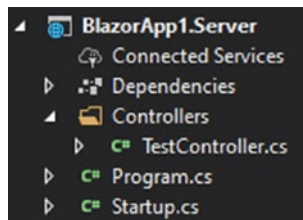


Figure 5-11. Web api part of the solution

Listing 5-7. Test controller

```

using Microsoft.AspNetCore.Mvc;

namespace BlazorApp1.Server.Controllers
{
    public class TestController : Controller
    {
        [Route("api/test1")]
        public string Test1()
        {
            return "response 1";
        }
    }
}

```

```

[Route("api/test2")]
public Shared.TestObject Test2()
{
    return new Shared.TestObject() { a = 5, b = "test
    string", c = 0.5 };
}

[Route("api/test3")]
public Shared.TestObject Test3([FromBody]Shared.
TestObject tobj)
{
    tobj.a += 15;
    tobj.b += " works and uses header: " + Request.
    Headers["headervalue"];
    tobj.c = 15;
    return tobj;
}

[Route("api/test4")]
public Shared.TestObject Test4([FromForm]Shared.
TestObject tobj)
{
    tobj.a = 15;
    tobj.b += " works and uses header: " + Request.
    Headers["headervalue"];
    tobj.c = 15;
    return tobj;
}
}
}

```

In the controller (see Listing 5-7), we have four basic routes for different kinds of calls in the client-side. The first one simply returns a string; we will need this because you retrieve a string a different way than the other object. On the second one, we will have JSON object returned to us on response. Finally, the third and fourth ones are the same, and they will read our object on request and return a modified version of it on response. The reason we need two routes for the same thing is because we want to test both Application/JSON and form-data ways of posting.

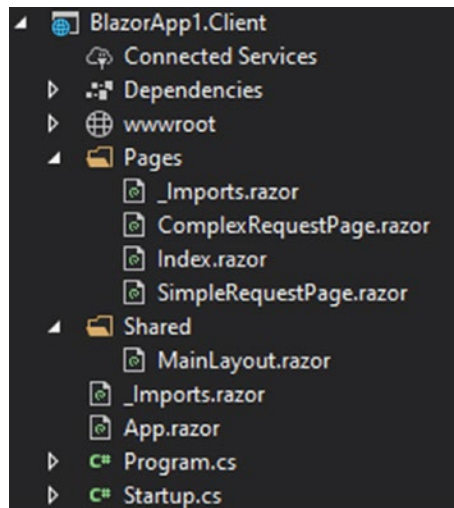


Figure 5-12. Project with http request pages

Our client-side part contains a couple of pages (see Figure 5-12) – `SimpleRequestPage.razor` and `ComplexRequestPage.razor`. In one of them, we will try the simplest and most efficient way to make API calls on Blazor and the other one will explore a more sophisticated option.

Listing 5-8. Main page with navigation links

```
@page "/"

<p> <NavLink href="complexrequestpage">complex request
page</NavLink></p>
<p><NavLink href="simplerequestpage">simple request
page</NavLink></p>
```

In the `Index.razor` (see Listing 5-8), we simply have two navigation elements to reach our pages.

JSONfull way

This option allows you to send your object directly, as well as retrieve them to your object. There is no additional conversion required, but there are some restrictions and limitations with this method.

Listing 5-9. Simple request page

```
@page "/simplerequestpage"
@inject HttpClient http

<p><button @onclick="@{async () => await GetTest()}">get
test</button></p>
<p><button @onclick="@{async () => await GetTest1()}">get
test with object</button></p>
<p><button @onclick="@{async () => await GetTest2()}">post
test</button></p>
<p>@testouput</p>
<p>@tobj.a</p>
<p>@tobj.b</p>
<p>@tobj.c</p>
```

```

@code {
    string testoutput;
    BlazorApp1.Shared.TestObject tobj = new BlazorApp1.Shared.
    TestObject();

    protected override Task OnInitializedAsync()
    {
        http.DefaultRequestHeaders.Add("headervalue", "tst");
        return base.OnInitializedAsync();
    }

    async Task GetTest()
    {
        try
        {
            testoutput = await http.GetStringAsync("api/test1");
        }
        catch (Exception e)
        {
            testoutput = e.Message;
        }
    }

    async Task GetTest1()
    {
        tobj = await http.GetJsonAsync<BlazorApp1.Shared.
        TestObject>("api/test2");
    }
}

```

```

async Task GetTest2()
{
    tobj = await http.PostJsonAsync<BlazorApp1.Shared.
    TestObject>("api/test3", tobj);
}
}

```

Listing 5-10. Api call

```

try
{
    testouput = await http.GetJsonAsync("api/test1");
}
catch (Exception e)
{
    testouput = e.Message;
}

```

As you can see (Listing 5-9), we have three methods for our three routes in the server part. First, of course, we declare the route for the page, and after that, we need to inject our `HttpClient`; this is what will do all that there is to do related to APIs. First, let us take a look at the `GetTest` method, which may seem like the simplest one, but you do have to be careful with it. When you have a string, you need to use `GetStringAsync` to retrieve it. While it may look possible to do `GetJsonAsync`, as it is shown in other methods, it is actually not. You can try it by changing the code in the `GetTest` method with the code provided (Listing 5-10). Also, it is always a good idea to handle errors in `try/catch` statement, as there is no other way to check for failures.

The next option for us is the `GetJsonAsync`, and this is where the `Shared` models come into play. As you can see in the code, we have a variable named `tobj` – that is our `TestObject` model. The variables in the object are displayed in HTML paragraphs for us to see the result.

In the `GetTest1`, we simply assign the return of `GetJsonAsync` method, which simply requires us to supply the return type and the route as a parameter. Later, we will explore the downfalls of it, but you will also see how efficient in terms of coding this can be. Finally, in the `GetTest2`, we have a post request, which gives us a return of the same `TestObject` type. Using `PostJsonAsync` saves even more time, as you do not need to convert anything, establish many variables, and do other tedious tasks.

Probably the biggest downfall of this is the fact that you cannot set headers for each request. So, you would either have to construct separate http clients and add different default headers or simply send all the headers for all the request. In our example, we do have a default header, which comes back with our Json-based requests. You can find this in the `OnInitializedAsync` override. Furthermore, it may be a good idea to construct one global http client or several with different header sets. If you decide to do this in a .NET standard class library, you will need to declare using statement with namespace – `Microsoft.AspNetCore.Components`.

HTTP client manipulations

Now that we know the simple and efficient way, we will take a look at another option. This is useful if you need to check statuses on response, use body of types other than `Application/JSON`, or if you need to deal with cookies and custom headers for each request.

Listing 5-11. Complex request page

```
@page "/complexrequestpage"
@inject HttpClient http

<p><button @onclick="@{async () => await GetTest1()}">get test
with object</button></p>
<p><button @onclick="@{async () => await GetTest2()}">post
test</button></p>
```

```

<p>@tobj.a</p>
<p>@tobj.b</p>
<p>@tobj.c</p>

@code {

    BlazorApp1.Shared.TestObject tobj = new BlazorApp1.Shared.
    TestObject();

    async Task GetTest1()
    {
        tobj = await http.GetJsonAsync<BlazorApp1.Shared.
        TestObject>("api/test2");
    }

    async Task GetTest2()
    {
        var msg = new HttpRequestMessage(HttpMethod.Post,
        "api/test4");
        msg.Headers.Add("headervalue", "tst");
        var formdt = new MultipartFormDataContent();
        formdt.Add(new StringContent(tobj.a.ToString()), "a");
        formdt.Add(new StringContent(tobj.b), "b");
        formdt.Add(new StringContent(tobj.c.ToString()), "c");
        msg.Content = formdt;

        var resp = await http.SendAsync(msg);
        if (resp.IsSuccessStatusCode)
        {
            var result = System.Json.JsonObject.Parse(await resp.
            Content.ReadAsStringAsync());
            tobj.a = result["a"];
            tobj.b = result["b"];
        }
    }
}

```

```

        tobj.c = result["c"];
    }

}
}

```

Our code (see Listing 5-11) in this case provides two methods; the first one, `GetTest1`, simply retrieves an object for us to test with using the simple option of doing it. The next one, `GetTest2`, is where everything gets interesting. In this case, we will be using `SendAsync` method in the `HttpClient` which requires us to supply a request message. Initially, you construct the request message by providing the http method and the route for the message. If you need to call on third-party API, simply use the full url. After the initial construction, we can add a header for this specific request. The interesting part here is the content; you can choose between `StreamContent`, `StringContent`, `ByteArrayContent`, `FormUrlEncodedContent`, and `MultipartFormDataContent`. In this case, we will be using form-data content, which can contain key/value pairs of any other content types. This is particularly useful if you need to upload files, where you would use either stream or byte array. The content then is simply assigned to the `Content` property in the message. After we have all that, we can send the request and get a response message. In the response message, you get lots of things – status, content, headers, and more. In this case, we test out a built-in property, `IsSuccessStatusCode`, which checks for response status being 200. If that succeeds, we then get to the content and read it as string. Afterward, we need to turn it into some object, and as you can already see with the `GetJsonAsync` or `PostJsonAsync`, this would be a lot easier. In the end, if you want something quick and simple, you would use the JSON-based way, and if you want something more thorough, you would take the `SendAsync` route.

Example

This example will cover pretty much all that you have learned previously, including, of course, API calls. You may call it a store management system, where we will be able to create a user, create a product, remove a product, and then create a purchase from the list of buyers and products.

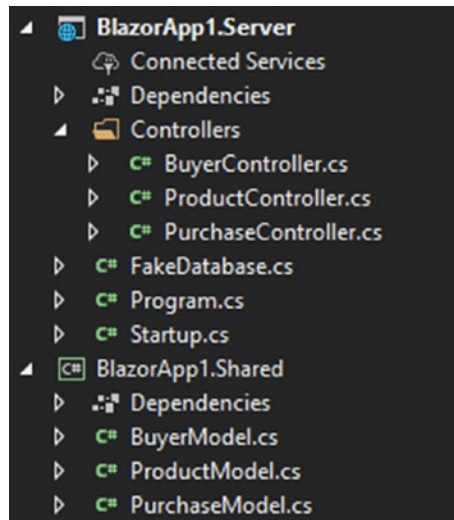


Figure 5-13. Web api and shared library of the example project

As usual with Blazor hosted projects, we first want to set up our back-end part (see Figure 5-13). Alongside that, we will establish the shared models as they are used in the controllers.

Listing 5-12. Buyer model

```
namespace BlazorApp1.Shared
{
    public class BuyerModel
    {
        public string id { get; set; }
    }
}
```

```

        public string name { get; set; }

        public decimal totalspent { get; set; }
    }
}

```

Listing 5-13. Product model

```

namespace BlazorApp1.Shared
{
    public class ProductModel
    {
        public string id { get; set; }

        public string name { get; set; }
        public string description { get; set; }

        public double value { get; set; }
        public bool available { get; set; }
    }
}

```

Listing 5-14. Purchase model

```

using System;
using System.Collections.Generic;

namespace BlazorApp1.Shared
{
    public class PurchaseModel
    {
        public string id { get; set; }

        public DateTime datecreated { get; set; }
    }
}

```



```

        public List<string> products { get; set; }
        public string buyer { get; set; }
    }
}

```

Before anything else, we need to establish our models which will be shared between client and server parts. First, we have our buyer model (Listing 5-12) which holds id, name, and total spent for the buyer. Second model is for the product (Listing 5-13), which is quite basic as well; the important part here is the available variable, since we want to be able to delete the product, but statistics will still depend on that. The more complicated one is the Purchase (Listing 5-14) model; here we have a list of products that were bought on that purchase, but they are stored as strings rather than full objects. You should always store it like that, especially in the database; otherwise, it can become very inefficient.

Listing 5-15. Fake database

```

using System.Collections.Generic;

namespace BlazorApp1.Server
{
    public class FakeDatabase
    {
        public static List<Shared.BuyerModel> buyers = new
            List<Shared.BuyerModel>();
        public static List<Shared.ProductModel> products = new
            List<Shared.ProductModel>();
        public static List<Shared.PurchaseModel> purchases =
            new List<Shared.PurchaseModel>();
    }
}

```

Since we are learning Blazor, not database interactions, we have a simple mock database (Listing 5-15) setup which is basically just three static list variables, containing three different models.

