Building Modern Web Applications

with

ASP.NET Core Blazor

Learn how to use Blazor to create powerful, responsive, and engaging web applications

Brian Ding

bpb

Building Modern Web Applications with ASP.NET Core Blazor

Learn how to use Blazor to create powerful, responsive, and engaging web applications

Brian Ding

www.bpbonline.com

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-798

www.bpbonline.com

Dedicated to

My beloved parents:
Zhong Ding
Yi Hu
&
My wife, Haoran Diao

About the Author

Brian Ding has over 8 years of experience in TypeScript and .NET development, specializing in areas such as WinForm, WPF, ASP.NET, and ASP.NET Core. Currently employed at BMW Archermind Information Technology Co Ltd, he holds the position of tech leader, where he focuses on creating engaging digital driving experiences for BMW customers. Throughout his career, Brian has worked in diverse domains including Software Development, DevOps, Automation tools, and Cloud Technologies. His passion lies in coding and developing scalable solutions that are easy to maintain and adaptable.

About the Reviewer

Trilok Sharma is a seasoned technical architect with 14 years of expertise in designing, developing, and implementing enterprise-level solutions on the Microsoft technology stack. Throughout his career, Trilok Sharma has demonstrated mastery in Microsoft technologies, including Blazor Server, Blazor Web Assembly (WASM), .Net 7.0, Net Core, C#, Angular, React, SQL Server, Azure, and AWS. He has a strong command over object-oriented programming principles and has leveraged their knowledge to architect scalable and efficient applications.

With their strong technical understanding, attention to detail, and commitment to quality, Trilok Sharma continues to make valuable contributions as a technical reviewer in the Microsoft technology space.

Trilok holds a Bachelor's in Computer Science and MBA in Project +IT Management.

Acknowledgement

This book would not exist without the help of many people, mostly including the continuous support from my parents and my wife's encouragement for writing the book. They've taken most of the housework so that I can focus on writing the book — I could have never completed this book without their support.

My gratitude also goes to the team at BPB Publications for being supportive enough to provide me with quite a long time to finish the book. This is my first book ever, and I would like to thank them for their professionalism, guidance, and patience along the way.

Preface

This book covers many different aspects of developing Blazor applications, a modern way to build rich UI web applications. And this book introduces how to leverage .NET and its eco-systems to build a modern enterprise application. This book will introduce WebAssembly and how it enables web applications to be written in any programming language. It also compares different Blazor hosting models and the strategy to select a model that suits that business requirements.

This book takes a demonstrative approach for Blazor learners. Every chapter comes with a lot of code examples and Blazor source code analysis. It covers basic Blazor directives and components and how these concepts can be combined together to build a more complex customized component. This book also explains some advanced techniques to control component rendering and improve performance.

This book is divided into 13 It will start with the introduction of WebAssembly and cover the basic concepts in Blazor Framework and some advanced techniques you may find handy when developing production-ready applications, as well as explaining source code structures and designing patterns and styles. So, readers can learn from the bottom how a Blazor application is running. The details are listed below.

Chapter 1: WebAssembly will introduce what WebAssembly is and the roadmap of WebAssembly. The chapter will explain why WebAssembly is

proposed while JavaScript is powerful enough. A hello world demonstration is given by compiling C/C++ source code into WebAssembly. Calling WebAssembly functions from JavaScript code will also be discussed. WASM binary format will be discussed along with the introduction to different sections in the binary code. It will introduce the popular languages that can produce WebAssembly modules, and ASP.NET Core Blazor is one of those platforms that can be leveraged to build web applications beyond WebAssembly.

Chapter 2: Choose Your Hosting will discuss WebSocket and compare the difference between WebSocket and HTTP. Will introduce SignalR, a .NET library that implements WebSocket and can fallback to long polling for compatibility. This chapter will introduce the basic structure of a Blazor application and compare three different Blazor hosting models, Blazor Server, Blazor WebAssembly, and Blazor Hybrid.

Chapter 3: Implementing Razor and Other will cover basic components. Blazor applications are made of components, and they share many useful features, including directives, binding, cascading, and event handling. It will explain the lifecycle of a typical component by introducing those virtual lifecycle methods that can be overridden. It will introduce layout, a special component type that can be useful in building an application with multiple functional spaces. Will introduce some popular third-party libraries that we can use to build enterprise applications.

Chapter 4: Advanced Techniques for Blazor Component will cover the components source code and learn more advanced components features. You will learn how to reference other components in code, how to preserve components, how to use components with a template, and how to define a CSS style dedicated to a specific component using CSS isolation.

Chapter 5: File Uploading in will cover the common file transfer protocols and compare the differences between them. Will learn the component used to upload files in Blazor Framework. This will explain the source code and detail usage with code examples.

Chapter 6: Serving and Securing Files in will explain one of the most important mechanisms in ASP.NET Core, middlewares. Middles work as pipelines handling the requests from clients. We will cover serving static files and dynamic files in Blazor framework, and a few basic security rules you will apply to protect servers from attacks.

Chapter 7: Collecting User Input with will cover web forms which are generally used when data input is required from application users. Will explain the default data validation implemented in the source code and how to customize validation rules and error prompts. Will cover some key events and concepts in Blazor forms, including submission, context, and state.

Chapter 8: Navigating Over will cover page navigations in a Blazor application. An enterprise level application usually needs multiple pages to fulfil a complete business requirement. It will also explain the key routing components in Blazor framework with source code and introduce different types of routing with parameters. And we will cover the navigation events and how to navigate in an asynchronous approach.

Chapter 9: .NET and JavaScript will cover serialization and deserialization with JSON, a common way to communicate between web services, and that applies to the interop between .NET and JavaScript as

well. Will explain how to load customized JavaScript code in a simple approach and in a more dynamic approach. Will cover calling JavaScript from .NET and the vice versa, with code examples. Will introduce some advanced topics related to .NET/JavaScript interop in Blazor, including cache, element reference and type safety.

Chapter 10: Connecting to the World with will cover the most famous HTTP protocol, and the separation of front-end and back-end services. HTTP protocol is mostly used between the front-end and back-end. Will cover the limits and risks come with the CORS when applications are connected using HTTP protocol. Will explain built-in types HttpClient and HttpClientFactory that will be used when communicating with the outside world with the source code. Will cover RPC and gRPC, an implementation of RPC from the Google with code examples.

Chapter 11: Data Persistence with EF will cover data persistence with EntityFramework Core and compare 2 key concepts, stateless and stateful. EntityFramework Core is popularly used in .NET Core project to store data in a selected database. Will explain the design ideas behind EntityFramework Core and analyze its source code to learn the patterns supporting different databases. Will cover key concepts in EntityFramework Core including entity, context, query, and migration with detailed examples.

Chapter 12: Protecting Your Application with will cover authentication and authorization in Blazor applications. Will explain the authentication mechanism in Blazor and learn the source code of AuthenticationStateProvider, which can be used to implement a customized authentication. Will cover different authorization approaches, including role-based and policy-based authorizations, with code examples.

Chapter 13: Deploying with Docker and will cover Blazor application deployments. One of the modern ways to deploy your applications is using Docker techniques and Kubernetes. Readers will learn how to containerize Blazor applications and deploy it with Azure Kubernetes Services and Azure Container Registry.

Code Bundle and Coloured Images

Please follow the link to download the

Code Bundle and the Coloured Images of the book:

https://rebrand.ly/i1gakbz

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a

general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Structure

Objectives

Razor components

Directive

Directive Attribute

One-way Binding

Binding Event

Binding Format

Unparsed value

Two-way binding

Cascading

Event Handling

Lifecycle

CHAPTER 1

WebAssembly Introduction

Introduction

In this chapter, we will introduce the concept and roadmap of WebAssembly and how it enables web applications to be written in any programming language. We will also discuss a few popular WebAssembly languages and illustrate the benefits of building a web application with ASP.NET Core Blazor.

Structure

In this chapter, we will discuss the following topics:

What is WebAssembly

How to compile a WebAssembly module

What does a WebAssembly module look like

.NET Core with WebAssembly

Objectives

This chapter is intended to guide you briefly through the world of WebAssembly, get familiar with WebAssembly modules and how .NET Core is involved with WebAssembly. We will learn how to install Emscripten SDK and will also get familiar with emcc command. We will explore the world of WebAssembly binary format and understand how a module was constructed. Finally, we will introduce the new generation of .NET --- .NET Core with the WebAssembly framework, ASP.NET Core Blazor.

What is WebAssembly

WebAssembly (abbreviated as Wasm) is a target for modern languages for compilation of more than one language, designed to be highly efficient while maintaining a safe sandbox environment. The definition seems to be too official. But if we break it down to two words Web and we might get a better understanding of WebAssembly. of course, everyone from a three-year-old kid to the elders nowadays know that they are living in the world of it. We can buy goods from watch videos on Youtube.com and check out how friends' life is on

Assembly, on the other way around, might not be that obvious to those who do not work with computer science. From recent years, most developers write programs with advanced programming languages like Golang, C# or But earlier, we did not have those advanced languages; programmers used to write code with Assembly, which is more specific to the hardware platform. For example, writing Assembly code for x86 CPU and ARM CPU will have different key words and syntax. As a language that is closer to the hardware level, Assembly language usually has higher runtime efficiency than advanced languages. Now, you might guess that WebAssembly is another assembly language running in the "web" - browsers.

History of WebAssembly

Early in 1990s, the first web was created. At that time, the web was mainly used by the scientists to share information. The web was designed to be the media of static content. HTML defines the content. URL locates the resources in the world of webs. The client (browser) would then send a HTTP request to the server through URL and then render the HTML content returned by the server. In this process, all the information transported was static, and that means there was no way that a user could interact with the web.

In 1995, Branden Eich designed a new language called JavaScript within only ten days. It looks like Java, but is easier to use than Java, and even non-professional website workers can understand it. However, Branden himself seemed to not like JavaScript that much. He was of the belief that everything that is excellent is not original, and everything original is not excellent. With the Chrome from Google getting more and more popular, JavaScript soon took place everywhere on the website. Even on the server nowadays. Engine V8 from Google is enabling JavaScript to be used in a large and complex project.

Hello World with WebAssembly

JavaScript has been good enough, then why do we bother creating another "Assembly Language" for the web? As far as we all know, JavaScript is a dynamic language, which means the type of a variable could be changed in runtime, unlike C# or Java. For programmers or developers, it is very convenient to write code, but it becomes cumbersome when it comes to the interpreter. The interpreter must judge of which type the variable is while running the code. Even armed with JIT compiler, compiling JavaScript into machine code ahead, sometimes, it must be rolled back to the original code under some circumstances. For this reason, many companies that build browsers are looking for a more performance enhanced solution.

In April 2015, WebAssembly Community Group was founded. Two years later, WebAssembly became one of the W3C standards. In 2019, WebAssembly became one of the standard web languages, along with HTML, CSS, and JavaScript. Through the years, most of the popular web browsers have supported WebAssembly.

Many languages, for example, C/C++, C#, Go can be compiled to WebAssembly now.