Listing 5-16. Buyer controller

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace BlazorApp1.Server.Controllers
{
    public class BuyerController : Controller
    {
        [Route("api/getbuyerlist")]
        public Task<List<string>> GetAllBuyers()
        {
            var templist = new List<string>();
            foreach (var item in FakeDatabase.buyers)
            {
                templist.Add(item.id);
            }
            return Task.FromResult(templist);
        }

        [Route("api/getbuyerdetails")]
        public Task<Shared.BuyerModel>
        GetDetailsForSingleBuyer(string id)
        {
            return Task.FromResult(FakeDatabase.buyers.Where
                (x => x.id == id).ToArray()[0]);
        }
    }
}
```

```

[Route("api/createbuyer")]
public Task<bool> CreateBuyer([FromBody]Shared.
BuyerModel buyer)
{
    try
    {
        FakeDatabase.buyers.Add(buyer);
        return Task.FromResult(true);
    }
    catch
    {
        return Task.FromResult(false);
    }
}
}
}

```

The first controller (Listing 5-16) is all about the buyer; we only have three routes here. The first one will get the list of buyers, and the second one will get details for items. The reason we separate those two is because most of the time you will want to display only a set of records. Of course, there are other ways to do it; you may also limit and skip directly in database, but this is one of the more efficient way to do it. Finally, we have a route that allows us to create a buyer. Notice how it takes a model from the request body and then simply inserts it into the database.

Listing 5-17. Product controller

```

using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

```

```

namespace BlazorApp1.Server.Controllers
{
    public class ProductController : Controller
    {
        [Route("api/getproductlist")]
        public Task<List<string>> GetAllproducts()
        {
            var templist = new List<string>();
            foreach (var item in FakeDatabase.products)
            {
                if (item.available)
                {
                    templist.Add(item.id);
                }
            }
            return Task.FromResult(templist);
        }

        [Route("api/getproductdetails")]
        public Task<Shared.ProductModel> GetDetailsForSingle
        Product(string id)
        {
            return Task.FromResult(FakeDatabase.products.
                Where(x => x.id == id).ToArray()[0]);
        }

        [Route("api/createproduct")]
        public Task<bool> CreateProduct([FromBody]Shared.
        ProductModel product)
        {

```

```

        try
        {
            FakeDatabase.products.Add(product);
            return Task.FromResult(true);
        }
        catch
        {
            return Task.FromResult(false);
        }
    }

    [Route("api/removeproduct")]
    public Task<bool> CreateProduct(string id)
    {
        try
        {
            FakeDatabase.products.Find(x => x.id == id).
                available = false;
            return Task.FromResult(true);
        }
        catch
        {
            return Task.FromResult(false);
        }
    }
}

```

The products controller (Listing 5-17) will be a bit different from the buyers one. We do have the same system for retrieving and inserting records, but with that, we allow for removing a product. As mentioned previously, we do not actually delete the product, we simply set it as

unavailable. This is always a good practice when you have statistics or other items that rely on some particular dataset like product in this case.

Listing 5-18. Purchase controller

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace BlazorApp1.Server.Controllers
{
    public class PurchaseController : Controller
    {
        [Route("api/getpurchaselist")]
        public Task<List<Shared.PurchaseModel>>
            GetAllPurchases()
        {
            return Task.FromResult(FakeDatabase.purchases);
        }

        [Route("api/createpurchase")]
        public Task<bool> CreatePurchase([FromBody]Shared.
            PurchaseModel purchase)
        {
            try
            {
                FakeDatabase.purchases.Add(purchase);
                return Task.FromResult(true);
            }
            catch
            {
                return Task.FromResult(false);
            }
        }
    }
}
```

```

    }
  }
}

```

Our final controller is for purchases, where we have something a bit more simplified. Instead of going through id retrieval and details retrieval separately, this time we just get the whole “database” at once. This approach would be fine if you only expect 20–30 records, but once you go into hundreds, it will not only be a waste of your resources, but it will slow down your user experience which may be even worse than wasting resources.

Listing 5-19. Buyer page

```

@page "/createbuyerpage"
@using datamodels = BlazorApp1.Shared;
@Inject HttpClient http
@Inject IJSRuntime js

<p>name</p>
<p><input @bind="@currentbuyer.name"></p>
<p><button @onclick="@{async () => await Create()}">
Create</button></p>

@code {
    datamodels.BuyerModel currentbuyer = new datamodels.
    BuyerModel() { id = Guid.NewGuid().ToString() } ;

    async Task Create()
    {
        if (await http.PostJsonAsync<bool>("/api/
createbuyer",currentbuyer))

```

```

    {
        currentbuyer = new datamodels.BuyerModel(){ id =
            Guid.NewGuid().ToString() };
    }
    else
    {
        await js.InvokeAsync<object>("alert", "Something
            went wrong");
    }
}
}

```

We will begin with the client part from the create buyer page. This page is the most basic one out of all of them. We have to understand the client part has a folder shared for layouts, but there is also a library named Shared, so to solve the conflict, the using statement is used to assign the namespace to a variable. In the code section, we simply declare a variable `currentbuyer` and bind its properties to appropriate input fields. With that, the insertion system becomes quite simple, we just use `PostJsonAsync` and check if it was inserted, and if it was, then we simply re-assign `currentbuyer` variable with an empty construct.

Listing 5-20. Product page

```

@page "/createproductpage"
@using datamodels = BlazorApp1.Shared;
@inject HttpClient http
@inject IJSRuntime js

<p>name</p>
<p><input @bind="currentproduct.name"></p>
<p>description</p>
<p><input @bind="currentproduct.description"></p>

```



```

<p>value</p>
<p><input @bind="currentproduct.value"></p>
<p><button @onclick="@(async () => await
SubmitProduct())">Submit</button></p>
<p>----</p>
<p>Products</p>

@if (products != null)
{
    @foreach (var item in products)
    {
        <BlazorApp1.Client.Components.ProductComponent
        @key="@item" id="@item" OnDelete="Remove">
        </BlazorApp1.Client.Components.ProductComponent>
    }
}

@code {
    datamodels.ProductModel currentproduct = new datamodels.
    ProductModel() { id = Guid.NewGuid().ToString(), available
    = true };

    List<string> products;

    protected override async Task OnInitializedAsync()
    {
        products = await http.GetJsonAsync<List<string>>("/api/
        getproductlist");
    }

    async Task SubmitProduct()
    {

```

```

    try
    {
        if (await http.PostJsonAsync<bool>("/api/
createproduct", currentproduct))
        {
            currentproduct = new datamodels.ProductModel()
            { id = Guid.NewGuid().ToString(), available =
            true };
        }
        else
        {
            await js.InvokeAsync<object>("alert",
            "Something went wrong");
        }
    }
    catch (Exception e)
    {
        await js.InvokeAsync<object>("alert", e.Message);
    }
}

async void Remove(string id)
{
    if (await http.GetJsonAsync<bool>("/api/
removeproduct?id=" + id))
    {
        products.Remove(id);
    }
}
}

```

Listing 5-20 shows the contents of `CreateProductPage.razor` file, where we have the interface for creating a new product.

Listing 5-21. Product component

```
@using datamodels = BlazorApp1.Shared;
@inject HttpClient http
@if (product != null)
{
    <p>id: @product.id</p>
    <p>title: @product.name</p>
    <p>description: @product.description</p>
    <p>value: @product.value</p>
    <p><button @onclick="@delete">Delete</button></p>
}

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

    [Parameter]
    public EventCallback<string> OnDelete { get; set; }

    datamodels.ProductModel product;

    protected override async Task OnParametersSetAsync()
    {
        product = await http.GetJsonAsync<datamodels.
            ProductModel>("/api/getproductdetails?id=" + id);
    }

    async void delete()
    {
        await OnDelete.InvokeAsync(id);
    }
}
```

Our product create page is where things become a lot more interesting and at the same time a lot more complex. Let us start with the creation part, which is very similar to what we have in product create page. It is similar for a reason; this is a very efficient way to do it, and if you have similar tasks like that, you should keep them almost identical so that you could simply copy and paste it and change the names of variables accordingly. In the page (Listing 5-20), we simply create a variable for our product model and bind its properties to the inputs. After that, we simply use `PostJsonAsync` which inserts our new product.

The more interesting part of this arrangement is the list output of the products. We display each product as a component generated in a for loop. The page (Listing 5-20) retrieves our product list on being initialized and sets it to a list of strings variable. Then, foreach loop creates a component using each one of the id values and passes them to the component. The component (Listing 5-21) takes that id and retrieves the details for product; in this case, it does that whenever a parameter is set. The removal of the product is a more complex feature, as we want to update the list after it was removed. Therefore, in the component (Listing 5-21) we have an event callback declared, which will have an argument holding the id. The event is set in the page (Listing 5-20) where we simply take the id argument and remove the appropriate product.

Listing 5-22. Purchase page

```
@page "/createpurchasepage"
@using datamodels = BlazorApp1.Shared;
@inject HttpClient http

<div style="float:left;width:33%;">
    @if (buyers.Count != 0)
    {
```

```

@foreach (var item in buyers)
{
    <BlazorApp1.Client.Components.BuyerForPurchase
    PageComponent id="@item" OnSelected="Buyer
    Checkselected"></BlazorApp1.Client.Components.
    BuyerForPurchasePageComponent>

}
}
else
{
    <p>No buyers found</p>
}

<p><button @onclick="@(async () => await
RefreshBuyers())">Refresh buyers</button></p>
</div>
<div style="float:left;width:33%;">
    @if (products.Count != 0)
    {
        @foreach (var item in products)
        {
            <BlazorApp1.Client.Components.
            ProductForPurchasePageComponent id="@item"
            OnSelected="ProductCheckselected">
            </BlazorApp1.Client.Components.ProductFor
            PurchasePageComponent>

        }
    }
}

```

```

        else
        {
            <p>No products found</p>
        }

        <p><button @onclick="@(async () => await
            RefreshProducts())">Refresh products</button></p>
    </div>
<div style="float:left;width:33%;">
    <button @onclick="SubmitPurchase">Create purchase</button>
</div>
@code {
    List<string> buyers = new List<string>();
    List<string> products = new List<string>();

    string buyer_selected;

    List<string> products_selected = new List<string>();
    void BuyerCheckselected(KeyValuePair<string,bool> arg)
    {
        buyer_selected = arg.Key;
    }

    void ProductCheckselected(KeyValuePair<string,bool> arg)
    {
        if (arg.Value)
        {
            products_selected.Add(arg.Key);
        }
    }
}

```

```

        else
        {
            products_selected.Remove(arg.Key);
        }
    }

    async Task RefreshBuyers()
    {
        buyers = await http.GetJsonAsync<List<string>>("api/
        getbuyerlist");
    }

    async Task RefreshProducts()
    {
        products = await http.GetJsonAsync<List<string>>("api/
        getproductlist");
    }

    async Task SubmitPurchase()
    {
        var tempobj = new datamodels.PurchaseModel() { id = Guid.
        NewGuid().ToString(), datecreated = DateTime.UtcNow,
            buyer = buyer_selected, products = products_selected
        };

        ;

        try
        {
            if (await http.PostJsonAsync<bool>("/api/
            createpurchase", tempobj))
            {
            }
        }
    }

```

```

        else
        {

        }

    }
    catch (Exception e)
    {

    }

}
}

```

Listing 5-22 shows the contents of `CreatePurchasePage.razor`, which has all the user interface for creating a purchase.

Listing 5-23. (`BuyerForPurchaseComponent.razor`)

```

@using datamodels = BlazorApp1.Shared;
@Inject HttpClient http

@if (buyer != null)
{
    <div style="float:left;width:100%;background-color:
    @selectioncolor">
        <div style="float:left;width:25%;">
            <p>@buyer.id</p>
        </div>
        <div style="float:left;width:25%;">
            <p>@buyer.name</p>
        </div>
        <div style="float:left;width:25%;">
            <p>@buyer.totalspent</p>
        </div>
    </div>

```



```

        <div style="float:left;width:25%;">
            <p><button @onclick="@(async () => await
                ToggleSelection())">select</button></p>

        </div>
    </div>
}

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

    [Parameter]
    public EventCallback<KeyValuePair<string,bool>> OnSelected
    { get; set; }

    string selectioncolor = "#ffd800";

    public bool is_selected { get; set; }

    datamodels.BuyerModel buyer;

    protected override async Task OnParametersSetAsync()
    {
        buyer = await http.GetJsonAsync<datamodels.BuyerModel>
            ("/api/getbuyerdetails?id=" + id.ToString());
    }

    async Task ToggleSelection()
    {
        is_selected = is_selected ? false : true;
        selectioncolor = is_selected ? "#4cff00" : "#ffd800";
        await OnSelected.InvokeAsync(new KeyValuePair<string,
            bool>(id,is_selected));
    }
}

```

The “buyer for purchase” component (Listing 5-23) will be used to display available buyers in the purchase page.

Listing 5-24. (ProductForPurchasePageComponent.razor)

```
@using datamodels = BlazorApp1.Shared;
@inject HttpClient http

@if (product != null)
{
    <div style="float:left;width:100%;background-color:
    @selectioncolor">
        <div style="float:left;width:20%;">
            <p>@product.id</p>
        </div>
        <div style="float:left;width:20%;">
            <p>@product.name</p>
        </div>
        <div style="float:left;width:20%;">
            <p>@product.description</p>
        </div>
        <div style="float:left;width:20%;">
            <p>@product.value</p>
        </div>
        <div style="float:left;width:20%;">
            <p><button @onclick="@ (async () => await
            ToggleSelection())">select</button></p>
        </div>
    </div>
}
```

```

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

    [Parameter]
    public EventCallback<KeyValuePair<string, bool>> OnSelected
    { get; set; }

    string selectioncolor = "#ffd800";

    public bool is_selected { get; set; }

    datamodels.ProductModel product;

    protected override async Task OnParametersSetAsync()
    {
        product = await http.GetJsonAsync<datamodels.
        ProductModel>("/api/getproductdetails?id=" +
        id.ToString());
    }

    async Task ToggleSelection()
    {
        is_selected = is_selected ? false : true;
        selectioncolor = is_selected ? "#4cff00" : "#ffd800";
        await OnSelected.InvokeAsync(new KeyValuePair<string,
        bool>(id, is_selected));
    }
}

```

Purchase page (Listing 5-22) may come across as simple, but it is rather complex. What helps us keep it clean are the components, in this case, one for buyer list items (Listing 5-23) and the other for product list items (Listing 5-24). First, we declare two list variables in the page, one for

buyers and the other for products; the lists are assigned either in method `RefreshBuyers` or `RefreshProducts` accordingly, and in this case, we have refresh buttons instead of simply fetching data on initialized. For both lists, we have `foreach` loops where we generate our components and pass the id into them; we will get back to the `Onselect` event later. Both product (Listing 5-24) and buyer (Listing 5-23) components retrieve their details according to the id passed and display them in HTML accordingly.

Another important feature of this arrangement is the selection of items. For that, we have `buyer_selected` and `products_selected` variables; as you can probably guess from this, we only allow one buyer to be selected. In the buyer component (Listing 5-23), we have several things related to selection; first we have `OnSelected` event callback, then is `_selected` Boolean, and finally `selectioncolor` string. The color is used for the background, and when selected, it changes; the Boolean is not necessary in this case, but it would be useful if you had to deal with changes more inside the component. The most important piece of this is the `OnSelected` callback, which is invoked in the `ToggleSelection`. What happens there is quite simple; we check if the product is selected or not on click and we switch that state. With that, we invoke the callback passing the id for product and the new state of selection. The product (Listing 5-24) component is a very similar case; in fact, the selection in component itself is identical. The difference is in the set callback method in the page (Listing 5-22), where we either remove item from the list or insert it. For the buyer, we simply replace the string if the buyer was selected, not deselected.

Finally, for the submission we have one single method `SubmitPurchase`, where we construct new purchase object and add the details to it alongside that. Afterward, we just need to use `PostJsonAsync` to add to our database. One thing to note here is the error handling. While `Json` get and post methods are very clean and simple, they lack the handling of errors. Fortunately for us, `c#` sharp error handling is quite detailed and efficient. But a good practice for all calls would be to use the `try catch` and then sort through exceptions as you would do with statuses.

Listing 5-25. Buyer component

```

@using datamodels = BlazorApp1.Shared;
@Inject HttpClient http

@if (purchases != null)
{
    @foreach (var item in purchases)
    {
        <BlazorApp1.Client.Components.PurchaseComponent
            id="@item.id"
            buyer="@item.buyer" datecreated="@item.
                datecreated">
        </BlazorApp1.Client.Components.PurchaseComponent>
    }
}

@code {
    List<datamodels.PurchaseModel> purchases;

    protected override async Task OnInitializedAsync()
    {
        purchases = await http.GetJsonAsync<List<datamodels.
            PurchaseModel>>("/api/getpurchaselist");
    }
}

```

Listing 5-26. Purchase component

```

@using datamodels = BlazorApp1.Shared;
@Inject HttpClient http
<div style="width:300px;">
    <p>id: @id</p>
    <p>created at: @datecreated</p>
    <p>products: </p>
    @{double total = 0;}
    @foreach (var item in products)
    {
        total += item.value;
        <p>id: @item.id</p>
        <p>name: @item.name</p>
        <p>description: @item.description</p>
        <p>value: @item.value</p>
    }
    <p>total: @total</p>
    <p>buyer: @buyer</p>
</div>

@code {
    [Parameter]
    public string id { get; set; }
    [Parameter]
    public DateTime datecreated { get; set; }
    [Parameter]
    public List<string> productsparam { get; set; }
    [Parameter]
    public string buyer { get; set; }

    List<datamodels.ProductModel> products { get; set; } = new
    List<datamodels.ProductModel>();

```

```

protected override async Task OnParametersSetAsync()
{
    foreach (var item in productsparam)
    {
        products.Add(await http.GetJsonAsync<datamodels.
            ProductModel>("/api/getproductdetails?id=" +
            id.ToString()));
    }
}
}

```

Our final feature simply displays all the purchases, and this once again requires a component. Our purchase component (Listing 5-26) will take all the purchase data through the parameters. Do note the parameters will work perfectly as long as you use them as a component. As we saw in the example of the previous chapter, page is opened using query string, and it would not be as simple to convert the list. What we do need to fetch is the list product details for each purchase, and that is done on parameters set. The lookup page (Listing 5-25) is even simpler; we only retrieve the list of purchases and generate components via the foreach loop.

Summary

At this point, you have seen what each type of Blazor has to offer; you should now be able to choose what you need for your project. Throughout these chapters, you have probably noticed that most things covered for one type will work well in the others too. In the next chapter, we will learn a bit more about Blazor and explore some additional features and use cases.

CHAPTER 6

General Blazor

In the last three chapters, we have covered each type more specifically. In this chapter, we will look at several things that can be very useful for any Blazor type.

Interacting with JavaScript

While Blazor can access many things directly, for some cases you will still need JavaScript. Fortunately for us, Blazor allows us to interact with JavaScript in a very simple way. This can be useful to access storage, deal with files, and access JavaScript libraries.

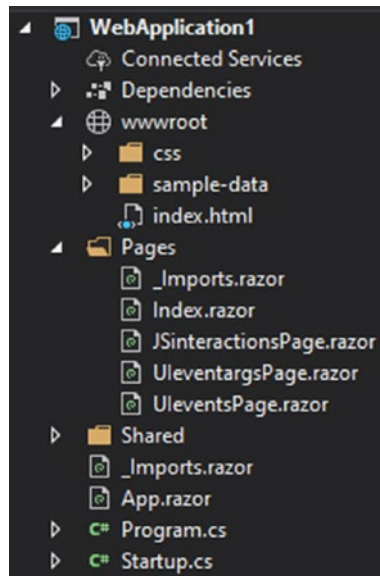


Figure 6-1. *The example project layout with pages added*

To understand the interactions better, we will use a single project (Figure 6-1) with three pages, and in those, we will explore JavaScript interactions, events, and their arguments. The project is a normal client-side project, which contains three pages: `JSinteractionsPage`, `UleventargsPage`, and `UleventsPage`. The routes for pages are stated according to their names.

Execute JavaScript Function

JavaScript interaction happens through `IJSRuntime` interface, which needs to be injected for a page where you will use it.

Listing 6-1. JavaScript interactions page

```
@page "/JSinteractionsPage"
@inject IJSRuntime js

<p>a</p>
<p><input @bind="@a"></p>
```

```

<p>b</p>
<p><input @bind="@b"></p>

<p><button @onclick="@ (async () => await TestMethod())">test
</button></p>

<p>@result</p>
@code {
    double a;
    double b;
    double result;
    async Task TestMethod()
    {
        result = await js.InvokeAsync<double>("TestFunction",
        a, b);
    }
}

```

Listing 6-1 shows the contents of JSinteractionsPage.razor, where we will try to execute a JavaScript function.

Listing 6-2. Test function

```

<script>
    function TestFunction(a, b) {
        return a * b;
    }
</script>

```

The JavaScript function is declared in index.html (Listing 6-2) found in the root folder (wwwroot). It simply takes two variables and multiplies them. In our JSinteractionsPage (Listing 6-1), we have two input tags bound to two double variables, a result variable, and a button which on click/tap will execute our method, which in turn will execute the function in JavaScript. For that to happen, we first inject the IJSRuntime. After that,

we simply use the one and only method it has, `InvokeAsync`, which takes the name of the JavaScript function as its first argument, and the rest of the parameters will be the parameters passed in the JavaScript function. Do note that you can also create an object array for passing the parameters and always mind the return type of JavaScript function, so it is correct. Everything is quite simple, and there is nothing to worry if you need to access your old JavaScript libraries or the ones that may not be widely available for Blazor.

UI Events

HTML elements have several events, some generic and some tag specific. The good news is that you can use all of them directly in Blazor with no JavaScript interactions.

Listing 6-3. User interface events page

```
@page "/UIeventsPage"

<textarea @onpaste="@OnpasteTest"></textarea>
<video @onpause="@OnPauseTest" ></video>
<p>@output</p>
@code {
    string output;
    void OnpasteTest()
    {
        output = "text pasted";
    }

    void OnPauseTest()
    {
        output = "Don't give up watching";
    }
}
```

You have already used one of them, onclick, but now let us take a look at some more. Here (Listing 6-3) we have onpaste event for text area, which will occur once some text is pasted. As you can see, we execute a simple method which simply assigns a value to the output variable. The second one is video tag event specific, onpause; this one would occur once the video is paused. The important thing to remember is that a lot of events are element specific; therefore, you have to know which event belongs to which element.

UI Arguments

Every event has an argument and there is a special way to access it in Blazor. Arguments can be very useful to check new value, changes that occur, or get some other current data for UI element. Just remember, arguments are specific to events and events are specific to elements.

Listing 6-4. User interface arguments page

```
@page "/UIeventargsPage"

<p><input @onchange="@{async (changeargs) => await TestChange
Arguments(changeargs)}" /></p>

<p><div style="width:300px;height:300px;border:3px solid
#ff0000" type="checkbox" @onmousemove="@{async (changeargs) =>
await TestMouseArguments(changeargs)}" ></div></p>

<p>@output</p>
@code {
    string output;
    Task TestChangeArguments(ChangeEventArgs e)
    {
        output = (string)e.Value;
        return Task.CompletedTask;
    }
}
```

```
Task TestMouseArguments(MouseEventArgs e)
{
    output = "x: " + e.ScreenX + "; y: " + e.ScreenY;
    return Task.CompletedTask;
}
}
```

Our example (Listing 6-4) has two events with two different arguments, so we will start with the first one – `UIChangeEventArgs`. This particular one holds very little information, but the most important piece is the change value. The argument is declared in the `TestChangeArguments` Task, in it we simply access the parameter `e` and take the value (type object) and set it to the output variable. The more difficult part is the execution of it. You need to do it through lambda expression, and it is best done using an asynchronous way as shown in the input. The next one is a bit more exciting – `UIMouseEventArgs`; this one gives you information about your mouse. You will get position, button clicked, and some other data. We are executing this one in our div element using `onmousemove` event.

Do mind that the event will only occur when you are hovering inside the div element.

As you can see, this can be very useful in dynamic calculations or some other advanced UI. For example, drawing applications or even drag-and-drop functionality can be established using these arguments and the associated events.

Local Storage

For the storage section, we will use one single project and explore some possible options in it.

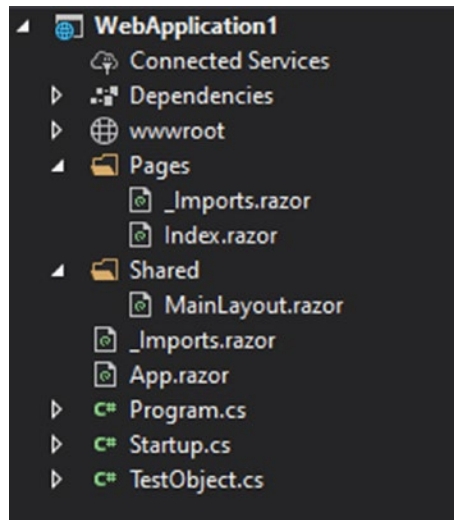


Figure 6-2. *Basic template of project*

The project (Figure 6-2) is a basic template with unwanted files removed and TestObject class added to it; we will explore that one later. The whole code is in the Index.razor; therefore, you will not need to create any other additional files.

Where to Store?

This question will surely be asked in one of your projects and most likely most of your projects. Of course, your first option is simply to store your values in a static variable defined in a class. While that may not be a viable long-term solution, it is the most efficient one and it is the best option if you simply want to move variables from page to page. It would apply for things like authentication tokens, color scheme, and other settings. The other options are more traditional and more permanent, that is, store data in local storage or session storage. Local is great for things like “remember me” or session-free settings; on the other hand, session will be destroyed once the tab is closed which is very similar to just storing in variables and makes this option rather pointless.

Of course, for modern applications you will most likely need to store the data on the server at some point, as that would be the safest as well as the most convenient way to do things. But as long as you do not need to preserve the data, use one of these options and try to save some resources of your server.

Store Text

If you do decide that you need something more permanent, this is what you will need to do.

Listing 6-5. Javascript interactions for local storage access

```
@page "/"
@inject IJSRuntime js

<p>test key</p>
<p><input @bind="@testkey"></p>
<p>test value</p>
<p><input @bind="@testvalue"></p>
<p><button @onclick="@{async () => await AddStringToSession
Storage()}">Insert to session</button></p>
<p><button @onclick="@{async () => await AddStringToLocal
Storage()}">Insert to local</button></p>

<p>key to retrieve</p>
<p><input @bind="@testkey_forget"></p>
<p><button @onclick="@{async () => await GetStringFromSession
Storage()}">Get from session</button></p>

<p><button @onclick="@{async () => await GetStringFromLocal
Storage()}">Get from local</button></p>
<p>@output</p>
```

```

@code {
    string testkey;
    string testvalue;

    string testkey_forget;

    string output;

    async Task AddStringToLocalStorage()
    {
        await js.InvokeAsync<object>("localStorage.setItem",
            testkey, testvalue);
    }

    async Task GetStringFromLocalStorage()
    {
        output = await js.InvokeAsync<string>("localStorage.
            getItem", testkey_forget);
    }

    async Task AddStringToSessionStorage()
    {
        await js.InvokeAsync<object>("sessionStorage.setItem",
            testkey, testvalue);
    }

    async Task GetStringFromSessionStorage()
    {
        output = await js.InvokeAsync<string>("sessionStorage.
            getItem", testkey_forget);
    }
}

```

The code (Listing 6-5) in the index shows some options on how you would interact with JavaScript and set or get your variables. First, let us look at the methods. As you can see, they are quite basic and we do not

need to write any JavaScript as the functions are already included in either local or session storage. We have some input variables like `testkey` and `testvalue` which will be the values used for setting your variables. Then, we have `testkey_forget`, which is the value you will be retrieving, and finally, we have the output string which will be assigned once a specific value is retrieved. For handling errors, you can either use `try` or `catch` in C#, or for something more accurate, you would create your own functions in JavaScript.

Store Other Types

When you have a string, you can simply store it as is, and if you have an int, you simply convert it to a string. However, when you have some more complex object, it becomes more difficult. You basically have two ways to do it; you can either serialize your object to json or do something more advanced which is something we will look at.