Let us take C/C++ as an example and write a simple C++ program that says Hello World. Save it as hello.cpp under

#include

```
int main() {


    printf("Hello World!\n");

}
```

Emscripten SDK is an open-source SDK that compiles C/C++ to WebAssembly, and auto-generates JavaScript code that can run the .wasm file. Install the SDK following the instructions here https://emscripten.org/docs/getting_started/downloads.html and compile the code with the following command:

```
emcc hello.cpp -o hello.html
```

Now, you will get three output files, hello.js and shown as follows:

```
my-hello-world-demo

├———hello.cpp

└———hello.html

└———hello.js

└———hello.wasm
```

hello.html is the default web page, and hello.js is the code logic running on it, designed by the Emscripten SDK.

Next, you will install Python from Now, open your command line or terminal and move to my-hello-world-demo and enter python -m python will start a server listening on port Open your browser and navigate to it will show the files under my-hello-world-demo, then click a default frontend page provided by emscripten will show. Refer to Figure

Figure Default Frontend Web Page

In the black box area, it shows Hello World! that we printed. A web that is actually running the C++ code. If we open the DevTools and switch to Console Tab, Hello World! is also printed there.

Call WebAssembly from JavaScript

Now, let us try something different, write a simple C++ function that can be called by the page using JavaScript. Create another file function.cpp and write the following code:

```
#include

extern "C"

{

   EMSCRIPTEN_KEEPALIVE

   int myAddFunc(int a, int b)

   {

      int c = a + b;

      return c;

   }

   EMSCRIPTEN_KEEPALIVE
```

```
    int myMinusFunc(int a, int b)

    {

        int c = a - b;

        return c;

    }


}
```

We have two functions here, myAddFunc will get the sum of two integers and myMinusFunc will get the subtraction. Similarly, compile with the command:

emcc function.cpp -o function.html

And the folder would look like:

my-hello-world-demo

├——function.cpp

└——function.html

└─────function.js

└─────function.wasm

└─────hello.cpp

└─────hello.html

└─────hello.js

└─────hello.wasm

We use python to start a server again and go to Nothing was printed in the black box area this time and that's because we did not print anything in the function! But we can call the two math functions provided by WebAssembly this time. Open the DevTools, switch to the Console tab and write _myAddFunc(1,2) and you will get 3 as the result. In fact, when you are typing _myAddFunc the IntelliSense will tell you that the function does exist in the context of page function.html. Try _myMinusFunc and it will work as well. How exactly the web page loads the two math functions we wrote here? Let us take a look at the generate function.html and

In the HTML body, it defines the frontend layout and page logic. We will focus on the script section. It first initiated a Module object, which has a few properties, for example, print, canvas, print shows Hello World! in the

previous code example on the web page by changing the value of the element with ID "output" and print it to the console as well with console.log(text);. setStatus is actually called when the page is first loaded and if you refresh the page a few times quickly, you will see a caption says And you might already guess it. It is downloading the WebAssembly file, Next, we will discuss how the function.wasm was loaded and how the function was called:

```
var asm = createWasm();

function createWasm() {

    function receiveInstance(instance, module) {

        var exports = instance.exports;

        Module['asm'] = exports;

    }

    function receiveInstantiationResult(result) {

        receiveInstance(result['instance']);

    }



    function instantiateArrayBuffer(receiver) {
```

```
        return getBinaryPromise().then(function (binary) {

            return WebAssembly.instantiate(binary, info);

        }).then(function (instance) {

            return instance;

        });

    }

    function instantiateAsync() {

        if (!wasmBinary && typeof WebAssembly.instantiateStreaming ==
'function' &&

            !isDataURI(wasmBinaryFile) && !isFileURI(wasmBinaryFile)
&& !ENVIRONMENT_IS_NODE &&

            typeof fetch == 'function') {

            return fetch(wasmBinaryFile, { credentials: 'same-origin'
}).then(function (response) {

                var result = WebAssembly.instantiateStreaming(response, info);
```

```
            return result.then(
                receiveInstantiationResult,
                function (reason) {

                    return instantiateArrayBuffer(receiveInstantiationResult);
                });
        });
    } else {
        return instantiateArrayBuffer(receiveInstantiationResult);
    }
}
if (Module['instantiateWasm']) {
    var exports = Module['instantiateWasm'](info, receiveInstance);
    return exports;
}
```

```
    instantiateAsync();

    return {};

}

function createExportWrapper(name, fixedasm) {

    return function () {


        var displayName = name;

        var asm = fixedasm;

        if (!fixedasm) {

            asm = Module['asm'];

        }

        if (!asm[name]) {

            assert(asm[name], 'exported native function `' + displayName + '`
not found');

        }
```

```
        return asm[name].apply(null, arguments);

    };

}

var _myAddFunc = Module["_myAddFunc"] =
createExportWrapper("myAddFunc");

var _myMinusFunc = Module["_myMinusFunc"] =
createExportWrapper("myMinusFunc");
```

Here is the key code of and it is fairly self-explained. A call to createWasm() starting the process. Inside this function, it goes to instantiateAsync() and we can guess from the function name that it will initiate the WebAssembly. And it does provide two ways to instantiate. If possible, it will fetch the wasm file through http protocol, in this case,

In this way, the wasm was loaded as a network stream, so WebAssembly.instantiateStreaming was used to process the http response, and if you open DevTools, switch to Network tab and refresh the page again, you will notice a request to http://localhost:8000/function.wasm and it returns Refer to Figure

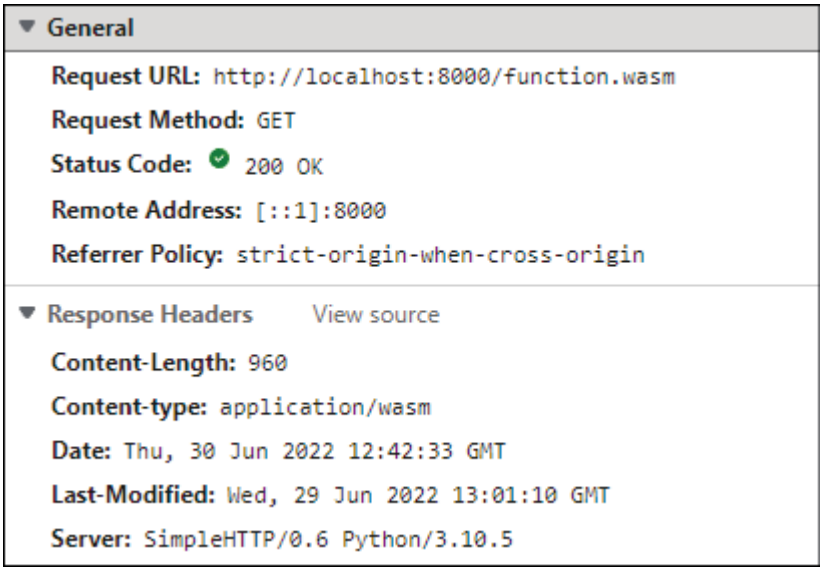

Figure fetch function.wasm

WebAssembly.instantiateStreaming() will be responsible for compiling and initiating the WebAssembly module, and it will be more efficient than load wasm code directly by In practice, most of the WebAssembly frameworks will choose WebAssembly.instantiateStreaming() to load the WebAssembly and this explains that some websites built by WebAssembly will be take longer to load for the first time than a website built with purely JavaScript, since they will download the .wasm file through network.

Otherwise, it will fall back to WebAssembly.instantiate() inside and the name of the function indicates that it is load the binary format of directly.

Once the WebAssembly module was loaded, receiveInstantiationResult() will be the callback to handle the instance of WebAssembly, and instance.exports will be assigned to Module['asm'] to save the exports from the WebAssembly. Finally, two lines of code generated by the

Emscripten SDK call and it will find the exported functions by name in Function apply will be used to run the desired function with arguments.

We can prove it by opening DevTools, switch to Console tab and type: _myAddFunc(2,3) and hit enter. As expected, the result is 5. Or we could use Module['asm'] directly: Module['asm']['myAddFunc'](4,5) and it shows 9 correctly. Great! Now, we know how the WebAssembly runs in the web, but what exactly is in the Can we manually load it?

Introducing .WASM binary format

Let us try another example.

```
#include

extern "C"

{

   EMSCRIPTEN_KEEPALIVE

   int myMultiplyFunc(int a, int b)



   {

      int c = a * b;
```

```
        return c;

    }

}
```

This time, we will compile it to .wasm only, without generating html and js file.

In the Terminal, type:

```
emcc manual.cpp -O3 –no-entry -o manual.wasm
```

Notice that –no-entry is required since we do not have a main() function and we will build in STANDALONE_WASM mode. And the folder would look like:

```
my-hello-world-demo

├———function.cpp

└———function.html

└———function.js

└———function.wasm

└———hello.cpp
```

└──hello.html

└──hello.js

└──hello.wasm

└──manual.cpp

└──manual.wasm

Open manual.wasm with a binary viewer or VS Code with appropriate extensions.

00000000 00 61 73 6d 01 00 00 00 01 17 05 60 00 01 7f 60 .asm……..`..`

00000010 00 00 60 02 7f 7f 01 7f 60 01 7f 00 60 01 7f 01 ..`..`..`..

00000020 7f 03 07 06 01 02 00 03 04 00 04 05 01 70 01 02 .....p..

00000030 02 05 06 01 01 80 02 80 02 06 09 01 7f 01 41 90 .....A.

00000040 88 c0 02 0b 07 80 01 08 06 6d 65 6d 6f 72 79 02 .....memory.

00000050 00 0e 6d 79 4d 75 6c 74 69 70 6c 79 46 75 6e 63 ..myMultiplyFunc

00000060 00 01 19 5f 5f 69 6e 64 69 72 65 63 74 5f 66 75
..__indirect_fu

00000070 6e 63 74 69 6f 6e 5f 74 61 62 6c 65 01 00 0b 5f
nction_table..._

00000080 69 6e 69 74 69 61 6c 69 7a 65 00 00 10 5f 5f 65
initialize...__e

00000090 72 72 6e 6f 5f 6c 6f 63 61 74 69 6f 6e 00 05 09
rrno_location...

000000a0 73 74 61 63 6b 53 61 76 65 00 02 0c 73 74 61 63
stackSave...stac

000000b0 6b 52 65 73 74 6f 72 65 00 03 0a 73 74 61 63 6b
kRestore...stack

000000c0 41 6c 6c 6f 63 00 04 09 07 01 00 41 01 0b 01 00
Alloc.....A....

000000d0 0a 30 06 03 00 01 0b 07 00 20 00 20 01 6c 0b 04 .0..... . .l..

000000e0 00 23 00 0b 06 00 20 00 24 00 0b 10 00 23 00 20 .#.... .$....#.

000000f0 00 6b 41 70 71 22 00 24 00 20 00 0b 05 00 41 80 .k"pq".$.
....A.

00000100 08 0b ..2

Refer to the following code consisting of the magic number:

00000000 00 61 73 6d 01 00 00 00 01 17 05 60 00 01 7f 60

^^ ^^ ^^ ^^

The first four bytes are what we called the magic numbers, 0x00 0x61 0x73 0x6d representing \0asm if you convert by ASCII code. It means that this is a .wasm file.