Listing 6-6. Binary formatter in blazor component (page)

```
@page "/"
@inject IJSRuntime js
@using System.Runtime.Serialization.Formatters.Binary;

<p>key to retrieve</p>
<p><input @bind="@testkey_forget"></p>

<p><button @onclick="@(()=> await GetObjectFromLocal
Storage())">Get object from local</button></p>

<p>@output</p>

<p>Insert object</p>
<p>key</p>
<p><input @bind="@testkey_forobject"></p>
```

```

<p>id</p>
<p><input @bind="@ObjectToInsert.id"></p>
<p>value</p>
<p><input @bind="@ObjectToInsert.value"></p>
<p><button @onclick="@async () => await AddObjectToLocalStorage()">Insert object to local</button></p>

@code {
    string testkey_forget;

    string testkey_forobject;
    TestObject ObjectToInsert = new TestObject();

    string output;

    async Task AddObjectToLocalStorage()
    {
        BinaryFormatter formatter = new BinaryFormatter();
        var tempstream = new System.IO.MemoryStream();
        formatter.Serialize(tempstream, ObjectToInsert);
        string base64 = Convert.ToBase64String(tempstream.
            ToArray());
        await js.InvokeAsync<object>("localStorage.setItem",
            testkey_forobject, base64);
    }

    async Task GetObjectFromLocalStorage()
    {
        string base64 = await js.InvokeAsync<string>("local
            Storage.getItem", testkey_forget);
        output = base64;
        BinaryFormatter formatter = new BinaryFormatter();
    }
}

```

```

        var tempstream = new System.IO.MemoryStream(Convert.
        FromBase64String(base64));
        ObjectToInsert = (TestObject)formatter.
        Deserialize(tempstream);
    }
}

```

The idea here (Listing 6-6) is to store raw object of c#, but you still need it to be a string. To accomplish that, we will convert the object to base64 using several steps. First, we need to serialize the object to a stream (in method `AddObjectToLocalStorage`); for that, we use `BinaryFormatter` (namespace `System.Runtime.Serialization.Formatters.Binary`). If you have never used binary formatter before, you should remember it, and it might change the way you develop applications. Getting back to the code, we establish a temporary stream to which we will serialize our object. After that, we simply pass the stream and the object to parameters of `Serialize` method in the formatter. Remember to use `MemoryStream` as it exposed the `ToArray` method, which will give the byte array of the stream. Finally, we simply convert the byte array to a base64 string and insert that to the local storage.

To retrieve your data, we will also need to use `BinaryFormatter`. In the `GetObjectFromLocalStorage`, we first retrieve base64 value from storage, which is then converted to a byte array and put in a stream. The stream is deserialized using `Deserialize` method, and the output is cast to your desired object type. In between, we output the base64 string just to see that it really works, and finally we assign that to the initial `ObjectToInsert` variable to see the output.

Pick and Save Files

File handling is a truly troublesome part of Blazor. While generating/downloading files is actually quite easy, picking one is a different story. Unfortunately, in its early versions, Blazor does not support any direct access to a file stream; therefore, the only way to use a file is by loading it to the memory, which in turn presents another problem, and that is the limits of the browser. We will take a look at a single example which picks a file and then saves it.

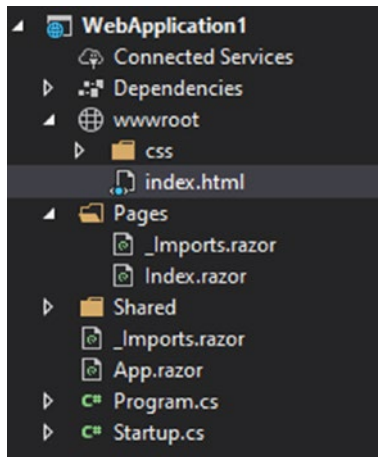


Figure 6-3. Example client-side project, default template

Notice that in this case (Figure 6-3) we will be using both `index.html` and `Index.razor`.

Listing 6-7. Index.html with javascript

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
```

```

<title>WebApplication1</title>
<base href="/">
<link href="css/bootstrap/bootstrap.min.css"
rel="stylesheet">
<link href="css/site.css" rel="stylesheet">
</head>
<body>
  <app>Loading...</app>

  <script src="_framework/blazor.webassembly.js"></script>
  <script>
    var FileManager = {
      downloadfile: function(name, bt64) {
var downloadlink = document.createElement('a');
downloadlink.download = name;
downloadlink.href = "data:application/octet-stream;
base64," + bt64;
document.body.appendChild(downloadlink);
downloadlink.click();
document.body.removeChild(downloadlink);
      },
      filedata: [],
readeron: true,

openreading: function () {
      document.getElementById("fileinput").click();
      return true;
    },
startreading: function () {
      this.readeron = true;

      var reader = new FileReader();
      reader.onloadend = function () {

```

```

        try {

            var dtw = new DataView(reader.result);

            for (var i = 0; i < dtw.byteLength; i++) {
                FileManager.filedata.push(dtw.getInt8(i));
            }
        } catch (e) {
            alert(e);
        }

        this.readeron = false;
    };
    reader.readAsArrayBuffer(document.
getElementById("fileinput").files[0]);
},

    getfile: function () {
        return FileManager.filedata;
    }
};
</script>
</body>
</html>

```

Listing 6-8. File reader in component

```

@page "/"
@using System.Linq;
@inject IJSRuntime js
<button @onclick="@{async () => await getfile()}">get file
</button>

```

```

<input id="fileinput" type="file" @onchange="@{async () =>
await OpenFile()}">
<button @onclick="@{async () => await
DownloadFile()}">Download</button>
<p>@statustext</p>
<p>@statustext1</p>
@code {
    byte[] selectedfilebytes;
    string statustext;
    string statustext1;
    async Task getfile()
    {
        await js.InvokeAsync<object>("FileManager.
        openreading");
    }

    async Task OpenFile()
    {
        try
        {
            statustext = "reading";
            await js.InvokeAsync<object>("FileManager.
            startreading");

            await Task.Delay(7000);
            int[] ob = await js.InvokeAsync<int[]>("FileManager.
            getfile");
            selectedfilebytes = ob.Select(x => (byte)x).
            ToArray();
            statustext1 = "done";
        }
        catch (Exception e)

```

```

        {
            statustext = e.Message + "\n\n" + e.InnerException;
        }
    }

    async Task DownloadFile()
    {
        string base64 = Convert.ToBase64String(selected
        filebytes);
        await js.InvokeAsync<object>("FileManager.
        downloadfile", "testfile_" + DateTime.UtcNow.
        ToFileTimeUtc().ToString() + ".avi",base64);
    }
}

```

Pick File

As mentioned before, picking a file is not a simple task and there are limits to it. But there are ways to get at least an image file into the memory in the C# part. The idea is to read the file stream in JavaScript, either read it as base64 or as an array of integers. We will go with integer array in this case, but the process for base64 would be very similar. The idea is to retrieve the string and convert it to byte array. The same goes for int array, with some differences in conversion.

First, we have some JavaScript code (Listing 6-7) in the index.html. We create a variable `FileManager` which will hold our functions. In the Index.razor (Listing 6-8), we have a fileinput with an id. We also have a button which basically triggers the click of the fileinput, and once the contents of fileinput change, another method is executed which then uses `JSinterop` to start reading the file. The file stream is read by the function `startreading` in the JavaScript part. We also have a JavaScript variable in the `FileManager`,

which is the part where we will set array of integers to be retrieved. To create that array, we first need to establish a data view from the result of the reader and loop through each element adding them to this basic array of integers. Getting back to the C# part (Listing 6-8), in this example after the reader starts, we delay execution for 7 seconds, so that the reader would finish and our array would be ready. After that, we retrieve the array and convert it to byte array using a very simple Linq arrangement. If you have never worked with bytes before, this may seem a bit confusing; how can an integer become a byte? A good way to clarify that would be for you to create a console application and break to check some byte array variable. In the byte array display, you will notice that it is actually an array of integers, and that is how our conversion works in this case.

As you can see, this is both a limited and a very inefficient way to deal with files. But having said that, the point is not to be efficient but rather not to use server resources. Businesses pay for server resources, not for browser resources, and that is where you make a difference. Just a simple image conversion can save you lots of money.

Save File

Saving a file may seem to be more of an unusual task, but it is a lot simpler and works better than picking one. Our JavaScript FileManager variable holds a function `downloadfile`, which takes the name of the file and the base64 string as the data of the file. After that, it is all quite simple; an element is created and a data link to href is added. At this point, you only need to initiate a click and the file will be downloaded. In the C# part, we `DownloadFile` method where we convert the byte array to base64 string and execute JavaScript function which does the downloading.

Summary

As you can see, Blazor is truly capable of doing anything you need to do. While some parts may be a bit unorthodox at this point, they still do the job, and in the future, we can expect them to be improved. Now that you have learned about Blazor, it is time to practice, and the following chapters will help you do just that.

CHAPTER 7

Practice Tasks for Server-side

Now that you are done with learning, you need to practice. We will start with two tasks for server-side Blazor and explore its use case further.

Task 1

The first task will be a simple project that only takes the data from user interface and inserts it into a database, or retrieves and displays the data. This is what server-side is really useful for, when you work with simple forms and you need quick access to the server.

Description

Create a product management dashboard.

The user should be able to

- Insert a product
- Retrieve product list
- Delete a product

Product data

- Id
- Title
- Seller's name
- Description
- Value

The product list items should also contain a button or other element which when clicked would delete the element.

Resources

Since we are not learning databases here, we need to create a fake one. This way, you can focus on Blazor-related matters only.

Listing 7-1. Fake database

using System.Collections.Generic;

namespace BlazorApp1

```
{  
    public class FakeDatabase  
    {  
        public static List<your product model> products = new  
            List<DataModels.ProductModel>();  
    }  
}
```

Create .cs file according to the code provided (Listing 7-1). This will be your database, where you will insert your product object, retrieve them, and delete them.

Solution

As usual, there are many solutions to this task, but we will still take a look at one possibility and explore it as much as possible. We will start with the general setup of the project and then move to services and then to pages.

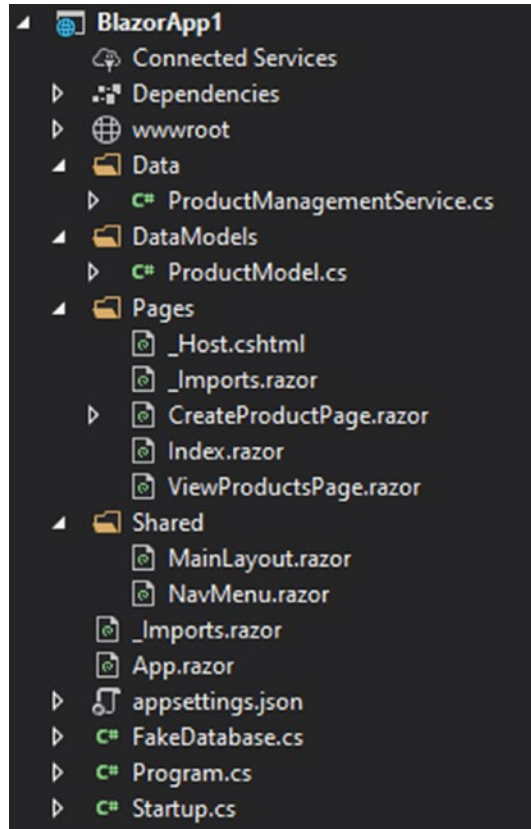


Figure 7-1. *The solution project*

As you can see (Figure 7-1), the project has all most default contents removed, but we still leave `Index.razor`, `MainLayout.razor`, and `NavMenu.razor`. First, the `Shared` folder contains our main layout, as well as the nav menu where we will have navigation setup for our two pages.

We also have the index page, which will only contain our navigation links. With that, we created a couple of pages for creating new product and retrieving the list. For the logic part, we have Data and DataModels folders. In the DataModels, we will have the model for the product and Data will contain our logic. This is a good way to lay out your project; this way, you know exactly where to file for their purposes.

Listing 7-2. Product model

```
using System;

namespace BlazorApp1.DataModels
{
    public class ProductModel
    {
        public Guid id { get; set; }
        public string title { get; set; }
        public string sellername { get; set; }
        public string description { get; set; }
        public decimal value { get; set; }
    }
}
```

The code (Listing 7-2) shows the contents of ProductModel.cs file found in DataModels folder.

Listing 7-3. Product management service

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

```
namespace BlazorApp1.Data
{
    public class ProductManagementService
    {
        public Task<bool> CreateProductAsync(DataModels.
            ProductModel pmodel)
        {
            try
            {
                FakeDatabase.products.Add(pmodel);
                return Task.FromResult(true);
            }
            catch (Exception)
            {
                return Task.FromResult(false);
            }
        }

        public Task<List<DataModels.ProductModel>>
            GetAllProductsAsync()
        {
            return Task.FromResult(FakeDatabase.products);
        }

        public Task<bool> DeleteProductAsync(Guid id)
        {
            try
            {
                FakeDatabase.products.Remove(FakeDatabase.
                    products.Where(x => x.id == id).ToArray()[0]);
                return Task.FromResult(true);
            }
        }
    }
}
```

```

        catch (Exception)
        {
            return Task.FromResult(false);
        }
    }
}

```

First, we need to establish the data model for the product. As you can see, it simply contains all the required properties, including an id of type Guid. This id needs to be referred to in your FakeDatabase class (see Listing 7-3). Once we have that set up, we can move on to the logic. We will only be using one service, kind of like you would have a controller in api. Except in this case, we have methods instead of http method parameters: POST (CreateProductAsync), GET (GetAllProductsAsync), and DELETE (DeleteProductAsync). This way, everything is conveniently placed and it is easy to find. The first method will simply take our model object as a parameter and insert it in the list in the FakeDatabase. The second one is even more basic as it only returns the list. Finally, the last one is a bit more complex; for you to make it more realistic, we want to pass only the id. The Remove method in the list type takes the whole object, so in this case, we have to use a little Linq to find it by id.

Listing 7-4. Service registry

```

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

```


You also have to register (Listing 7-4) your service in the Startup.cs. Afterward, you can move on to other tasks, although it would be recommended to just register the service after you have created the code file.

Listing 7-5. Create product page

```
@page "/createproductpage"
@Inject Data.ProductManagementService productmanagement

<p>title</p>
<p><input @bind="@producttoinsert.title"></p>
<p>seller name</p>
<p><input @bind="@producttoinsert.sellername"></p>
<p>description</p>
<p><textarea @bind="@producttoinsert.description"></p>
<p>value</p>
<p><input @bind="@producttoinsert.value"></p>
<p><button @onclick="@{async () => await
InsertNewProduct()}">Insert a product</button></p>
<p>@result</p>
@code {
    string result;
    DataModels.ProductModel producttoinsert = new
    DataModels.ProductModel() { id = Guid.NewGuid() };

    async Task InsertNewProduct()
    {
        if (await productmanagement.CreateProductAsync(product
        toinsert))
        {
            result = "product created";
            producttoinsert = new DataModels.ProductModel();
        }
    }
}
```

```

        else
        {
            result = "failed to create";
        }
    }
}

```

The create page (see Listing 7-5) we try to simplify as much as possible by binding the variables from a constructed object rather than declaring them separately in the page. But before anything else, we establish a route for the page and inject the product management service. In the code section, we have a result string which will simply tell us if the product was inserted successfully. After that, we declare a product variable, which has its contents bound to corresponding input fields. InsertNewProduct gets executed on the click of the button, and it executes CreateProductAsync then checks the return Boolean. Finally, the method re-assigns the producttoinsert variable, so that the new product could be inserted.

Listing 7-6. View products page

```

@page "/viewproductspage"
@Inject Data.ProductManagementService productmanagement

<table>
    <tbody>

        @if (products != null)
        {
            @foreach (var item in products)
            {
                <tr>
                    <td>@item.id</td>
                    <td>@item.title</td>

```

```

        <td>@item.description</td>
        <td>@item.sellername</td>
        <td>@item.value</td>
        <td><button @onclick="@{async () => await
Delete(item.id)}">Delete</button></td>
    </tr>
    }
}
else
{
}
</tbody>
</table>
@code {
    List<DataModels.ProductModel> products;

    protected override async Task OnInitializedAsync()
    {
        products = await productmanagement.
            GetAllProductsAsync();
    }

    async Task Delete(Guid id)
    {
        await productmanagement.DeleteProductAsync(id);
    }
}

```

For the product display, we have a rather complex page (see Listing 7-6), but to simplify it, we will be using a table to display our products. The alternative to that would be using components for each item. As always, we first declare a route for the page, and alongside that, we have an injection

for our main service. The code section contains one variable, that is, the list of products. We retrieve and assign the list once, on the initialization of the page. Alternatively, you may have chosen to add a refresh button or simply have a button that fetches data without doing that on initialization. We also have a Delete method which will delete the product. For the display, we first check if the list is assigned, and then we loop through each item by using foreach loop. The items are displayed in table data cells, with the exception of delete button. For the delete button, we establish an onclick event where we set our delete method and pass the id for the current item.

Listing 7-7. Navigation page

```
<div>
  <ul class="nav flex-column">
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="createproductpage" >
        <span class="oi oi-plus" aria-
          hidden="true">Create product </span></NavLink>
    </li>
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="viewproductspace">
        <span class="oi oi-home" aria-
          hidden="true">Manage products</span>
      </NavLink>
    </li>
  </ul>
</div>
```

Listing 7-8. Index page

```
@page "/"
```

```
<p><NavLink href="">Create product</NavLink></p>
```

```
<p><NavLink href="viewproductspage">View products</NavLink></p>
```

Finally, we have two ways to navigate to our pages. The first and initial option is to use the links in the index (Listing 7-8), and the second option is to go through the nav bar (Listing 7-7).

Task 2

This task will help you focus on component-based development – rather than working with lots of pages, you will rely on components.

Description

Create a basketball game tracking application. This particular application will focus on you using Blazor, but with that, you will need to explore how and why Blazor server-side would be useful for such task.

Teams A and B are tracked separately; you should be able to register a statistics item by clicking a single button.

Allow to register

Score 1 pt

Score 2 pt

Score 3 pt

Foul

Rebound

Block

You do not need to save anything, but do allow for that. Try to establish methods and structure on how you would save the data for the game.

Solution

Just like the first task and all the upcoming ones, this will not be the only solution. But this is one of the more efficient ones. We will explore the general logic, as well as how you could go further if you actually needed to save those updates.

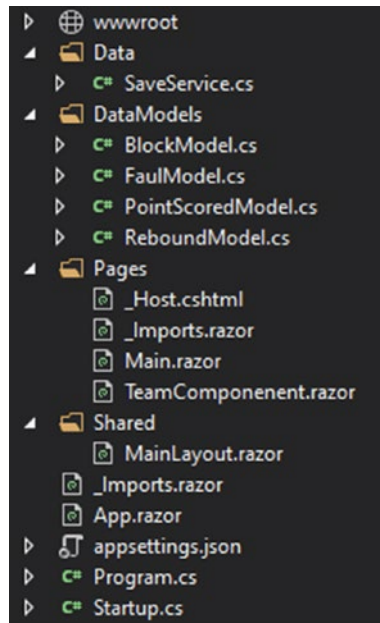


Figure 7-2. *Main.razor and TeamComponent.razor*

As you can see, there are only two pages: `Main.razor` and `TeamComponent.razor`. Also, the layout has been completely cleaned and we only have `@body` in it.

Listing 7-9. Point model

```
using System;

namespace WebApplication1.DataModels
{
    public class PointScoredModel
    {
        public Guid id { get; set; }
        public int value { get; set; }
    }
}
```

Listing 7-10. Rebound model

```
using System;

namespace WebApplication1.DataModels
{
    public class ReboundModel
    {
        public Guid id { get; set; }
    }
}
```

Listing 7-11. Foul model

```
using System;

namespace WebApplication1.DataModels
{
    public class FoulModel
    {
        public Guid id { get; set; }
    }
}
```

Listing 7-12. Block model

```
using System;

namespace WebApplication1.DataModels
{
    public class BlockModel
    {
        public Guid id { get; set; }
    }
}
```

As you can see, for the most part, the models are quite straightforward (see listings [7-9](#), [7-10](#), [7-11](#), [7-12](#)) with the exception of score. Since we have three types of score (1 pt, 2 pt, 3 pt), we could have three different models, but that would be inefficient and hard to read, and it would also present problems when displaying total score for the team. If need be, you can always expand these models – add time of the game, add quarter, and add player’s number.

Listing 7-13. Main page

```

<p>Current score:
@{
    int currentscore = 0;
}
@foreach (var item in PointsList)
{
    currentscore += item.value;
}
<label>@currentscore</label>
</p>
<p>Total fouls: @Foullist.Count</p>
<p>Total rebounds: @ReboundList.Count</p>
<p>Total blocks: @BlockList.Count</p>
<p><button @onclick="@(() => AddPoint(1))">Add 1 pt</button></p>
<p><button @onclick="@(() => AddPoint(2))">Add 2 pt</button></p>
<p><button @onclick="@(() => AddPoint(3))">Add 3 pt</button></p>
<p><button @onclick="@(() => AddFoul())">Add Foul</button></p>
<p><button @onclick="@(() => AddRebound())">Add rebound
</button></p>
<p><button @onclick="@(() => AddBlock())">Add block</button></p>
@code {
    [Parameter]
    public int team { get; set; } = 1; // A - 1 or B - 2

    [Parameter]
    public Guid gameid { get; set; }

    List<DataModels.PointScoredModel> PointsList = new
    List<DataModels.PointScoredModel>();

    List<DataModels.FoulModel> Foullist = new List<DataModels.
    FoulModel>();

```

```

List<DataModels.BlockModel> BlockList = new
List<DataModels.BlockModel>();

List<DataModels.ReboundModel> ReboundList = new
List<DataModels.ReboundModel>();

void AddPoint(int val)
{
    PointsList.Add(new DataModels.PointScoredModel { id =
        Guid.NewGuid(), value = val });
}

void AddFoul()
{
    Foullist.Add(new DataModels.FoulModel() { id = Guid.
        NewGuid() });
}

void AddBlock()
{
    BlockList.Add(new DataModels.BlockModel() { id = Guid.
        NewGuid() });
}

void AddRebound()
{
    ReboundList.Add(new DataModels.ReboundModel() { id =
        Guid.NewGuid() });
}
}

```

As you have seen in Figure 7-2, we only have one component for team data and we have two teams. Therefore, we need to identify each component, and we do that by passing an integer as a parameter. We also pass the game id as parameter, where the id will be generated in

Main.razor. Going further with the variables, you can notice four lists created, each with their own type of object. Also, you can see that we only have one list for score, even though a score has three types. The types of score are declared as value in the record. To display the current results, for the most part, we simply bind Count property of the Lists with the exception of score. For the score, we are displaying total; therefore, we need to calculate that. To make it simple, we just run a loop in the page, which gets re-run every time the count property changes. Finally, we have a few methods that will simply add new item on click.

Listing 7-14. Index page

```
@page "/"

<div style="width:50%;float:left;">
    <TeamComponent team="1" gameid="@gameid">
    </TeamComponent>
</div>

<div style="width:50%;float:left;">
    <TeamComponent team="2" gameid="@gameid" >
    </TeamComponent>
</div>

@code {
    Guid gameid;

    protected override Task OnInitializedAsync() {
        gameid = Guid.NewGuid();
        return base.OnInitializedAsync();
    }
}
```

For our Main page, we have a default route declared, so it works like your generic Index.razor, except in this case, we have Main.razor. We also have our gameid variable declared, which is set on initialization, although you could simply set it on declaration. We also have two div elements, in which our team components are set. And as planned, we pass integers for each team; with that, we have gameid as well.

Listing 7-15. services

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

namespace WebApplication1.Data
{
    public class SaveService
    {
        public async Task SaveProgress(Guid gameid,
            int team,params object[] datatosave)
        {

        }

        public async Task SaveProgress_Score(Guid gameid,
            int team, List<DataModels.PointScoredModel> scores)
        {

        }

        public async Task SaveProgress_Fouls(Guid gameid,
            int team, List<DataModels.FoulModel> fouls)
        {

        }
    }
}
```

```

public async Task SaveProgress_Rebounds(Guid gameid,
int team, List<DataModels.ReboundModel> rebounds)
{

}

public async Task SaveProgress_Blocks(Guid gameid,
int team, List<DataModels.BlockModel> blocks)
{

}

}
}

```

For the saving of records (Listing 7-15), we can elect to have a couple of options. If, say, you decide to include quarters and timer in general, you will probably want to save the whole thing after the end of some period of time. But if you want to be really safe, you will save on every action; therefore, you will need a method like `SaveProgress_Blocks`.

Summary

As you can see, server-side Blazor is really convenient, but it is important not to forget that it uses lots of server resources. With that in mind, it is best used for tasks where requirement is for data to reach server frequently.

CHAPTER 8

Practice Tasks for Client-side

We have already learned a lot from this book, but to truly learn, you have to practice. In this chapter, we will have a couple of projects for you to build.

You will find in this chapter

- Description for the first task
- Solution for the first task
- Description for the second task
- Solution for the second task

Task 1

Your first task will provide with several simple exercises to practice general syntax of Blazor, as well as a more complex exercise where you will need to use components and local storage.

Description

Create Blazor client-side application that would allow you to make calculations according to the instructions provided.

Age Calculator

Age calculator simply allows to enter two dates and return the difference in years.

Cylinder Surface Area

Use the following formula:

$$A = 2\pi rh + 2\pi r^2$$

where

A = area

r = radius

h = height

Allow to calculate all variables from the rest of them.

Rectangular Area

Use the following formula:

$$A = a * b$$

where

A = area

a = side a

b = side b

Allow to calculate all variables from the rest of them.

Allow the calculations to be saved locally for later use.

Trapezoid Area Calculator

Use the following formula:

$$A = (a + b) / 2 * h$$

where

A = area

a = base 1

b = base 2

h = height

Allow to calculate all variables from the rest of them.

Area of Triangle Calculator

Use the following formula:

$$A = (h * b) / 2$$

where

A = area

h = height

b = base length

Allow to calculate all variables from the rest of them.

Rectangular Area Calculator

Use the following formula:

$$A = a * b$$

where

A = area

a = side a

b = side b

This calculation is quite basic, but there is an additional task to go along with it. You will need to locally save each calculation, if the user wants them to be saved. Then, the calculation history will be displayed in the page, and the user will be able to select one of them and insert variable values from the record.

Solution

Our solution will be separated in several parts, for each part of description and as always; this is just one of many possible solutions rather than the only one.

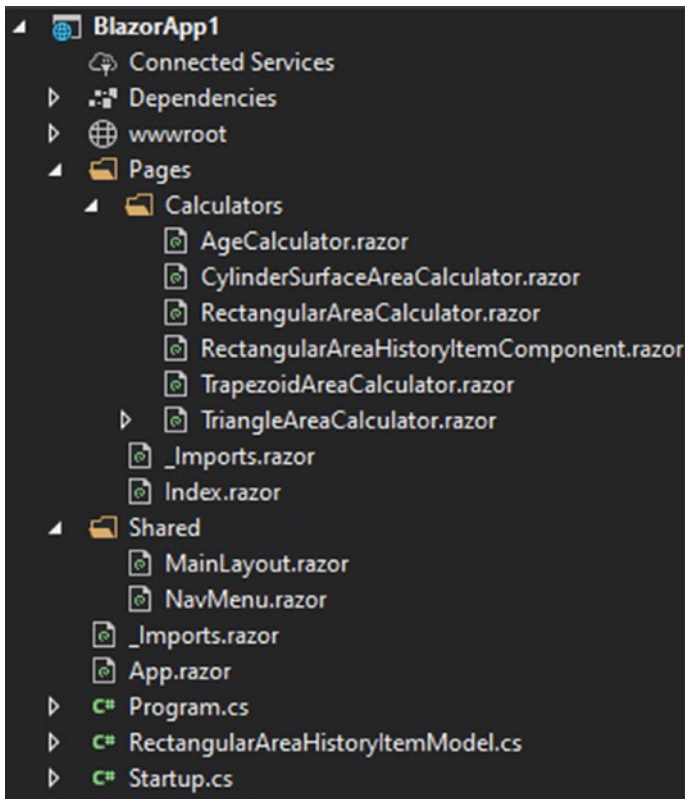


Figure 8-1. Project structure for the solution

Every calculation has its own page (see Figure 8-1), except for the rectangular where we also have a component for history.