00000000 00 61 73 6d 01 00 00 00 01 17 05 60 00 01 7f 60

^^ ^^ ^^ ^^

The next four bytes are the version number, and we have 0x01 0x00 0x00 0x00 (little endian), version 1 here:

00000000 00 61 73 6d 01 00 00 00 01 17 05 60 00 01 7f 60

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

00000010 00 00 60 02 7f 7f 01 7f 60 01 7f 00 60 01 7f 01

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

00000020 7f 03 07 06 01 02 00 03 04 00 04 05 01 70 01 02

^^

The byte is the start of a section, which composes binary format of WebAssembly. The starting byte of a section represents the section type, and the next byte would be the length of the section. You may refer to the following table for more possible values.

Table 1.1 : WebAssembly Module sections

Sections usually start with 01 – type section, in this section, .wasm defines signatures of functions, or type information. In the first line of the first code discussed in this chapter, 0x01 flags the start of the type section, and the following byte 0x17 indicates that this section has length 23 (excluding type byte and length byte) till byte 0x20. The next byte 0x05 tells us that there are five functions, so we could find 5 0x60 following:

00000010: 00 01 7F 60 00 00 60 02 7F 7F 01 7F 60 01 7F 00

^^ ^^ ^^ ^^ ^^ ^^

Bytes 0x16 to 0x1b defines a function (0x60) with 2 (0x20) i32 (0x7F) parameters, and 1 (0x01) i32 (0x7F) output, and this would be our function that calculates the multiplication, and the function signature is exactly two integers (int a, int b) and an integer result (int c).

We don't have anything imported here, so there is no import section. And we jump to function section (0x03):

```
00000020 7f 03 07 06 01 02 00 03 04 00 04 05 01 70 01 02

         ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^
```

This section would be of length 7 (0x07). The next byte indicates there are 6 (0x06) function indices, and the array of indices is 1 (0x01), 2 (0x02), 0 (0x00), 3 (0x03), 4 (0x04), 0 (0x00):

```
00000020 7f 03 07 06 01 02 00 03 04 00 04 05 01 70 01 02
                           ^^ ^^ ^^ ^^ ^^ ^^

00000030 02 05 06 01 01 80 02 80 02 06 09 01 7f 01 41 90

      ^^

00000030 02 05 06 01 01 80 02 80 02 06 09 01 7f 01 41 90

         ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^


00000030 02 05 06 01 01 80 02 80 02 06 09 01 7f 01 41 90
                  ^^ ^^ ^^ ^^ ^^ ^^ ^^

00000040 88 c0 02 0b 07 80 01 08 06 6d 65 6d 6f 72 79 02
```

^^ ^^ ^^ ^^

The next is table section with id 4 (0x04) and length 5 (0x05), and it defines the JavaScript objects mapping, for they could not be accessed by WebAssembly directly and this helps to ensure that WebAssembly runs in a safe sandbox environment. Followed by section with id 5, (0x05) represents memory information, which defines the minimum and maximum memory usage. After that would be the global section (0x06) of length 9 (0x09), which defines 1 (0x01) global i32 (0x7f) variable and the variable is mutable (0x01). We are familiar with the global variables concept in many other advanced languages. This variable is initialized with the following instructions (0x41 0x90 0x88 0xc0 0x02 0x0b):

00000040 88 c0 02 0b 07 80 01 08 06 6d 65 6d 6f 72 79 02

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

00000050 00 0e 6d 79 4d 75 6c 74 69 70 6c 79 46 75 6e 63

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

00000060 00 01 19 5f 5f 69 6e 64 69 72 65 63 74 5f 66 75

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

00000070 6e 63 74 69 6f 6e 5f 74 61 62 6c 65 01 00 0b 5f

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

```
00000080 69 6e 69 74 69 61 6c 69 7a 65 00 00 10 5f 5f 65
         ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

00000090 72 72 6e 6f 5f 6c 6f 63 61 74 69 6f 6e 00 05 09
         ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

000000a0 73 74 61 63 6b 53 61 76 65 00 02 0c 73 74 61 63
         ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

000000b0 6b 52 65 73 74 6f 72 65 00 03 0a 73 74 61 63 6b
         ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

000000c0 41 6c 6c 6f 63 00 04 09 07 01 00 41 01 0b 01 00
         ^^ ^^ ^^ ^^ ^^ ^^ ^^
```

Now comes the export section (0x07) with length 129 (0x80 + 0x01). Contrary to the import section, it defines the functions, tables, memory, or global variables that will be exported to the world of JavaScript. In we have 8 (0x08) exported object. And if you look closer, you may notice that bytes from 0x51 to 0x61 defines the multiplication function. To help you find the function, we will demonstrate with the following code:

00000050 00 0e 6d 79 4d 75 6c 74 69 70 6c 79 46 75 6e 63

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

00000060 00 01 19 5f 5f 69 6e 64 69 72 65 63 74 5f 66 75

^^ ^^

The 5th (0x05) exported function is named 0x6d 0x79 0x4d 0x75 0x6c 0x74 0x69 0x70 0x6c 0x79 0x46 0x75 0x6e 0x63. And if you look them up in the ASCII table, you will find out that the function name is The last 2 bytes claims that the exported object type is function (0x00) and the index of the function is 1 (0x01):

000000c0 41 6c 6c 6f 63 00 04 09 07 01 00 41 01 0b 01 00

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

Since we compile the manual.cpp with the command –no-entry, we will skip the start section (0x08) and jump directly to the element section (0x09):

000000d0 0a 30 06 03 00 01 0b 07 00 20 00 20 01 6c 0b 04

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

000000e0 00 23 00 0b 06 00 20 00 24 00 0b 10 00 23 00 20

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

000000f0 00 6b 41 70 71 22 00 24 00 20 00 0b 05 00 41 80

^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^

00000100 08 0b

^^ ^^

The code section (0x0a) defines 6 (0x06) functions. The first one takes 3 (0x03) bytes (0x03), has no (0x00) parameters and the next 2 bytes would be the instructions code. You may guess, the next function takes 7 (0x07) bytes and no parameters as well.

Now you have a basic understanding of the WebAssembly binary format. Let's try to run manual.wasm manually in the browser.

Case With your favourite browser, open DevTools and switch to Console tab. Enter the following code all in once, and it will print out a log says "3 * 4 = 12"

var array1 =

var array2 = Uint8Array.from(array1);

```
WebAssembly.instantiate(array2).then(({instance}) => {

  console.log(`3 * 4 = ${ instance.exports.myMultiplyFunc(3, 4)}`);



});
```

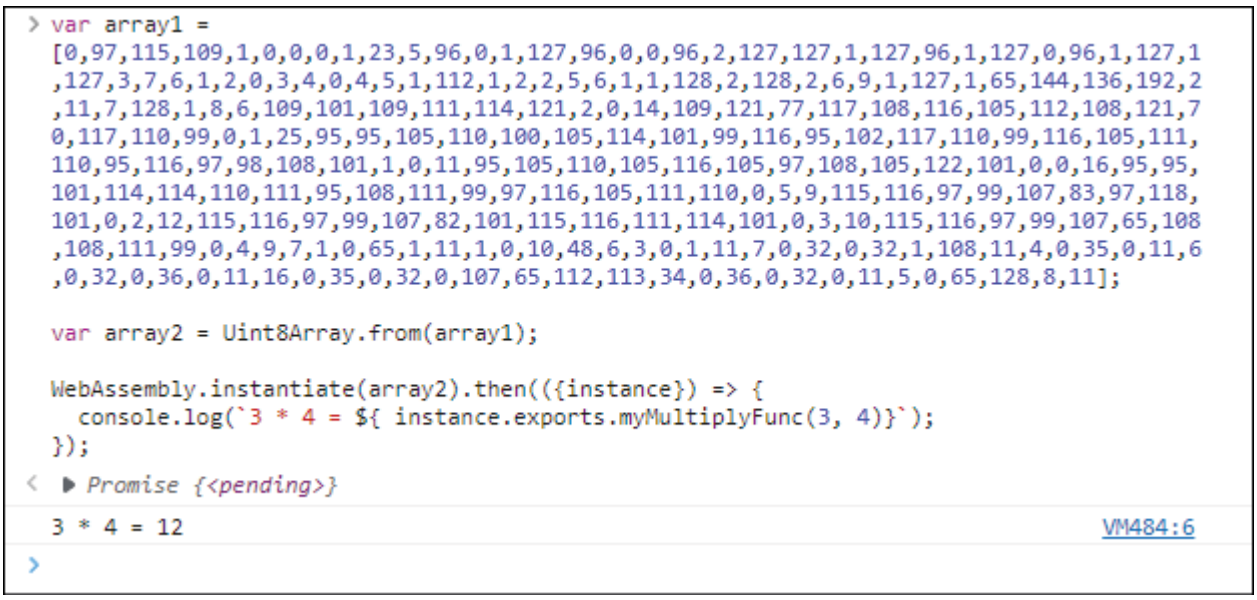

Figure Manually run WebAssembly

Case We will manipulate the manual.wasm a little, changing the function name from myMultiplyFunc to And then the new bytes would be:

00 61 73 6d 01 00 00 00 01 17 05 60 00 01 7f 60

00 00 60 02 7f 7f 01 7f 60 01 7f 00 60 01 7f 01

7f 03 07 06 01 02 00 03 04 00 04 05 01 70 01 02

02 05 06 01 01 80 02 80 02 06 09 01 7f 01 41 90

88 c0 02 0b 07 80 01 08 06 6d 65 6d 6f 72 79 02

00 0e 79 6f 75 72 4d 61 6e 69 70 75 6c 61 74 65

00 01 19 5f 5f 69 6e 64 69 72 65 63 74 5f 66 75

6e 63 74 69 6f 6e 5f 74 61 62 6c 65 01 00 0b 5f

69 6e 69 74 69 61 6c 69 7a 65 00 00 10 5f 5f 65

72 72 6e 6f 5f 6c 6f 63 61 74 69 6f 6e 00 05 09

73 74 61 63 6b 53 61 76 65 00 02 0c 73 74 61 63

6b 52 65 73 74 6f 72 65 00 03 0a 73 74 61 63 6b

41 6c 6c 6f 63 00 04 09 07 01 00 41 01 0b 01 00

0a 30 06 03 00 01 0b 07 00 20 00 20 01 6c 0b 04

00 23 00 0b 06 00 20 00 24 00 0b 10 00 23 00 20

00 6b 41 70 71 22 00 24 00 20 00 0b 05 00 41 80

08 0b

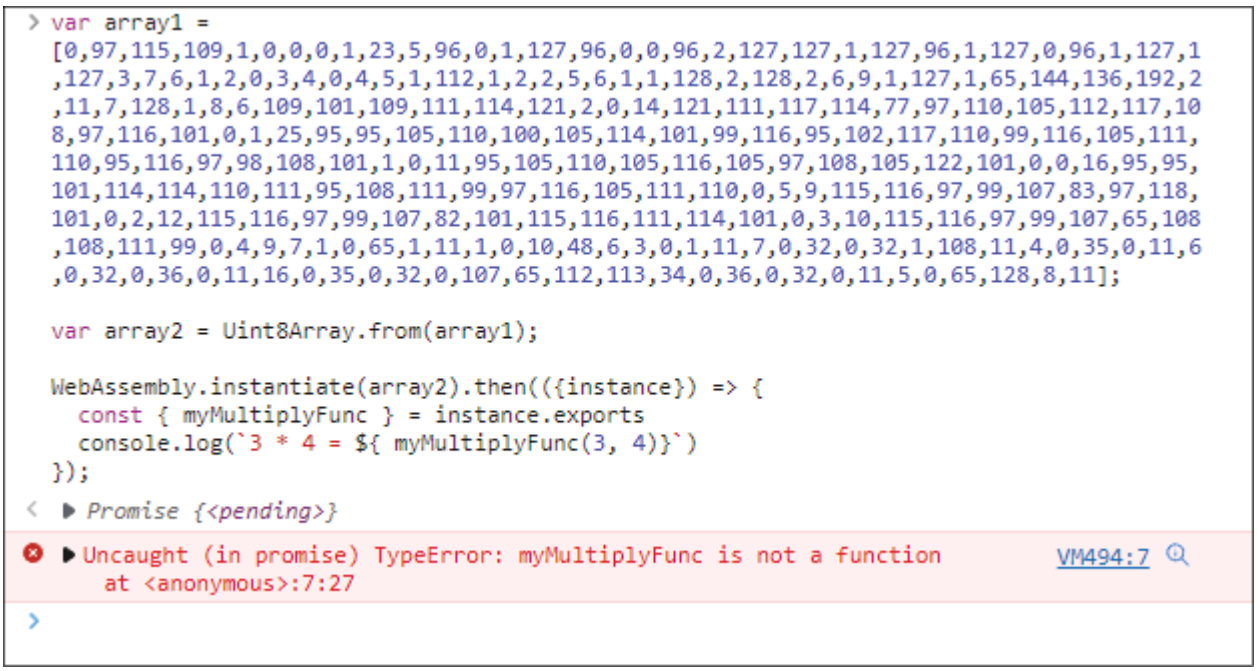

Figure Run manipulated WebAssembly

Obviously, it will complain that myMultiplyFunc is not a function, for we have changed it! You see, WebAssembly is like any other languages, and emscripten is just one of the tools that could convert exist code to WebAssembly. Compiled and instantiated by or WebAssembly.instantiateStreaming as a preferred way, WebAssembly runs in a safe sandbox in a more efficient and faster way.

WebAssembly in the future

The official is confident that everything that, if it is possible to be compiled targeting WebAssembly, eventually, will be compiled to WebAssembly. Now let us get back to the name --- WebAssembly, it is designed for the web initially and that is true, but it is not limited to the web, it could be running in the server as well. There is a possibility that one day in the future, all the computing unit will be running WebAssembly, regardless of the selected language, both for the client side and the server side.

Popular WebAssembly languages

Many popular languages now have been supporting; for example C/C++, Rust, C#, Golang and many more. Some of these supports are still under experiments period. On the other hand, some are pretty mature to be used in a production environment. In this book, we will choose C#, and if you have been working with C# for years, luckily, build a WebAssembly application will be a very smooth process for you. For those who never deal with C# before, don't worry, we will cover all the knowledge and concept you need to go through this book.

.NET Core

In the 2000s, .NET Framework was born. It is a reaction from Microsoft to the world of Java. The initial name was Next Generation Windows Services It is designed to be a new platform that could run not only one selected language, but also in a secure and extensible way, and to adapt the new world of web, which is born one or two decades earlier. With the framework of ASP.NET, programmers can develop websites by easily dragging and dropping web controls, and that greatly lowers the efforts required to build an enterprise level website. Since then, C# and .NET Framework has been evolving until 2015. In Build 2015, Microsoft has pronounced a new .NET platform called .NET Core, which is available to run on every platform including Windows, Linux and and the new world of C# begins. C# is not closely coupled with Windows Server anymore. It could be run in a docker container, more cloud native, and obviously, more adapted to the new world of web, again.

ASP.NET Core

ASP.NET Core was born with the new .NET Core platform, which is aimed to upgrade from the old ASP.NET. By providing a default and replaceable dependency injection container, and a newly designed middleware mechanism, it has quickly become a popular framework for the microservice architecture. Basically, it is another normal .NET Core console application, except that when it starts, it will run a few configurations and listens for the http requests.

When to choose ASP.NET Core Blazor

ASP.NET Core Blazor was the framework that is aiming to build a client web application running both on WebAssembly, Server or even native apps, while enjoying the benefits of your existing C# ecosystem, for example, your entity objects could be shared between client app and backend services running on Linux servers, enjoying also the secure and efficiency brought from WebAssembly, enjoying the interactions with your application build upon C# instead of JavaScript and enjoying the web standards that allows your application to be ran on any modern browsers, even on a mobile device.

Blazor has been one of the top WebAssembly enthusiasts. There are many active open sources projects that are contributing to the communities, providing many UI Components and library available on your hand. And this book will cover every corner you need to build your production-ready Blazor application.

Conclusion

WebAssembly is crafted to be a target for programming languages compiler to be target with. It is now one of the W3C standards with HTML, CSS and making it available to run on the most modern web browsers like Chrome, Edge, Safari, and not only on PC, but on mobile devices as well.

WebAssembly is safe to be executed in a sandbox environment, while maintaining efficiency and speed. It could be executed at a near native speed. It is open and debuggable with help from its textual format. And remember, despite having started with "Web", WebAssembly is not limited to build web applications, it is possible to run WebAssembly with backend services as well.

In the next chapter, you will be introduced to WebSocket, another communication protocol between clients and servers other than http, and SignalR, a .NET Core library that helps you build a WebSocket server easily. We will also discuss three major blazor hosting models. Blazor Server executes in the server based on the SignalR WebSocket connection, while Blazor WebAssembly, as its name suggests, leverages WebAssembly to run in the client environment. Blazor Hybrid will be discussed which is a new model that combines blazor with native application development so that you could share your code between Blazor Server or Blazor WebAssembly and applications run in desktops or mobile devices.

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2

Choose Your Hosting Model

Introduction

In the last chapter, we briefly introduced ASP.NET Core Blazor. We got to know that Blazor is becoming one of the most popular frameworks for frontend developing. In this chapter, we will continue exploring and comparing three different Blazor hosting models. One model, Blazor WebAssembly, that you may guess from its name, is a framework dedicated to WebAssembly. Let us deep dive into the different Blazor hosting models.

Structure

In this chapter, we will discuss following topics:

Why WebSocket when we have HTTP

Using SignalR to create a WebSocket server

Blazor Server

Blazor WebAssembly

Blazor Hybrid

Objectives

This chapter aims to introduce one of the most important protocols that will be used in Blazor, WebSocket, and the three common hosting models provided by Blazor.

Before going into the world of hosting models, let us introduce a few concepts that would be useful.

WebSocket

When we open a web page, for example, the browser actually sends a GET HTTP request to one of Google's servers, and the server responds with HTML, CSS or JavaScript that are required to render the page. Before the existence of WebSocket, all the requests are imitated by the client, or web browser. The server is not able to send a message to the client initially. How could a web application, for example, a web chatting room allows you to chat with your friends remotely, get the notification that your friend sent you a message? The following image depicts the protocol between clients and servers in the most natural way.

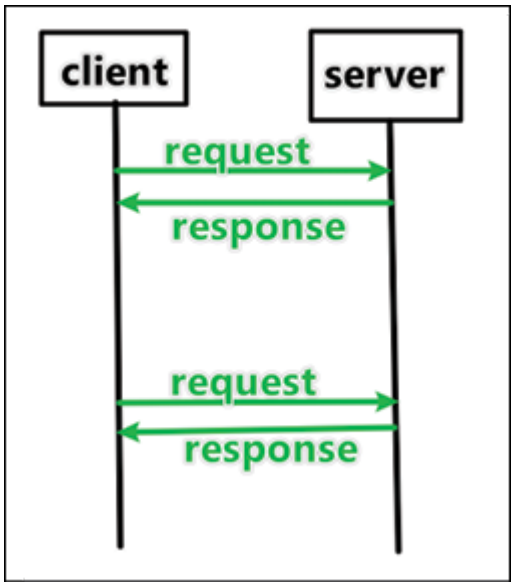

Figure HTTP Model

An easy way to implement this is that we could loop sending check requests to the server with Ajax. The shorter the loop, the more "instantly" you get your friends' messages. Imagine an app with hundreds of thousands of users, each having hundreds of friends, millions of check requests will be sent to the app servers, and that is only the requests to check if there are new message yet. No reasonable servers could handle such a large quantity of requests.

A request sent from the user's browser to the server, goes through the internet by the HTTP protocol based on a TCP connection. After the server responds to the request, it may or may not close the TCP connection. If it decides to close the connection, the next time the client is sending the request, a new TCP connection must be built again. It is called a Request/Response

WebSocket is not entirely independent on the HTTP protocol. The client first sends a hand shake request to the server based on the HTTP protocol, with pre-defined standard headers, including WebSocket Key which is used to identify the client and the server, version and sub-protocol that is aligned between the client and the server so that they will both communicate based on the aligned sub-protocol. If the server is compatible with WebSocket, it will respond to the client with pre-defined headers as well. The client will not communicate with the server again until it validates the response from the server, and a WebSocket connection is built upon. With this WebSocket connection, the client and the server can talk in a bi-directional way. They both could actively send message to the other as shown in the following figure:

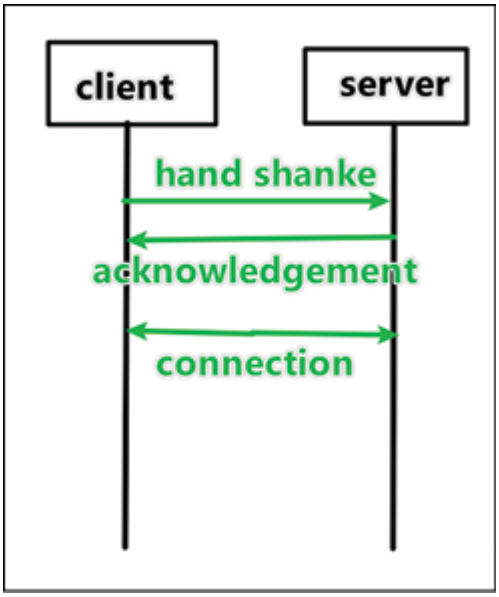

Figure WebSocket model

Both HTTP and WebSocket protocols are based on TCP, so in general, it would be easy for the server to support WebSocket, and the default ports are also 80 and 443 for the WebSocket. WebSocket soon became popular in online games, chat rooms, and many other applications which are dependent on real-time or live interactions between users and the platforms.

In general, almost all the real-time services are exchanging information on the WebSocket connection, while the stateless HTTP protocol is more popular with RESTful API services. A HTTP connection can be setup directly, while the WebSocket must have a hand shake request first before establishing the connection. HTTP protocol contains more overheads in each transport than WebSocket. In addition, WebSocket supports sending binary messages.

SignalR

WebSocket is apparently more popular in real-time applications. However, implementing a WebSocket from scratch might be both complex and time-consuming. In addition, some old version browsers might not have WebSocket supported. You do not know how your server will respond to these customers when you cannot force them to upgrade, they just cannot upgrade due to irregularity or other reasons. It would just be more complicated if your server tries to switch between long-polling and WebSocket for different clients. It would be really great if there is a library that can both help you build a real-time server based on WebSocket while also being compatible with older browsers where long-polling is the only solution to be real-timing.