Age Calculator Solution

Let's look into the solution to age calculator in this section.

Listing 8-1. Age calculator page

```
@page "/agecalculator"
<p>Birthdate: <input @onchange="@((args) => { birthdate = Convert.
ToDateTime((string)args.Value); Calculate(); })" type="date"/></p>

<p>To: <input @bind="@To" type="date"/></p>
<p><button @onclick="@InsertToday">Insert today</button></p>

<p>Age: @age</p>

@code {
    DateTime birthdate = new DateTime(1965,12,15);
    DateTime To = DateTime.Now;
    double age;

    void InsertToday()
    {
        To = DateTime.Now;
    }

    void Calculate()
    {
        age = birthdate.Subtract(To).TotalDays / 365;
    }
}
```

As you can see, the age calculator is quite straightforward (see Listing 8-1); we simply have two variables of type `datetime`, and we bind them with appropriate input fields. The more interesting part of this is how we execute the `Calculate` method. The task does not have any specific

requirements, but you can either do a simple button and execute it on click/tap or do something more exciting like we have here. On change of the value, we assign the new value to the variable, and with that, we also execute calculations. This is a good quick way to handle more than one operation in a single event.

Cylinder Surface Area Calculator

Let's look into calculating the surface area of cylinder in this section.

Listing 8-2. Cylinder surface area calculator page

```
@page "/cylindersurfaceareacalculator"
<h3>Cylinder Surface Area Calculator</h3>
<p><input class="inputstyle" @bind="@A" placeholder="A"/> =
2π * <input @bind="@r" class="inputstyle" placeholder="r"> *
<input @bind="@h" class="inputstyle" placeholder="h">
    + 2π * <input @bind="@r" class="inputstyle"
    placeholder="r" /><sup>2</sup></p>
<p>A = 2πrh + 2πr<sup>2</sup></p>
<p>A - area <button @onclick="@Calculate_A">calculate
A</button></p>
<p>r - radius <button @onclick="@Calculate_r">calculate
r</button></p>
<p>π - @Math.PI</p>
<p>h - height <button @onclick="@Calculate_h">calculate
h</button></p>
@code {
    double r = 0;
    double h = 0;
    double A = 0;
```

```

void Calculate_A()
{
    A = 2 * Math.PI * r * h + 2 * Math.PI * Math.Pow(h,2);
}

void Calculate_r()
{
    r = 0.5 * Math.Sqrt(Math.Pow(h, 2) + 2 * (A / Math.
    PI)) - (h / 2);
}

void Calculate_h()
{
    h = (A / (2 * Math.PI * r)) - r;
}
}

```

For the cylinder area calculator (Listing 8-2), we mostly have some basic calculations being made. The trick in this task is to put things in proper places and not make a mess of things, such as having the same formula for two different outputs. First, we display our formula in basic text format, and then we also have our formula with input fields in it. For each variable, we have different methods that calculated a value, and for each of them, we have different buttons that execute those methods.

Trapezoid area calculator

Let's look into calculating the area of trapezoid in this section.

Listing 8-3. Trapezoid area calculator page

```

@page "/trapezoidareacalculator"

<p>@A = (<input @bind="@a"    class="inputstyle"
placeholder="a"/> +

```

```

<input @bind="@b"    class="inputstyle" placeholder="b">)
/ 2 * <input @bind="@h"    class="inputstyle"
placeholder="h"></p>
<p>A = (a + b) / 2 * h</p>
<p>A - area</p>
<p>a - base 1</p>
<p>b - base 2</p>
<p>h - height</p>
<p><button @onclick="@Calculate"></button></p>

@code {
    double A;
    double a;
    double b;
    double h;

    void Calculate()
    {
        A = (a + b) / 2 * h;
    }
}

```

Trapezoid calculator is very similar to the previous one (Listing 8-2). Again, the difficulty is not in finding something new, but rather in properly assigning all the variables where they fit (Listing 8-3).

Triangle Area Calculator

Let's look into calculating the area of triangle in this section.

Listing 8-4. Triangle area calculator page

```
@page "/triangleareacalculator"
    <p>
        <input class="inputstyle" @bind="@A" placeholder="A"> =
        (<input class="inputstyle" @bind="@h" placeholder="h"> *
        <input class="inputstyle" @bind="@b" placeholder="b">) / 2
    </p>

    <p>A = (h * b) / 2</p>
    <p>A - area <button @onclick="@Calculate_A">Calculate
    </button></p>
    <p>h - height <button @onclick="@Calculate_h">Calculate
    </button></p>
    <p>b - base length <button @onclick="@Calculate_b">Calculate
    </button></p>

@code {
    double A;
    double h;
    double b;

    void Calculate_A()
    {
        A = (h * b) / 2;
    }

    void Calculate_h()
    {
        h = A * 2 / b;
    }
}
```

```

    void Calculate_b()
    {
        b = A * 2 / h;
    }
}

```

Our triangle calculation (Listing 8-4) is once again quite basic, and this is just one way to do it. You can, of course, do it on different events, or you may want to just display the output in a way where you would not have interactive formula.

Rectangle Area Calculator

Let's look into calculating the area of rectangle in this section.

Listing 8-5. Calculation history item component

```

<p>A: @item.A</p>
<p>side a: @item.a</p>
<p>side b: @item.b</p>
<p><button @onclick="@{async () => await OnSelect.
InvokeAsync(item.id)}">Pick</button></p>
@code {
    [Parameter]
    public BlazorApp1.RectangularAreaHistoryItemModel item {
        get; set; }

    [Parameter]
    public EventCallback<string> OnSelect { get; set; }
}

```


Listing 8-6. Calculator page

```

@page "/rectangularareacalculator"
@inject IJSRuntime js
@using System.Runtime.Serialization.Formatters.Binary;

<p>
    <input @bind="@currentcalculation.A" class="inputstyle"
    placeholder="A"> =
    <input @bind="@currentcalculation.a" class="inputstyle"
    placeholder="a">
    *
    <input @bind="@currentcalculation.b" class="inputstyle"
    placeholder="b">
</p>
<p>A = a * b</p>
<p>A - area <button @onclick="@Calculate_A">Calculate
</button></p>
<p>a - side a <button @onclick="@Calculate_a">Calculate
</button></p>
<p>b - side b <button @onclick="@Calculate_b">Calculate
</button></p>
<p>Save calculations <input type="checkbox" @bind=
"@savecalculation" /></p>
<p>History: </p>
@foreach (var item in calculationhistory)
{
    <RectangularAreaHistoryItemComponent @key=
"@item.id" item="@item" OnSelect="Selected">
</RectangularAreaHistoryItemComponent>
}

```

```

@code {
    var calculationhistory = new List<RectangularAreaHistory
    ItemModel>();
    bool savecalculation;
    var currentcalculation = new
    RectangularAreaHistoryItemModel();

    void Calculate_A()
    {
        currentcalculation.A = currentcalculation.a *
        currentcalculation.b;
        if (savecalculation)
        {
            SaveCalculation();
        }
    }

    void Calculate_a()
    {
        currentcalculation.a = currentcalculation.A /
        currentcalculation.b;
        if (savecalculation)
        {
            SaveCalculation();
        }
    }

    void Calculate_b()
    {
        currentcalculation.b = currentcalculation.A /
        currentcalculation.a;
    }
}

```

```

        if (savecalculation)
        {
            SaveCalculation();
        }
    }

    async void SaveCalculation()
    {
        var formatter = new BinaryFormatter();
        var tempstream = new System.IO.MemoryStream();
        currentcalculation.id = Guid.NewGuid().ToString();
        calculationhistory.Add(currentcalculation);
        formatter.Serialize(tempstream, calculationhistory);
        string base64 = Convert.ToBase64String(tempstream.
            ToArray());
        await js.InvokeAsync<object>("localStorage.removeItem",
            "rectareacalculationhistory");
        await js.InvokeAsync<object>("localStorage.setItem",
            "rectareacalculationhistory", base64);
    }

    protected override async Task OnInitializedAsync()
    {
        string base64 = await js.InvokeAsync<string>("local
            Storage.getItem", "rectareacalculationhistory");

        var formatter = new BinaryFormatter();
        var tempstream = new System.IO.MemoryStream(Convert.
            FromBase64String(base64));
        calculationhistory = (List<RectangularAreaHistoryItem
            Model>)formatter.Deserialize(tempstream);
    }

```

```

void Selected(string id)
{
    currentcalculation = calculationhistory.Find(x =>
        x.id == id);
}
}

```

The rectangular feature is a bit more complex than the others; besides the page, we will also need a component for history output. The component (Listing 8-5) does not do too much; it simply takes and displays the provided data and contains a callback variable, which will be used for removing the element. In the page (Listing 8-6), we first need to look at the method and insert the serialized list into local storage as base64. If our checkbox is checked, we execute SaveCalculation method on every calculation, if not – we do not save it, the way it gets save we have covered in Chapter 6; the only difference is that we want to clear the key/value pair without accessing JavaScript; therefore, we simply have to remove it first and create a new one. The reading part is also almost identical to the code we covered in Chapter 6; we only read the string and deserialize it to the list of our history items.

Listing 8-7. Navigation page

```

<div class="top-row pl-4 navbar navbar-dark">
    <a class="navbar-brand" href="">BlazorApp1</a>
    <button class="navbar-toggler" @onclick="ToggleNavMenu">
        <span class="navbar-toggler-icon"></span>
    </button>
</div>

<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
    <ul class="nav flex-column">
        <li class="nav-item px-3">

```

```

        <NavLink class="nav-link" href=""
        Match="NavLinkMatch.All">
            <span class="oi oi-home" aria-hidden="true">
                </span> Home
            </NavLink>
    </li>
    <li class="nav-item px-3">
        <NavLink class="nav-link" href="cylindersurfacearea
        calculator">
            Cylinder surface area calculator
        </NavLink>
    </li>
    <li class="nav-item px-3">
        <NavLink class="nav-link"
        href="triangleareacalculator">
            triangle area calculator
        </NavLink>
    </li>
    <li class="nav-item px-3">
        <NavLink class="nav-link" href="agecalculator">
            age calculator
        </NavLink>
    </li>
    <li class="nav-item px-3">
        <NavLink class="nav-link" href="rectangularareacal
        culator">
            Rectangle area calculator
        </NavLink>
    </li>

```

```

    <li class="nav-item px-3">
      <NavLink class="nav-link" href="trapezoidarea
        calculator">
        trapezoid area calculator
      </NavLink>
    </li>
  </ul>
</div>

@code {
  bool collapseNavMenu = true;

  string NavMenuCssClass => collapseNavMenu ? "collapse" : null;

  void ToggleNavMenu()
  {
    collapseNavMenu = !collapseNavMenu;
  }
}

```

Finally, since it suits us well, for the navigation we do not clear most of the defaults (see Listing 8-7); we just set up our navlinks according to the pages that we have. In real-world projects, it would be a good idea to change the designs if you take on this approach.

Task 2

Build an invoice generator. Invoice is basically a written request from one business to another for a payment. It states company details, items, values, and total values. Our version will be simplified.

Description

Since our invoice is simplified, we will only have a couple of company details, and the biggest part of the development will be sales item. The user should be able to add as many items as they want, and the total of each time should be added to the total of the invoice.

Inputs

- Id
- Description
- Total (generated from items)

Sales items

- Description
- Price
- Tax
- Total

In the sales items, you also need to provide the total for each item. Use component for sales item. Do note that you are not required to do any actual PDF, PNG, or other visual outputs of invoice.

Solution

Just like the previous task, this is not the only solution possible, but your project should have been very similar.

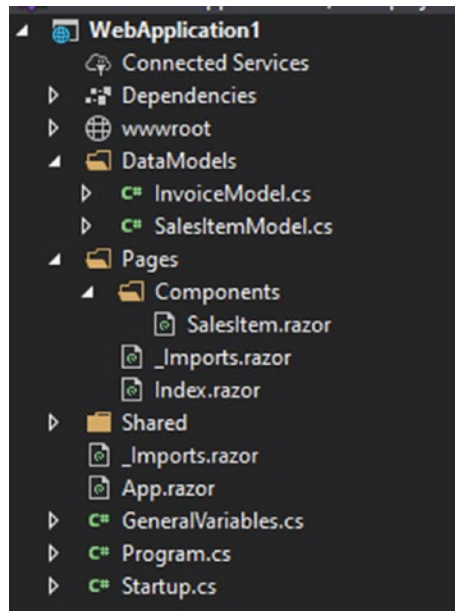


Figure 8-2. Project structure for the solution

Listing 8-8. Main layout

```
@inherits LayoutComponentBase

<div style="width:100%;float:left;">
    @Body
</div>
```

Since we only need one page for this application, we will not have too many files, but rather we will work on the index page and create a single component for sales item, which we will explore later. The layout is also very basic (Listing 8-8 and Figure 8-2); we have removed all the default stuff and left just a skeleton layout.

Listing 8-9. Invoice model

```
using System.Collections.Generic;

namespace WebApplication1.DataModels
{
    public class InvoiceModel
    {
        public string id { get; set; }
        public string description { get; set; }
        public double total { get; set; }
        public List<SalesItemModel> salesitems
        {
            get; set;
        }
    }
}
```

Listing 8-10. Sales item model

```
namespace WebApplication1.DataModels
{
    public class SalesItemModel
    {
        public string itemid { get; set; }
        public string description { get; set; }
        public double price { get; set; } = 0;
        public double tax { get; set; } = 0;
        public double total { get; set; } = 0;
    }
}
```

For our invoice, we need to create a couple of data models (see Listings 8-9, 8-10). While the task does not require us to generate the files, we still want to prepare for that. The invoice simply contains an id, description, total, and then the sales items added to it. The sales item is quite basic as well; we have item id, description, price, tax, and total. Everything here is very generic, and the interesting part will begin in the index page.