Voilà! An open-source library emerges, and that is ASP.NET Core SignalR library is an enabler that your application could leverage to send messages in both directions, from the client to the server, invoking code on the client side through remote procedure calls or the vice versa, and most importantly, if your application is facing customers with older browsers, SignalR automatically falls back to long-polling for you if conditions were not met, and of course, you could customize the fallback priorities. While sending messages simultaneously to all the clients, you could also send a message to a specific client. Sounds familiar? Nowadays, a typical instant chatting app would require those capabilities to create chat rooms and private chats. SignalR is not only saving your time on building a WebSocket server by yourself like many popular WebSocket libraries for .NET or other languages, but also provides an

automatic fallback mechanism that saves you and your applications from worrying about compatibilities.

Enough of talking, let us create a demo and see how easy it is to set up an instant notification receiving boards with SignalR.

cd to a fold of your desire for this demo:

dotnet new webapi -o NotificationApp

code .\NotificationApp\

The command above creates a new ASP.NET Core Web API project and opens the project by VS Code. Now, we will add the SignalR capabilities to it.

cd .\NotificationApp\

Or open the Terminal window and it will be automatically located in the project directory.

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();

This is generated from the ASP.NET Core Web template and it can barely do anything other than showing "Hello To provide a web page UI interface to users, we create a folder called and add a file

html>

lang="en">

Notification App

This is an application to send/receive centralized notifications.

To serve this html page, ASP.NET Core provides two methods: UseDefaultFiles() and The first method enables the default file path mapping, and that is where we put the The second method enables the server to host the static files, and index.html is one of them.

Now, the Program.cs looks like below, and we comment out the otherwise it will show "Hello World!" by default.

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();


app.UseDefaultFiles();

app.UseStaticFiles();
```

```
// app.MapGet("/", () => "Hello World!");

app.Run();
```

Type dotnet run in your command line and it will say that the server is listening on a certain port, and navigate to that URL in your web browser, in this case it is which will render the HTML page.

Next, we will use SignalR to implement a WebSocket server that could push notifications to clients.

First, we register SignalR into the container. A container in ASP.NET Core is a box, or a provider that you could register multiple interfaces, classes, or instances, and retrieve back later. It is an implementation of the design pattern, inverse of control. The idea behind is that you are not depending on a concrete implement layer anymore. Instead, you rely on a contract or an interface that promises to fulfil your requirements. Basically, we have two ways, inject and retrieve, to interact with the container, or ASP.NET Core calls it ServiceProvider.

```
    ///

    /// Adds SignalR services to the specified cref="IServiceCollection" />.

    ///



    /// name="services">The cref="IServiceCollection" /> to add services to.
```

```
    /// An cref="ISignalRServerBuilder"/> that can be used to further
configure the SignalR services.

    public static ISignalRServerBuilder AddSignalR(this
IServiceCollection services)

    {

        if (services == null)

        {

            throw new ArgumentNullException(nameof(services));

        }

        services.AddConnections();

        // Disable the WebSocket keep alive since SignalR has it's own

        services.Configure(o => o.KeepAliveInterval = TimeSpan.Zero);

        services.TryAddSingleton();

                    services.TryAddEnumerable(ServiceDescriptor.Singleton,
HubOptionsSetup>());
```

```
        return services.AddSignalRCore();

    }
```

The static method like many other dependency injection extension methods, put the required dependencies into the container, for example, Generally, these registrations are structured under the namespace even when you are creating your own libraries, for example, NLog takes this pattern to register itself as well.

Now, we add SignalR to our notification application by simply calling

```
var builder = WebApplication.CreateBuilder(args);

// Add SignalR

builder.Services.AddSignalR();

var app = builder.Build();
```

IServiceProvider plays an important role in dependency injection in ASP.NET Core. As the name of this interface suggests, it is a provider for all kinds of services that are registered into the container:

```
namespace System

{
```

```
    //


    // Summary:

    //     Defines a mechanism for retrieving a service object; that is, an
object that

    //     provides custom support to other objects.

    public interface IServiceProvider

    {

        //

        // Summary:

        //     Gets the service object of the specified type.

        //

        // Parameters:

        //   serviceType:

        //     An object that specifies the type of service object to get.
```

```
        //

        // Returns:

        //     A service object of type serviceType. -or- null if there is no
service object

        //     of type serviceType.



        object? GetService(Type serviceType);

    }

}
```

To take the dependencies out of the box, .NET defines a method GetService in the interface It takes the Type required as a parameter and will return the instance of that Type if possible.

Second, let us create a folder Hubs under root path and add a new file

```
using System.Collections.Concurrent;

using Microsoft.AspNetCore.Http.Features;

using Microsoft.AspNetCore.SignalR;
```

```
public class NotificationHub : Hub

{

   private static ConcurrentDictionarystring> Connections = new
ConcurrentDictionarystring>();

   public async Task Notify(string notification)

   {

      var caller = Clients.Caller;



      await Clients.All.SendAsync("onReceived", $"
{DateTime.Now.ToShortTimeString()}:
{Connections[Context.ConnectionId]} notifies: {notification}");

   }

   public override async Task OnConnectedAsync()

   {

      var feature = Context.Features.Get();

      Connections[Context.ConnectionId] = $"
{feature.RemoteIpAddress}: {feature.RemotePort}";
```

```
        await base.OnConnectedAsync();

        await Clients.All.SendAsync("onReceived", $"
{DateTime.Now.ToShortTimeString()}:
{Connections[Context.ConnectionId]} joins.");

    }

}
```

A hub in SignalR works as a communication manager. It defines the methods that will be called by the clients, and calls the methods defined by the clients. All the dirty work beyond that is handled by the SignalR library as we introduced before. In this NotificationHub we define a method Notify with a string parameter, and client will invoke this method to send a notification to our centralized notification manager. Meanwhile, the hub will invoke the function onReceived on the client side with one parameter as well. Notice that we override OnConnectionedAsync from the base Hub class to log the connected clients, so that we know where the notification is coming from.

One more step on the server side, mapping our hub to the asp.net core middleware pipelines by calling MapHub in

```
app.UseStaticFiles();

app.MapHub("/Notification");
```

We have completed the setup on the server side, let us turn to the client side.

microsoft-signalr - Libraries - cdnjs - The #1 free and open source CDN built to make life easier for developers

First, we update the index.html to import the SignalR JavaScript client code and our customized signalr.min.js enable us to build a connection with our SignalR server and send message back and forth. To allow users notify all the clients, we add a text input element with id txt and a button input with id Finally ad a bullets element to show all the notifications we will receive.

html>

lang="en">

Notification App

This is an application to send/receive centralized notifications.

type="text" id="txt" />

type="button" id="btn" value="Notify"
/>

/>

id="notifications">

In we build a connection with URL defined in ASP.NET Core pipeline. And in case the connection is break out, we restart the connection again in the onclose handler. When a user click the Notify button, we will invoke the method Notify defined in the which will in turn invoke the onReceived handler and append a new item in the bullets list.

```
const connection = new signalR.HubConnectionBuilder()

    .withUrl('/Notification')

    .configureLogging(signalR.LogLevel.Information)

    .build();

async function start() {

    try {

        await connection.start();

        console.log('SignalR Connected.');

    } catch (err) {
```

```
        console.log(err);

        setTimeout(start, 5000);

    }


};

// Listen for `DOMContentLoaded` event

document.addEventListener('DOMContentLoaded', (e) => {

    document.getElementById('btn').addEventListener('click', send);

});

async function send(e) {

    const message = document.getElementById('txt').value;

    console.log(message);

    try {

        await connection.invoke('Notify', message);

    } catch (err) {
```

```
        console.error(err);

    }

}

connection.on('onReceived', (message) => {

    const li = document.createElement('li');

    li.textContent = message;

    document.getElementById('notifications').appendChild(li);


});

connection.onclose(async() => {

    await start();

});

// Start the connection.

start();
```

Now type dotnet run in the terminal to run the app. This time let's bring up two browser clients and navigate to In the first browser window, you will see two clients joining messages. Try send notifications in these two browsers and the will both get notified. Refer to Figure

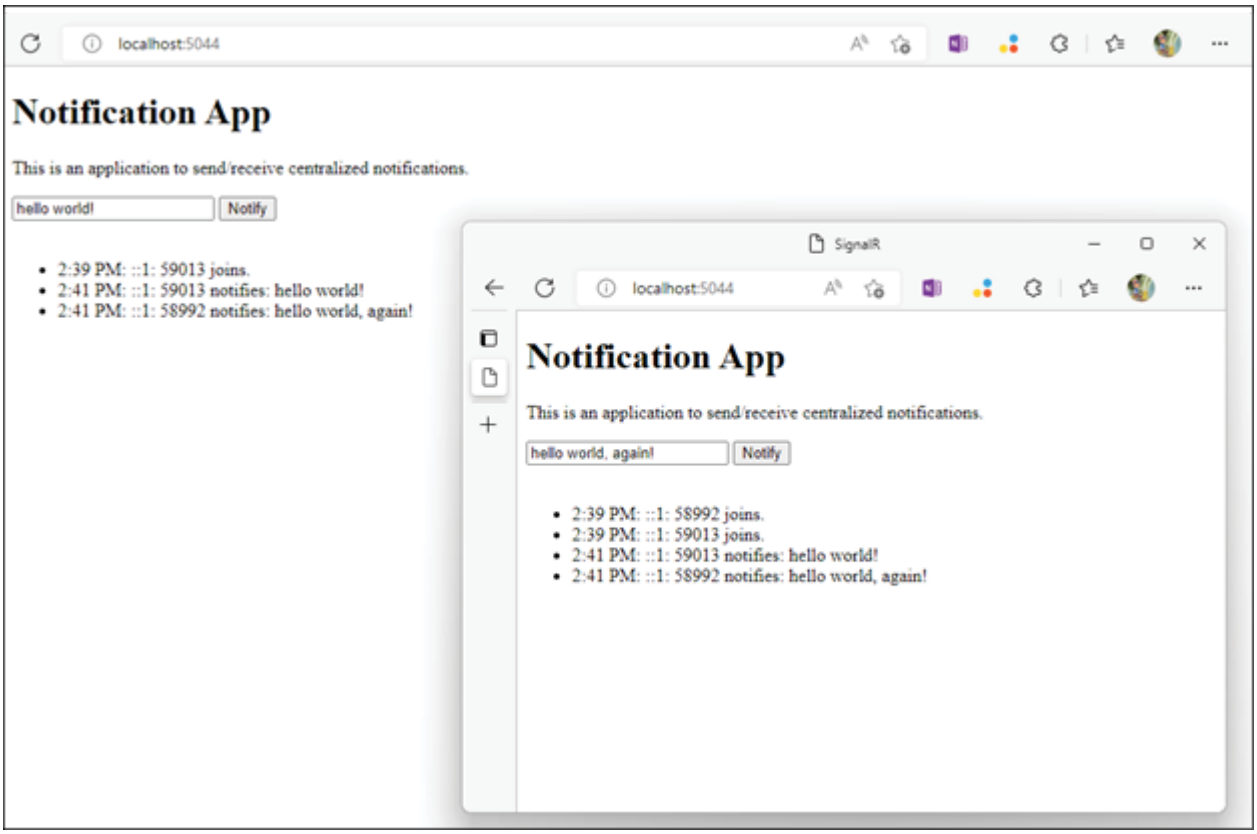

Figure Notification App

We are using Microsoft Edge Version 104.0.1293.46 on Windows 11 and it supports WebSocket, so SignalR will utilize WebSocket and RPC to implement these notifications. While in your case, if you are using an older browser that does not support WebSocket, you will notice that you are able to get notifications as well since SignalR can fallback communication strategies automatically.

Blazor Server

All different blazor hosting models leveraged Razor components (We will explain more in Chapter Implementing Razor and Other And where the components were hosted determines the hosting model. If the components run on the server side, we call it Blazor Server. When we say components run on the server side, it means that the app is executed in the ASP.NET Core application, And Blazor Server heavily depends on SignalR to update UI, handle event and execute JavaScript.

Let's create a Blazor Server app and see how SignalR helps here. Typing the commands in your terminal to generate a new Blazor Server app in the folder BlazorServerDemo

```
dotnet new blazorserver -o BlazorServerDemo
```

```
cd ./BlazorServerDemo
```

In you may find that one line code looks familiar, which we introduced in the SignalR section.