Listing 8-11. Main page

```
@page "/"
@inject IJSRuntime js
<div style="float:left;width:100%;">

    <p>Total: @total</p>
    <p>Total tax: @totaltax</p>
    <p>Description</p>
    <p><textarea @bind="@currentinvoice.description">
    </textarea></p>
    <p>Sales items</p>
    <p><button @onclick="@AddNewItem">Add</button></p>
</div>

@foreach (var item in currentinvoice.salesitems)
{
    <WebApplication1.Pages.Components.SalesItem
    OnDescription Change="ChangeForItemDescription" OnValueChange=
    "ChangeForItemValue" OnTotalChange="ChangeForItemTotal"
    OnTaxChange="ChangeForItemTax" OnRemove="RemoveItem" @key=
    "item.itemid" _itemid="@item.itemid"></WebApplication1.
    Pages.Components.SalesItem>
}

@code {
```

```

DataModels.InvoiceModel currentinvoice = new DataModels.
InvoiceModel() { id = Guid.NewGuid().ToString(),
salesitems = new List<DataModels.SalesItemModel>() };

double total = 0;
double totaltax = 0;

void AddNewItem()
{
    currentinvoice.salesitems.Add(new DataModels.
    SalesItemModel() { itemid = Guid.NewGuid().ToString() });
}

void RemoveItem( string id)
{
    currentinvoice.salesitems.Remove(currentinvoice.
    salesitems.Where(x => x.itemid == id).ToArray()[0]);
}

void ChangeForItemDescription(KeyValuePair<string,string>
args)
{
    currentinvoice.salesitems.Find(x => x.itemid == args.
    Key).description = args.Value;
}

void ChangeForItemValue(KeyValuePair<string,double> args)
{
    currentinvoice.salesitems.Find(x => x.itemid == args.
    Key).price = args.Value;
}

```

```

void ChangeForItemTax(KeyValuePair<string,double> args)
{
    currentinvoice.salesitems.Find(x => x.itemid == args.
    Key).tax = args.Value;
    totaltax = 0;
    foreach (var item in currentinvoice.salesitems)
    {
        totaltax += item.tax;
    }
}

void ChangeForItemTotal(KeyValuePair<string,double> args)
{
    currentinvoice.salesitems.Find(x => x.itemid == args.
    Key).total = args.Value;
    total = 0;
    foreach (var item in currentinvoice.salesitems)
    {
        total += item.total;
    }
}
}

```

Listing 8-12. Sales item component

```

<div style="float:left;width:100%;">
    <p><button @onclick="@{async () => await OnRemove.
    InvokeAsync(_itemid)}">Remove</button></p>
    <p>description:</p>
    <p><input @onchange="@{async (args) => await
    OnDescriptionChange.InvokeAsync(new KeyValuePair<string,
    string>(_itemid, (string)args.Value))}"></p>

```

```

    <p>value:</p>
    <p><input @onchange="@((args) => ReeveluateAfterValue
    Change(Convert.ToDouble(args.Value)))" ></p>

    <p>tax:</p>
    <p><input @onchange="@((args) => ReeveluateAfter
    TaxChange(Convert.ToDouble(args.Value)))" ></p>

    <p>total:</p>
    <p>@total</p>
    <p>@_itemid</p>
</div>
@code {
    [Parameter]
    public string _itemid { get; set; }

    [Parameter]
    public EventCallback<string> OnRemove { get; set; }

    [Parameter]
    public EventCallback<KeyValuePair<string,string>>
    OnDescriptionChange { get; set; }

    [Parameter]
    public EventCallback<KeyValuePair<string,double>>
    OnValueChange { get; set; }

    [Parameter]
    public EventCallback<KeyValuePair<string,double>>
    OnTaxChange { get; set; }

    [Parameter]
    public EventCallback<KeyValuePair<string,double>>
    OnTotalChange { get; set; }

```

```

double total;
double value;
double tax;

async void ReeveluateAfterValueChange(double newvalue)
{
    value = newvalue;
    await OnValueChange.InvokeAsync(new
    KeyValuePair<string, double>(_itemid,value));
    total = value + (tax / 100) * value;
    await OnTotalChange.InvokeAsync(new
    KeyValuePair<string, double>(_itemid, total));
}

async void ReeveluateAfterTaxChange(double newvalue)
{
    tax = newvalue;
    await OnValueChange.InvokeAsync(new
    KeyValuePair<string, double>(_itemid,value));
    total = value + (tax / 100) * value;
    await OnTotalChange.InvokeAsync(new
    KeyValuePair<string, double>(_itemid, total));
}
}

```

While the invoice page and items component may come across as complex, when you look closely, they only use the most basic features of Blazor. The most difficult part here is dealing with components and attempting calculation on changes. Component (Listing 8-12) simply takes an id for it, because when it gets generated, all the values are empty. The most important part here are the callbacks; as you can see, all the input fields have one and they all act differently. The description is the simplest one, as it only returns the id and the new description value. Tax and value

are more complex; we first need to establish methods, which will calculate the values in the component and display them in the component directly. Then, these methods invoke our callbacks; to get further, we need to switch to the page. Our page (Listing 8-11) handles the callbacks differently, but for the most part, the idea is to assign the values to the list of items, because that is what would be generated to some kind of visual format.

Summary

Both of these tasks not only give you an opportunity to practice your skills but also show you how client-side Blazor can make your business more efficient. For any of these tasks, there is absolutely no need to go to server-side, which saves you a lot of money. With that, the use of components simplifies development and keeps your code cleaner.

CHAPTER 9

Practice Task for Blazor Hosted

In this chapter, you will continue practicing what you have learned previously, but there will only be one project to complete. At this point, you should be comfortable with the client-side development, but a little more practice will not hurt.

This chapter will cover

- Introduction to the task
- Resources for the task
- Solution for the task

Task 1

For your Blazor hosted task, you will need to create a program that deals with statistics of poker players. During this task, you will notice how easy it is to use the shared data model feature, as well as how useful it is to have client-side file generation capability.

Description

You will need to first display a list of players, for which you can simply use a button containing the name of the player. On the selection (click) of the player, their statistics will be fetched and displayed. The statistics data for each player will be cached, meaning that if the player is selected again, after another one has been selected, the data will not be retrieved from the server unless the user refreshes it. The next part of the task is for client-side related; the program should allow to export a JSON file of selected player statistics or the statistics for all cached players.

Player statistics output

- Total tournaments
- Total winnings
- Tournaments in the money
- Date started
- Last updated (either retrieved or refreshed)

Resources

You will be provided with a class that contains a list of users and two methods that will retrieve both the list and statistics for each player.

Listing 9-1. Players.cs

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

namespace WebApplication1.Server.DataLogic
{
```

```
public class Players
{
    static List<PlayerData> PlayersList = new
    List<PlayerData>() {
        new PlayerData
        {
            id = 1,
            name = "John dow",
            totaltournaments = 1100,
            totalwinnings = 115000,
            totalinthemoney = 250,
            datestarted = DateTime.Parse("08/20/2005")
        },
        new PlayerData
        {
            id = 2,
            name = "John mark",
            totaltournaments = 1500,
            totalwinnings = 15005,
            totalinthemoney = 15,
            datestarted = DateTime.Parse("02/25/2009")
        },
        new PlayerData
        {
            id = 3,
            name = "John dean",
            totaltournaments = 1300,
            totalwinnings = 134000,
            totalinthemoney = 468,
            datestarted = DateTime.Parse("12/25/2017")
        },
    },
}
```

```

        new PlayerData
    {
        id = 4,
        name = "mark lee",
        totaltournaments = 150,
        totalwinnings = 5300,
        totalinthemoney = 7,
        datestarted = DateTime.Parse("06/25/2008")
    },
    new PlayerData
    {
        id = 5,
        name = "t young",
        totaltournaments = 101,
        totalwinnings = 18000,
        totalinthemoney = 19,
        datestarted = DateTime.Parse("08/25/2013")
    },
    new PlayerData
    {
        id = 6,
        name = "richar right",
        totaltournaments = 36,
        totalwinnings = 1300000,
        totalinthemoney = 10,
        datestarted = DateTime.Parse("08/25/1995")
    }
};

class PlayerData
{
    public int id { get; set; }
    public string name { get; set; }
}

```

```

    public int totaltournaments { get; set; }
    public double totalwinnings { get; set; }
    public double totalinthemoney { get; set; }
    public DateTime datestarted { get; set; }
}

public static Task<List<Shared.PlayerListItem>>
RetrievePlayerList()
{
    List<Shared.PlayerListItem> templist = new
    List<Shared.PlayerListItem>();
    foreach (var item in PlayersList)
    {
        templist.Add(new Shared.PlayerListItem() { id =
            item.id, name = item.name });
    }
    return Task.FromResult(templist);
}

public static Task<Shared.PlayerStatisticsItem>
RetrievePlayerStatistics(int id)
{
    var selectedplayer = PlayersList.Where(cl => cl.id
    == id).ElementAt(0);
    return Task.FromResult(new Shared.
    PlayerStatisticsItem() {
        playerid = id,
        totaltournaments = selectedplayer.
        totaltournaments,
        totalinthemoney = selectedplayer.
        totalinthemoney,
        totalwinnings = selectedplayer.totalwinnings,

```

```

        datestarted = selectedplayer.datestarted,
        lastrefresh = DateTime.UtcNow
    });
}
}
}

```

First, we have our `PlayerData` model (see Listing 9-1), which will be our main model in this fake database. It will also create a static list, with some items for you to work with. You will only need to deal with the following methods:

- `RetrievePlayerList` – Fetches the list of players, but only takes and gives an id and a name for each player
- `RetrievePlayerStatistics` – Fetches the full details for a single player

Solution

Now that you have your task completed, we can take a look at a solution. Do mind that as long as it is working, it is probably right, but there are many ways to do it, and some ways might be more efficient than others.

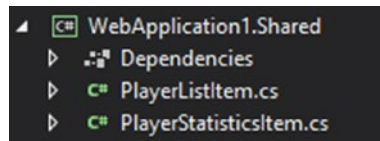


Figure 9-1. *Shared library*

Listing 9-2. Player list item model

```
namespace WebApplication1.Shared
{
    public class PlayerListItem
    {
        public string name { get; set; }

        public int id { get; set; }
    }
}
```

Listing 9-3. Player statistics item model

```
using System;

namespace WebApplication1.Shared
{
    public class PlayerStatisticsItem
    {
        public int playerid { get; set; }
        public int totaltournaments { get; set; }
        public double totalwinnings { get; set; }
        public double totalinthemoney { get; set; }
        public DateTime datestarted { get; set; }
        public DateTime lastrefresh { get; set; }
    }
}
```

First, we should start from the data models (see Figure 9-1) and then move to the back end. For this application, we will only need two models – one for listing (Listing 9-2) and the other one for statistics (Listing 9-3) of the user. As you can see, these do not contain all the user data, only what you need to display. Now that we have all that, we can move on to the server part (Figure 9-2).

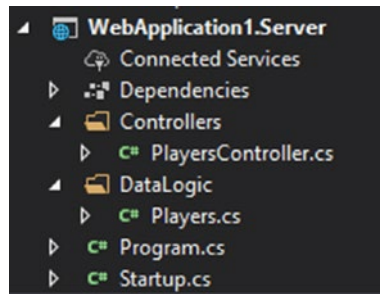


Figure 9-2. API project in the solution

Listing 9-4. Players controller

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace WebApplication1.Server.Controllers
{
    public class PlayersController : Controller
    {
        [Route("/retrieveplayerslist")]
        [HttpGet]
        public async Task<List<Shared.
        PlayerListItem>> GetPlayers()
        {
            return await DataLogic.Players.
            RetrievePlayerList();
        }

        [Route("/retrieveplayerstats")]
        [HttpGet]
        public async Task<Shared.PlayerStatisticsItem>
        GetPlayerStats(int id)
```

```

    {
        return await DataLogic.Players.
            RetrievePlayerStatistics(id);
    }
}

```

First, we need to create a `Player.cs` class, where we simply insert contents provided in the resources. The file is conveniently placed in the `DataLogic` folder, which in this structure would only contain classes that have methods for data retrieval, insertion, or other database-related procedures. After we have that, we can move on to the controller (Listing 9-4) where we have two routes – for list and statistics. As you can see, the action methods only execute the static methods from data logic; this helps to keep the controller completely clean.

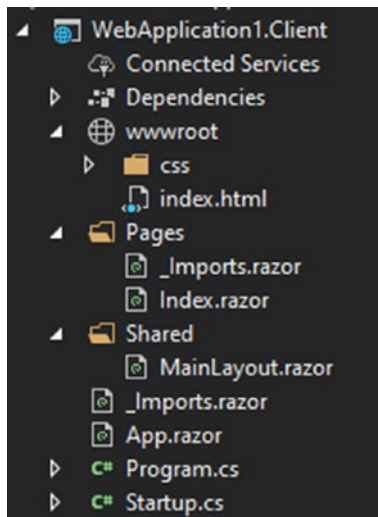


Figure 9-3. Client project in the solution

Listing 9-5. Main page

```

@page "/"
@using datamodels = WebApplication1.Shared;
@using Newtonsoft.Json;
@inject HttpClient http
@inject IJSRuntime jsruntime

```

Everything related to client-side (Figure 9-3) will be done in one single page (Index.razor in this case). However, you may have chosen to use components which would have made the page look cleaner, although it would have taken more time to set up. First, we need to establish some general declarations (Listing 9-5), starting with the page route. With the route, we need to declare a using statement for the shared folder; since we already have a namespace “Shared” in the client part, we elect to use a name for the namespace WebApplication1.Shared – “datamodels”. We will be using Newtonsoft for our JSON exports; therefore, it is convenient to declare Newtonsoft.Json namespace. Finally, we have two injections – one for the http client which will be used in API calls and the other IJSRuntime which we will use to save the file.