```
using Microsoft.AspNetCore.Components;
```

```
using Microsoft.AspNetCore.Components.Web;
```

```
using BlazorServerDemo.Data;
```

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

Builder.Services.AddRazorPages();



builder.Services.AddServerSideBlazor();

builder.Services.AddSingleton();

var app = builder.Build();

// Configure the HTTP request pipeline.

If (!app.Environment.IsDevelopment())

{

    app.UseExceptionHandler("/Error");

    // The default HSTS value is 30 days. You may want to change this for production scenarios, see

    App.UseHsts();

}
```

```
app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.MapBlazorHub();

app.MapFallbackToPage("/_Host");

app.Run();
```

It is reasonable to guess that MapBlazorHub will map an endpoint for the BlazorHub class, just like what we did in our previous SignalR notification application.

aspnetcore/ComponentEndpointRouteBuilderExtensions.cs at main · dotnet/aspnetcore

Check the open-source code on GitHub, and our assumption is verified. MapBlazorHub has a few overloads that will finally map the class ComponentHub to the default URL And it can be easily proved if you start the BlazorServerDemo application, browse to the home page and switch to the Console tab in the developer tools. The console log says Information: Normalizing '_blazor' to Another evidence is that when we register blazor into the container, the open-source code shows that it is

calling the method AddSignalR under the hood to register SignalR as well.

It is worth pointing out that while the application is running on the server, a circuit, represented the connection between a client and the server is, persisted until disconnection. Each connection established with the server will open a new circuit, and that includes when you open a new browser to the same page. The circuit will be closed when user closes a page or navigates to the external URL. And even better, you can inherit the CircuitHandler to customize the lifecycle handlers for the circuits. By taking notes of each connected client like we did in the SignalR example, you can show how many clients are reading your blogs or watching videos.

You usually choose Blazor Server if you want to take advantage of the .NET ecosystem, for example, .NET API, libraries, or tool chains. In addition, with Blazor Server you release the clients from downloading heavily as compared to Blazor WebAssembly. On the other side, since the application is running on the server, all the client interactions will be handled by your server as well, and that would definitely hurt your server's performance, and you can't even leverage CDN to boost static resource loading.

Blazor WebAssembly

Blazor WebAssembly on the other hand, is running in the web browser on the client side. It is based on a .NET WebAssembly runtime, and it will be downloaded by the browser for the very first visit. Unlike Blazor Server, all the events, or UI interactions are running on the client threads.

Let's take the default Blazor WebAssembly app as an example. We create a demo app with the following commands:

dotnet new blazorwasm -o BlazorWebAssemblyDemo

Then in the folder files similar to Figure will be generated:

Figure Code generated by dotnet new

Then run another command in BlazorWebAssemblyDemo to run the Balzor WebAssembly application:

dotnet run

Browse to the URL displayed in your Terminal and open up the network tab in DevTools, and you should see many dotnet libraries are being sent to the client as shown in Figure

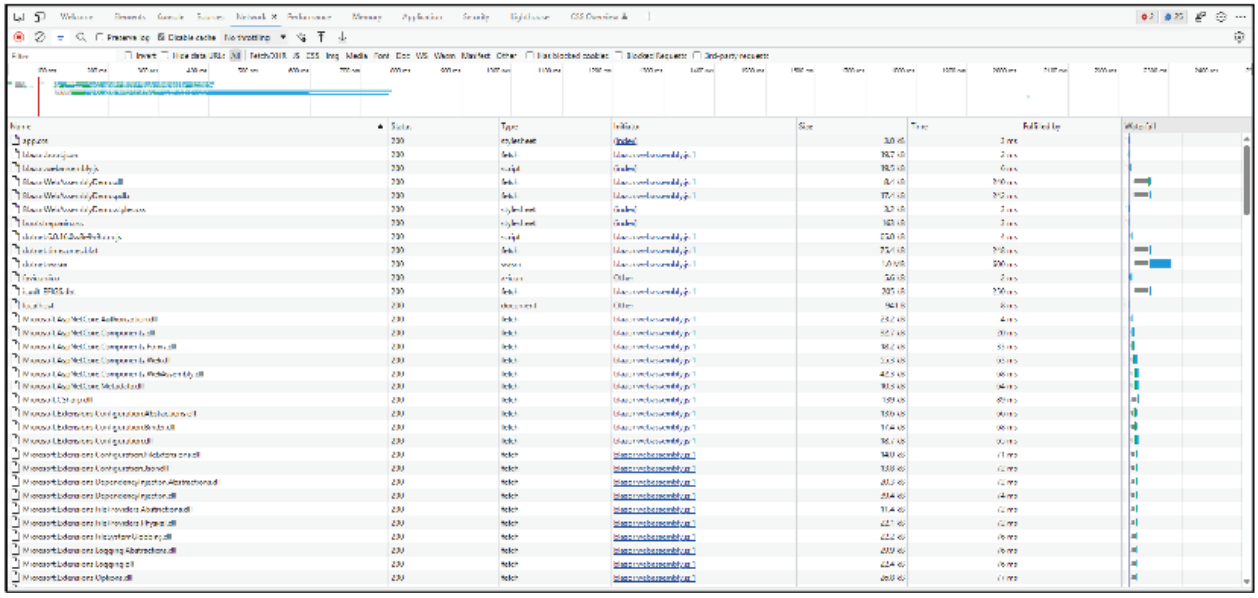

Figure dotnet libraries sent to clients

Some of these are exactly the same libraries that are normally required when you run a local dotnet console application, such as System.IO.dll, while others are dedicated for a Blazor application, such as Microsoft.AspNetCore.Components.WebAssembly.dll. When you run a local console application, those dll libraries are consumed by a .NET runtime. Similarly, Blazor WebAssembly provides such a runtime, sent to the client as well in Figure and this runtime makes it possible to consume unchanged dotnet libraries just as in a console application.

As we introduced in the last chapter, the wasm code is initiated by In this JavaScript file, it wraps a method to call the recommended WebAssembly.instantiateStreaming and falls back to array instantiation.

Here, we've picked up the core code to help you understand the process so that you can build a better connection with the WebAssembly we introduced in the last chapter:

```
await async function (e, t) {

    if ("function" == typeof WebAssembly.instantiateStreaming)

        try {

            return (await WebAssembly.instantiateStreaming(e.response,
t)).instance

        } catch (e) {

            console.info("Streaming compilation failed. Falling back to
ArrayBuffer instantiation. ", e)

        }

    const n = await e.response.then((e => e.arrayBuffer()));

    return (await WebAssembly.instantiate(n, t)).instance

}(t, e)
```

Other than the WebAssembly loading, let's take a closer look at the starting point of a Blazor WebAssembly application,