Listing 9-6. Player statistics component

```

<div>
    <p><button @onclick="@{async () => await
    FetchPlayers()}">Fetch players</button></p>
    @if (listofplayers.Count > 0)
    {
        foreach (var item in listofplayers)
        {
            <p><button @onclick="@{async () => await
            ShowPlayerStatistics(item.id)}">@item.name
            </button></p>

```

```

    }
  }
  else
  {
    <p>No players available</p>
  }
</div>

<div>
  @if (CurrentPlayerDisplayed != null)
  {
    <p>total tournaments played:
    @CurrentPlayerDisplayed.totaltournaments</p>
    <p>total winnings: @CurrentPlayerDisplayed.
    totalwinnings</p>
    <p>total in the money: @CurrentPlayerDisplayed.
    totalinthemoney</p>
    <p>date started: @CurrentPlayerDisplayed.
    datestarted.ToShortDateString()</p>
    <p>Last refreshed: @(Math.
    Ceiling(CurrentPlayerDisplayed.lastrefresh.
    Subtract(DateTime.UtcNow).TotalMinutes)) minutes
    ago</p>
    <p><button @onclick="@(async () => await Ref
    reshPlayerStatistics(CurrentPlayerDisplayed.
    playerid))">Refresh</button></p>
    <p><button @onclick="@(async () => await
    ExportCurrentPlayer())">Export player</button></p>
  }

```

```

        else
        {
            <p>Select a player for display</p>
        }
    </div>
</div>

<p><button @onclick="@(async () => await
ExportAllPlayers())" >Export all players</button></p>
</div>

@code {
    List<datamodels.PlayerListItem> listofplayers = new
    List<datamodels.PlayerListItem>();
    Dictionary<int, WebApplication1.Shared.
    PlayerStatisticsItem> PlayerStatisticsCache =
    new Dictionary<int, WebApplication1.Shared.
    PlayerStatisticsItem>();

    datamodels.PlayerStatisticsItem CurrentPlayerDisplayed =
    null;

    async Task FetchPlayers()
    {
        listofplayers = await http.
        GetJsonAsync<List<datamodels.PlayerListItem>>
        ("/retrieveplayerslist");
    }

    async Task ShowPlayerStatistics(int id)
    {
        bool iscached = PlayerStatisticsCache.TryGetValue(id,
        out CurrentPlayerDisplayed);
    }
}

```

```

    if (!iscached)
    {
        CurrentPlayerDisplayed = await http.
            GetJsonAsync<datamodels.PlayerStatisticsItem>
            ("/retrieveplayerstats?id="+id);
        PlayerStatisticsCache.Add(id,
            CurrentPlayerDisplayed);
    }
}

async Task RefreshPlayerStatistics(int id)
{
    CurrentPlayerDisplayed = await http.
        GetJsonAsync<datamodels.PlayerStatisticsItem>
        ("/retrieveplayerstats?id="+id);
    PlayerStatisticsCache[id] = CurrentPlayerDisplayed;
}

async Task ExportCurrentPlayer()
{
    string json = JsonConvert.SerializeObject(CurrentPlayer
        Displayed);
    string base64 = Convert.ToBase64String(System.Text.
        Encoding.UTF8.GetBytes(json));
    await jsruntime.InvokeAsync<object>("downloadfile",
        "PlayerStats_" + DateTime.UtcNow.ToFileTimeUtc().
        ToString() + ".json",base64);
}

```

```

    async Task ExportAllPlayers()
    {
        string json = JsonConvert.SerializeObject(Player
            StatisticsCache);
        string base64 = Convert.ToBase64String(System.Text.
            Encoding.UTF8.GetBytes(json));
        await jsruntime.InvokeAsync<object>("downloadfile",
            "AllPlayers_" + DateTime.UtcNow.ToFileTimeUtc().
            ToString() + ".json",base64);
    }
}

```

We start the Index.razor (Listing 9-6) page by declaring two variables which will be used to display outputs. The first one is the list `listofplayers`; this will hold the list objects where we find name and id. The method `FetchPlayers` will retrieve players from the server and assign them to the list. After that, the list is displayed it in the first div element – here we first check if the list contains any items; if not, we simply tell the user that the system has no players to display. On the other hand, if the list is filled, we will go through each item and assign the values to the button text (name), and in the onclick for the button, we will pass a parameter (id) to the method `ShowPlayerStatistics`. This method will retrieve the statistics data for that specific user from the server, but before that, we will try to check our dictionary `PlayerStatisticsCache` to see if the player statistics have already been cached; if they have, we will simply assign that to the `CurrentPlayerDisplayed` variable, and if not, we will retrieve the data and assign it then. Once it is assigned, the “if” statement is re-evaluated and the variables in the object are updated in all the necessary places.

Listing 9-7. File download script (javascript)

```
function downloadfile(name, bt64) {
    var downloadlink = document.createElement('a');
    downloadlink.download = name;
    downloadlink.href = "data:application/octet-stream;base64,"
    + bt64;
    document.body.appendChild(downloadlink);
    downloadlink.click();
    document.body.removeChild(downloadlink);
}
```

For the exporting part, we will need to use a bit of JavaScript, just to establish one function (Listing 9-7) which will “download” the file from the client-side. Then, to export a single player, we will use `ExportCurrentPlayer`, which serializes the `CurrentPlayerDisplayed` variable to a json string, then converts a string to a byte array, and finally converts that to Base64 string. Once we have a Base64 string, we can pass it to the JavaScript alongside the name for the file. The exportation of all players would use the method `ExportAllPlayers`, and it would work exactly the same way, except it would use `PlayerStatisticsCache` variable.

Summary

As you probably have noticed throughout this project, merging API and client-side solutions is a lot more efficient, mainly because of the shared models’ libraries. The testing is also a bit easier, as you do not need to launch two projects at the same time. With all the knowledge that you acquired and the practice that you have done, you should be able to develop real-world projects now.

Index

A

AddElement method, [57](#)

AddHobby method, [62](#)

API calls

- basic routes, [80](#)

- buyer controller, [90, 91](#)

- buyer model, [87, 89](#)

- buyer page, [95, 96](#)

- components, [99, 100](#)

- HTTP client

 - manipulations, [84, 86](#)

- JSONfull way, [81–84](#)

- product controller, [91, 93](#)

- product model, [88, 89](#)

- purchase component, [110, 111](#)

- purchase controller, [94, 95](#)

- purchase model, [88, 89](#)

- shared library, [77, 78](#)

- static list variables, [89, 90](#)

- web api and shared library, [87](#)

- web api part, [78](#)

B

Binding

- element, [12](#)

- functions section

 - paragraph tag, [13](#)

- parameters, accept and

 - pass, [14, 15](#)

- <script> tag, [13](#)

- testparam, [15](#)

- two-way, [14](#)

- page events, [16, 17](#)

Blazor, [1, 2](#)

- client hosted, [5, 6](#)

- client-side, [4, 5](#)

- save file, [130](#)

- server-side, [3, 4](#)

Blazor hosted

- clean up template, [73, 74](#)

 - client-side project, [72](#)

 - server-side project, [72, 73](#)

 - shared library .net standard

 - project, [73](#)

- navigation

 - API part, [76](#)

 - client-side part, [75](#)

 - server part, [76](#)

 - TestPage.razor, [77](#)

- structure

 - client-side project, [67, 68](#)

 - server-side project, [68](#)

 - shared library .net standard

 - project, [68](#)

 - Startup.cs file, [69–71](#)

INDEX

Blazor hosted task

- API project, [185, 186](#)
 - client project, [187, 188](#)
 - controller, [186, 187](#)
 - data models, [185](#)
 - declarations, [188](#)
 - exportation, [193](#)
 - Index.razor, [188, 189, 191, 192](#)
 - Newtonsoft.Json
 - namespace, [188](#)
 - player.cs, [180–184, 187](#)
 - player statistics, [180](#)
 - poker players, [179](#)
 - shared library, [184](#)
- BuyerForPurchaseComponent.
razor, [104, 105, 107, 108](#)

C

Client-side

- clean up template
 - MainLayout.razor, [48](#)
 - project template, [47, 48](#)
- components, [54](#)
 - complex system, [54](#)
 - Index.razor, [55, 56](#)
 - preservation, [56–58](#)
 - static variable, [55](#)
 - TestChanged event, [55](#)
- index.html/Index.razor,
[125, 127–129](#)
- navigation
 - advanced page, [51](#)
 - basic page, [50](#)

- component creation, [59, 60](#)
 - HobbyModel, [60](#)
 - NavigationManager, [50](#)
 - NavMenu component, [52, 53](#)
 - pages, [50](#)
 - parameter, [51, 52](#)
 - preview page, [63, 64](#)
 - project structure, [49, 58](#)
 - query string, [64](#)
 - Sign up page, [60–62](#)
- program and startup
- AddComponent<App>, [46](#)
 - App.razor file, [46](#)
 - host builder, [44](#)
 - index.html contents, [44, 45](#)
- template, [125](#)

Client-side tasks

- calculations
 - age calculator, [154](#)
 - area of triangle calculator, [155](#)
 - cylinder surface area, [154](#)
 - rectangular area, [154, 156](#)
 - Trapezoid area calculator, [155](#)
- invoice generator, [169](#)
 - components, [175–177](#)
 - data models, [172, 173](#)
 - index page, [173](#)
 - invoice page, [177](#)
 - layout, [171](#)
 - project structure, [171](#)
- solution
 - age calculator, [158](#)
 - cylinder surface area, [159, 160](#)
 - project structure, [157](#)

rectangle area, [163–167](#), [169](#)
 trapezoid area, [160](#), [161](#)
 triangle area, [162](#), [163](#)

CreateHostBuilder, [44](#)

CreateProductAsync
 method, [138](#), [140](#)

CreateProductPage.razor, [96–99](#)

CreatePurchasePage.razor,
[100–104](#), [107](#), [108](#)

D, E

DeleteHobby method, [63](#)

DeleteProductAsync
 method, [138](#)

F

FakeDatabase class, [138](#)

FetchPlayers method, [192](#)

G, H

GetAllProductsAsync
 method, [138](#)

GetJsonAsync method, [83](#), [84](#)

GetTest1 method, [84](#), [86](#)

GetTest2 method, [84](#), [86](#)

I

Index.razor, [25](#), [55](#), [74](#), [125](#),
[135](#), [192](#)

InvokeAsync method, [116](#)

J, K, L

JavaScript code, [129](#)

JavaScript function
 execution, [114](#), [115](#)
 project layout, [114](#)
 UI arguments, [117](#), [118](#)
 UI events, [116](#)

JSinteractionsPage.razor, [115](#)

JSinterop, [129](#)

M

MainLayout.razor, [26](#), [48](#), [74](#), [135](#)

Main.razor, [144](#), [145](#)

MapDefaultControllerRoute, [71](#)

MapFallbackToClientSideBlazor, [71](#)

N

NavigateTo method, [50](#)

Navigation project
 contact page, [38](#), [39](#), [42](#)
 cpnservice, [38](#)
 home page, [34](#)
 InfoPage, [36](#), [37](#)
 injection of service, [41](#)
 register service, [40](#)
 testing, [34](#), [35](#)
 web site, [33](#)

NavMenu.razor, [26](#), [135](#)

O

OnInitializedAsync method, [84](#)

P, Q

Player.cs class, [187](#)
 PostJsonAsync method, [84](#)
 ProductForPurchasePage
 Component.razor, [106–108](#)
 ProductModel.cs file, [136](#)

R

Razor *vs.* Blazor, [7](#)
 comment syntax, [8](#)
 for loop, [11](#)
 functions section, [11](#)
 if statement, [10](#)
 sections, [9](#)
 RemoveElement method, [57](#)
 RenderComponentAsync
 method, [28](#)

S

SaveCalculation method, [167](#)
 SendAsync method, [86](#)
 Server-side navigation
 components, [29, 30](#)
 HTML element, [26](#)
 MainLayout.razor, [26, 27](#)
 pages, [27–29](#)
 parameters, [30–32](#)
 project (*see* Navigation project)
 Shared folder, [26](#)
 Server-side tasks
 basketball game tracking, [143](#)
 count property, [149](#)

data model, [138](#)
 links, [143](#)
 Main.razor, [144, 145, 150](#)
 models, [146](#)
 nav bar, [142](#)
 page creation, [139, 140](#)
 product display, [140, 141](#)
 product management
 dashboard, [133, 134](#)
 resources, [134](#)
 saving of records, [150, 151](#)
 scores, [146](#)
 solution projects, [135](#)
 Startup.cs. file, [139](#)
 TeamComponent.razor, [144, 145](#)

Server-side technology

injection
 data folder, [23](#)
 Index.razor file, [25](#)
 Startup.cs, [25](#)
 TestService.cs file, [24](#)

startup

AddRazorPages, [22](#)
 AddServerSideBlazor
 method, [22](#)
 .NET Core, [19](#)
 Program.cs, [19, 20](#)
 Startup.cs, [20, 22](#)

ShowPlayerStatistics

method, [192](#)

Storage

BinaryFormatter, [124](#)
 location, [119](#)
 object to Base64, [122–124](#)

object to json, [122](#)
template, [118](#), [119](#)
text, [120](#), [121](#)

T

TeamComponent.razor, [144](#), [145](#)
TestChanged event, [55](#)

U, V

UseBlazorDebugging, [71](#)
UseClientSideBlazorFiles, [71](#)

W, X, Y, Z

WebAssembly, [2](#), [5](#), [6](#)