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
```

builder.RootComponents.Add("#app");

First, it creates a and it is responsible for configuring and creating a The next line of code configures the RootComponent for the application, which is an App component, followed by a parameter of CSS selector. It will select a HTML element with id app as in index.html under the wwwroot folder:

html>

lang="en">

id="app">Loading...

id="blazor-error-ui">

An unhandled error has occurred.

href="" class="reload">Reload

This index.html works as a blueprint for the Blazor WebAssembly application. It works the same way as any other web application. In the body section, we defined a div element with id And the component added to the RootComponents in builder will be inserted here.

In it defines two sections for found route and not found route, and intuitively, they are for a defined route in your application and what a 404 response should look like. We will discuss more about the Router component in a later chapter. And in the project root folder, there is another file, It defines global @using directive. Any directive added to this file applies to all the components in the project folder.

You usually choose Blazor WebAssembly when you desire to have an application that can work offline when there are no network connections. You can also benefit from CDN distributing static resources, or even as a whole, serving your applications. Blazor WebAssembly also helps to reduce the server performance pressure since all the event handling, UI interactions and heavy calculations are now the responsibility of the clients. In addition, the applications could be installed as a progressive web app on the clients' machines, and you can leverage all the capabilities of PWA, for example, notification.

Blazor Hybrid

Blazor Hybrid has joined the blazor family recently. It is a hybrid way to build native applications with HTML and CSS technologies. You will unlock all the capabilities which are not available on the web-alone platform, combined with .NET MAUI, WPF, and Windows Forms. We will not cover Blazor Hybrid in the following chapters. But you are encouraged to experiment with it in your own way, especially when you have a production ready web applications that is built with blazor. It will greatly reduce your workload when you are developing a corresponding native app.

Conclusion

In this chapter, we begin with a protocol WebSocket. It is proposed to help build a full duplex communication channel between the client and server. It is very popular in real time web applications, for example, instant messaging and gaming. Then we introduced a .NET library, SignalR. It is aimed to help you build real time applications with high performances. We continue this chapter by developing a demon on how clients can send notifications to each other online.

Blazor Server is heavily dependent on SignalR to build connections with clients, and all the handlers are running on the server side, while Blazor WebAssembly offloads this work to the browsers. We understand the key benefits of these two models. Blazor Hybrid is another hybrid way to develop native applications based on web technologies, and it will overall increase your productivity if you have a web based on blazor already, or you want to develop an app targeting multiple platforms, including web, desktop, and mobile at the same time.

In the next chapter, we will introduce the core building block in the world of blazor --- Razor Component, and dive deep into how components are implemented and how they can be combined to render UI, partial pages, and layouts.

Join our book's Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3

Implementing Razor and Other Components

Introduction

In this chapter, we will introduce Razor components and discuss one of the special components, layout, which shares many common properties with other components as well. We will jump into the detail of how a component is implemented and how to use them on a page. In addition, the community has been providing plenty of open-source libraries, and we will take a look at them.

Structure

In this chapter, we will discuss the following topics:

Directives

One-way binding

Two-way binding

Cascading

Event handling

Lifecycle

Layout

Libraries

Objectives

In this chapter, we will understand the process of creating a customized component and understand the binding between components and models to create an interactive application.

We will then understand the event handling for components and know the lifecycle of components to customize components' behavior. You will also learn how to reuse components by layouts.

Starting from this chapter, we will build an e-shop web application step by step, with the topics we discussed in each chapter, and eventually deploy it on the cloud.

Regardless of which Blazor model you choose, Razor components will be the common building blocks used to build your application ground up.

Razor components

Components are usually developed by the file with the extension It will be your bricks when building the application house. Razor syntax is based on both HTML and C#, while HTML defines how to render the components and on the other hand, C# defines the logic of the components. Let's take one of the simplest razor components as an example.

Before creating our first components, a blazor project is created by typing dotnet new blazorwasm in your favorite Terminal. We'll choose Microsoft Terminal here. Run the default application with dotnet run, and browse to the webpage with port, in my case, it is https://localhost:7045 and you will see a web page like shown in Figure

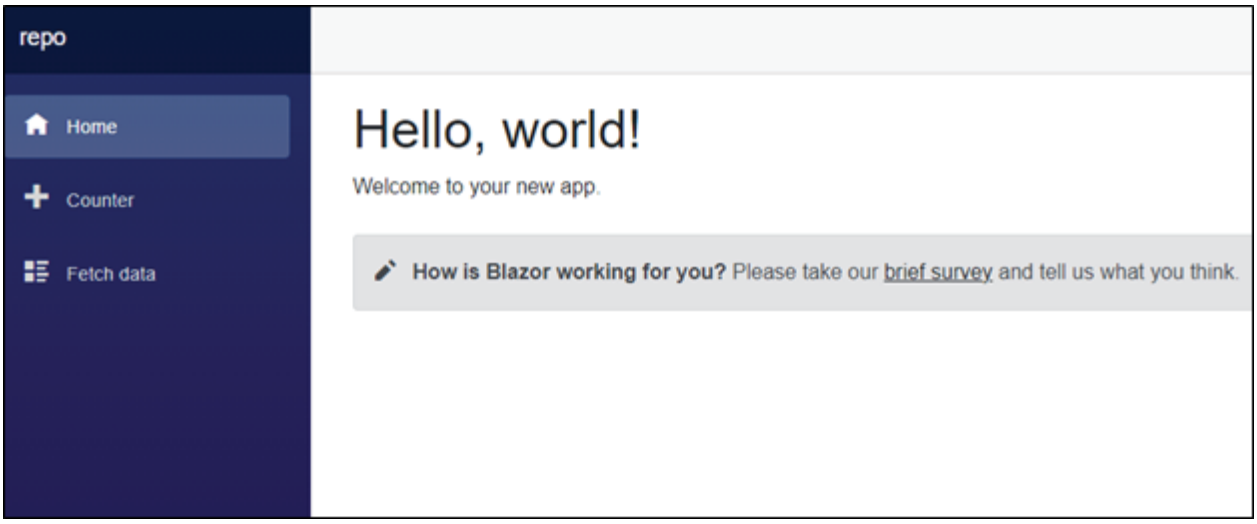

Figure Blazor WebAssembly home page

We will create a simple head component to replace Hello, world! with This is my new head! First, we go to the shared folder under the root and create a new file called

@head

@code {

    private string head = "This is my new head!";

}

Update the Index.razor under Pages folder to use the newly created component:

@page "/"

Index

@*

Hello, world!

*@

/>

Welcome to your new app.

Title="How is Blazor working for you?" />

We commented out the old h1 label and replaced with our newly created components. Now run the app again with dotnet run and refresh the browser. You will notice that, on the home page, we have a header that says, This is my new

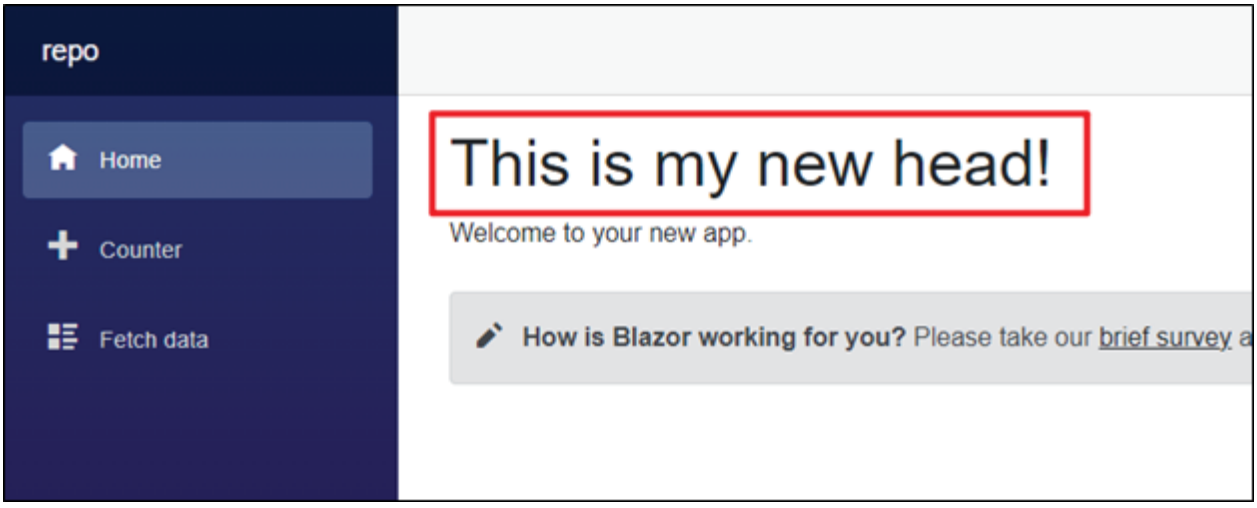

Figure Home page with the new header

In razor component, we extensively use @ sign prefixed to a keyword representing a directive or a directive attribute.

Directive

A directive changes how a component is parsed. Take our simple head component MyHead as an example. Inside the h1 label, we have a @ sign followed by the variable head. If we remove the @ sign, you will see literally "head" in the home page.

In this section let's build an order model that users could update the count of items in the cart.

First, we create a shop item model, and an order model, which contains key information, for example, which item your customers are buying and the numbers of these items:

```
namespace EShop.Models

{

   public class ShopItem

   {

      public string Name { get; set; }

      public string Description { get; set; }
```

```
        public double Price { get; set; }

        public ShopItem(string name, string description, double price)

        {

            Name = name;



            Description = description;

            Price = price;

        }

    }

}
```

The next model is It represents an item in the customers' shopping cart, including how many of this specific item:

```
namespace EShop.Models

{

    public class CartItem
```

```
    {
        public ShopItem Item { get; set; }
        public int Count { get; set; }
        public CartItem(ShopItem item, int count)
        {
            Item = item;
            Count = count;
        }
    }

}
```

A final model is the A customer can manage to add all kinds of CartItem in the

```
namespace EShop.Models
{
```

```
public class Cart

{

    public List Items { get; set; } = new List();

    public void Add(CartItem item)

    {

        Items.Add(item);

    }

    public void Remove(CartItem item)

    {

        Items.Remove(item);

    }

    public void Clear()

    {
```

```
            Items.Clear();

        }

    }

}
```

Next, we will create a Cart.razor page in the Pages folder to show the CartItems customer added to the

```
@page "/cart"

Cart
```

Cart

class="btn btn-danger" @onclick="Buy">Buy

class="list-group">

@foreach (var item in _cart.Items)

{

class="list-group-item">

class="d-flex w-100 justify-content-between">

class="mb-1">@item.Item.Name

3 days ago

class="mb-1">@item.Item.Description

class="d-flex flex-row mb-3">

class="p-2 bi bi-dash-square" @onclick="()=>Remove(item)">

class="p-2">@item.Count

class="p-2 bi bi-plus-square" @onclick="()=>Add(item)">

}

@code {

```
    private Models.Cart _cart = new Models.Cart();

    private void Buy()

    {

        _cart.Add(new Models.CartItem(new Models.ShopItem("T-shirt",
"One of the tops", 5), 1));

    }

    private void Add(Models.CartItem item)



    {

        item.Count++;

    }

    private void Remove(Models.CartItem item)

    {

        item.Count--;

    }
```

}

Directive Attribute

A directive attribute is mostly applied to a razor component and will affect parsing or the functionalities. It is more like a booster to the razor components.

One of the directive attributes that you might find extremely useful is It builds a data binding between the view and the model behind the scenes. When your application updates the model data, the frontend will render to reflect the change. Reminds you of MVVM right?

One-way Binding

Blazor provides a data binding feature, and we can bind the user input to our model.

For example, if a customer would like to change his item count in shopping cart from 1 to 100, with our former implementation, he might have to click on the add button for 99 times, which is definitely not a friendly user experience. And we could use binding with an input element so that this customer could enter any value he likes.

Replace the

label with a [ ] html element and bind it to To verify that the model value is updated correctly, write a Print method to log the count of the updated item:

@page "/cart"

@* some code *@

class="d-flex flex-row mb-3">

class="p-2 bi bi-dash-square" @onclick="()=>Remove(item)">

class="p-2" type="text"
@bind="item.Count" />

class="p-2 bi bi-plus-square" @onclick="()=>Add(item)">

class="p-2">@item.Count

@* some code *@

Let's buy a T-shirt, change the count to 100 and we should be able to the label next to the plus button is 100 now as well.

Binding Event

You may notice that in our last example, after changing the value of the item count, only when you click on somewhere else, the next label will be updated to the new value. But this may confuse our customers that if they don't click on anywhere, the value will not be updated, and they will pay for an order with undesired item numbers. It could be easily fixed with the directive

This directive defines under which event trigger, the binding will update value. By default, @bind:event is implicitly set to the onchange event. Defined in the HTML standards, onchange will be triggered when two conditions are met. The first is that content is changed, and the second one is when the element lost focus. So, in our case, when a customer clicks onelement and a, the item count input box will lose focus, and the onchange event is triggered, leading to the model value updated behind the scene.

To improve our user experience, we could set this bind directive on the input element to be oninput event:  class="p-2" type="text" @bind="item.Count" @bind:event="oninput" which is similar to the onchange event, and it will be fired immediately when the value changes, meaning every time a customer press any key, it will be triggered, and your binding model will be updated more frequently. It is a balance between the onchange or oninput event. If frequent updates fired by the oninput event will be a performance bottleneck for your application, you might consider using the default onchange event.

Binding Format

Another useful binding directive might be It defines the format to display the binding model. For example, our product owner decides to display an update time for each item in the cart with customized format, showing either when the item was put in the cart or when the item count was updated.

To show the time, we will add a DateTime property UpdateTime to the CartItem model and update it when the Count changes:

```
namespace EShop.Models

{

    public class CartItem

    {

        // some code

        private int _count;

        public int Count
```

```
        {
            get => _count;
            set
            {
                _count = value; UpdateTime = DateTime.Now;

            }
        }
        public DateTime UpdateTime { get; set; } = DateTime.Now;
        // some code
    }
}
```

And bind the UpdateTime to a readonly  element on the Cart razor page:

```
@* some code *@
```

class="d-flex w-100 justify-content-between">

class="mb-1">@item.Item.Name

readonly @bind="item.UpdateTime">

@* some code *@

When the format directive is not set, the application web page will display the default DatetTime string:

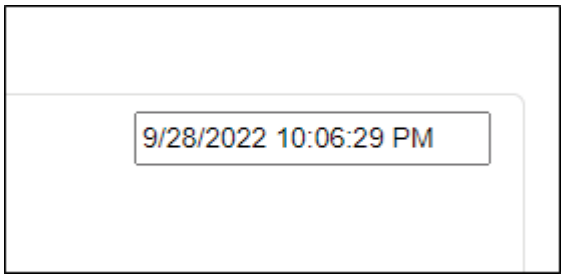

Figure 3.3: Default DateTime format

To match product owner's requirements, we could add a format directive to the element, readonly @bind="item.UpdateTime" @bind:format="HH:mm:ss"> and it will only display time but no date anymore:

Figure Customized DateTime format

As of the time of writing, format directive only supports DateTime expressions. To display more value formatting, developers have to add a customized string property for binding purposes.

Unparsed value

You might think of a question, what if customers enter a string instead of a number for the item count. And the truth is, binding directives will revert the value back to the last one automatically. In our example, we bind the value to the Count property of type int, and if you enter 10 first, and keep typing "H" on your keyboard, it will stay with 10 and has no effect.

Another customized way to handle unparsed value is to bind with a string value, and you can code the get and set methods to convert the binding string to the model data in your own way or your own format, as bind directive will call the get and set methods upon the binding event you specified.

Two-way binding

You realize that the small label section is used for each item and now it might be a good chance to refactor the Cart page and create a CartItemCount razor component dedicated to for the item count.

Let's create a new folder Components and a customized component CartItemCountComponent.razor under it:

class="d-flex flex-row mb-3">

class="p-2 bi bi-dash-square" @onclick="Remove">

class="p-2" type="text" @bind="Count" @bind:event="oninput" />

class="p-2 bi bi-plus-square" @onclick="Add">

class="p-2">@Count

```
@code{
    [Parameter]
    public int Count { get; set; }
    public void Add()

    {
        Count++;
    }
    public void Remove()
    {
        Count--;
    }
}
```

Note that we add a Count property and a Parameter attribute with it. This Parameter attribute tells blazor that Count will be exported as a parameter that can be bind to just like any other HTML attribute.

The cart page will also be updated accordingly to use our newly created component:

@page "/cart"

@using EShop.Components;

Cart

Cart

class="btn btn-danger" @onclick="Buy">Buy

class="list-group">

@foreach (var item in _cart.Items)

{

class="list-group-item">

class="d-flex w-100 justify-content-between">

class="mb-1">@item.Item.Name

readonly @bind="item.UpdateTime"

@bind:format="HH:mm:ss">

```
    class="mb-1">@item.Item.Description

        Count="@item.Count" />



    }


@code {

    private Models.Cart _cart = new Models.Cart();

    private void Buy()

    {

        _cart.Add(new Models.CartItem(new Models.ShopItem("T-shirt",
"One of the tops", 5), 1));


    }

}
```

Run the EShop application, buy a T-Shirt and try to update the item count. The label next to the input element is updated as well, however, there is a bug if you look closely, the updated time element does not sync with your typing. It never updates!

All we did is, in general, move the code from the page to a razor component that is used by this page, and the count label is showing the correct number, which means that the count does update. Now let’s take one step back and check our code where to update the Updated Time. We assign a new value to the UpdateTime property when the Count of CartItem is updated, and that gives us two potential reasons why the time is not displaying correctly in the browser. The first one is that we did not actually set the Count property, and the other one is that we updated the time but failed to render it. My bet is on the first one, and it can be easily proved. In the set method, we can add one line of code to print a log in the console, and set a debug breaking point on the set method. Now we run the application and enter some value in the input element. Neither the breaking point is hit nor the console prints

That is the essential difference between one-way and two-way binding directives. In one-way binding, we pass the data into the razor components as a parameter. But no matter how the components update the parameter, the world outside of the components will not be notified, and that explains why the time is never updated any more.

To fix this, we will officially introduce the two-way binding. And let's modify the CartItemCount razor component first.

class="d-flex flex-row mb-3">

class="p-2 bi bi-dash-square" @onclick="RemoveAsync">

class="p-2" type="text" value="@Count"
@oninput="OnInput" />

class="p-2 bi bi-plus-square" @onclick="AddAsync">

class="p-2">@Count

@code{

[Parameter]

public int Count { get; set; }

[Parameter]

public EventCallback CountChanged { get; set; }

```
    public async Task AddAsync()

    {

        Count++;

        await CountChanged.InvokeAsync(Count);

    }

    public async Task RemoveAsync()

    {

        Count--;

        await CountChanged.InvokeAsync(Count);

    }

    public async Task OnInput(ChangeEventArgs args)

    {

        if (args.Value is null ||
string.IsNullOrWhiteSpace(args.Value.ToString()))
```

```
        {

            Count = 1;

        }

        else

        {



            Count = int.Parse(args.Value.ToString());

        }

        await CountChanged.InvokeAsync(Count);

    }

}
```

There are a few high-lights in this version of CartItemCount component. The first is that we provide an EventCallBack event. With the Parameter attribute, this is the event that we expose to its' parent component. The parent component will handle the notification event with the generic payload to implement further logic.

In our example, we expose a CountChanged event, and we invoke this event everywhere the Count value get changed. From the page side, we upgrade the one-way binding to two-way binding as well, with the following code, @bind-Count="item.Count" @bind-Count:event="CountChanged" Unlike assigning item.Count to the Count property directly, we use the bind directive to bind the Count property with item.Count in two directions. If the value from the item.Count changed, the child item's property will be updated as usual. However, this time, if the child component's Count property is changed, since we invoke the notification event every time, the parent page will also be notified, and the set method of the CartItem model will be invoked as well, so that time displayed on the top right corner will reflect the latest changes.

Pay attention to the syntax here. Component parameters bind in a parent component using and {PROPERTY} represents the property to bind. In our case, the property is with the Parameter attribute. You also must implement an named by {PROPERTY}Changed with the Parameter attribute as well. These two "Parameters", together with the @bind syntax achieve the two-way binding between components.

Cascading

Passing parameters to low level components has been proved to be a powerful tool to build up a system with different levels. It helps to maintain clean code and clear business logic. But passing them just one level down is simply not enough. Sometimes, we need to refer to the context from more than one level up, as our application evolves, and as more features or requirements emerge. Once we found ourselves in such a situation with complex business rules, we will usually choose to refactor our code to be more object orientated with more than one level hierarchy.

Based on this reason, the developing team decided to refactor this Cart page once again. This time, we will build a dedicated Cart razor component, which will use a few CartItem components to represent the items customers selected for their cart.

A new challenge we are facing here is to take the data context from _dart object to the components deep down, with Parameter attributes, we are only allowed to pass the parameter one level down. To go further than that, we will introduce another attribute, With CascadingParameter and we could identify a uniformed data context throughout the components tree. A typical example would be to define a general CSS style for all the components in the hierarchy tree.

Our UI designed has defined a uniform fontsize for the EShop Cart page, and we will pass this fontsize standard to all the components referenced:

@using Models

@Cart.User

class="list-group">

@foreach (var item in Cart.Items)

{

class="list-group-item">

class="d-flex w-100 justify-content-between">

class="mb-1" style="@FontSize">@item.Item.Name

readonly @bind="item.UpdateTime"
@bind:format="HH:mm:ss">

```
class="mb-1">@item.Item.Description

    @bind-Count="item.Count" @bind-Count:event="CountChanged" />

  }

@code {

  [Parameter]

  public Cart Cart { get; set; }

  [CascadingParameter(Name = "FontSize")]

  public string FontSize { get; set; }

  [CascadingParameter(Name = "FontStyle")]

  public string FontStyle { get; set; }

}
```

Add the cascading properties to the CartItemCountComponent razor component:

class="d-flex flex-row mb-3">

class="p-2 bi bi-dash-square" style="@FontSize" @onclick="RemoveAsync">

class="p-2" style="@FontSize font-style: italic;" type="text" value="@Count" @oninput="OnInput" />

class="p-2 bi bi-plus-square" style="@FontSize" @onclick="AddAsync">

class="p-2" style="@FontSize">@Count

@code{

[Parameter]

public int Count { get; set; }

[Parameter]

```
    public EventCallback CountChanged { get; set; }

    [CascadingParameter(Name = "FontSize")]

    public string FontSize { get; set;}

    [CascadingParameter(Name = "FontStyle")]

    public string FontStyle { get; set;}

    @* some code *@

}
```

Define these CascadingValue in the most parent razor pages.

```
@page "/cart"

@using EShop.Components;

Cart
```

Cart

class="btn btn-danger" @onclick="Buy">Buy

Value="@_fontSize" Name = "FontSize">

Value="@_fontStyle" Name = "FontStyle">

Cart="@_cart" />

```
@code {
    private Models.Cart _cart = new Models.Cart("Brian");
    private string _fontSize = "font-size: 30px;";
    private string _fontStyle = "font-style: italic;";
    private void Buy()
    {
```

```
        _cart.Add(new Models.CartItem(new Models.ShopItem("T-shirt",
"One of the tops", 5), 2));

        _cart.Add(new Models.CartItem(new Models.ShopItem("Jacket",
"The most popular", 17), 1));

    }

}
```

The key idea is to wrap your components in a CascadingValue label. In this label, you will define a data context that will be passed downward throughout the whole components tree. Each child component, which is intended to take the data context, can define a property with a CascadingParameter attribute. If you have more than one you will give the Name attribute a value, and wrap these CascadingValue one inside another, and the child components will be placed in the most inside. Components' CascadingParameter will also catch the data by assigning a Name field.

In our example, we defined a universal fontsize and a universal fontstyle in the page level, and we passed these standard styles into the tree. The children are now also rendered with the new style. In general, CascadingValue will be your best option to set up a global theme or style, as the theme or style built in this way can take effect easily for your whole application.

Event Handling

Another popular feature in component is event handling. In the world of Blazor components, you can define an event using You've actually seen this feature in the two-way binding section, where we notify the parent component that data has been updated. Unlike the event keyword in normal .NET world, in components, you define an event with the help of the struct EventCallback can only be subscribed by one method, and this will be the only one method that will be invoked, when we emit an event. While in normal .NET, an event can be subscribed by multiple methods, and these methods will be invoked together while emitting the event.

A new user story comes from our product owner. EShop will allow customers to remove items from the card and record the log when an item is removed, so that the business operation can analyze the data and improve customers' experience.

Add an EventCallback OnCartItemRemoved to the CartComponent razor component:

@using Models

@Cart.User

class="list-group">

@foreach (var item in Cart.Items)

{

class="list-group-item">

class="d-flex w-100 justify-content-between">

class="mb-1" style="@FontSize">@item.Item.Name

readonly @bind="item.UpdateTime" @bind:format="HH:mm:ss">

class="p-2 bi bi-x-octagon-fill" @onclick="()=>Remove(item)">

```
    class="mb-1">@item.Item.Description


        @bind-Count="item.Count" @bind-
    Count:event="CountChanged" />




  }


@code {

  [Parameter]


  public Cart Cart { get; set; }

  [CascadingParameter(Name = "FontSize")]

  public string FontSize { get; set; }

  [CascadingParameter(Name = "FontStyle")]
```

```
  public string FontStyle { get; set; }

  [Parameter]

  public EventCallback OnCartItemRemoved { get; set; }

  private void Remove(CartItem item)

  {

    OnCartItemRemoved.InvokeAsync(item);

  }

}
```

We will pass an method as a delegate to the OnCartIteAlibaba and back:

```
@page "/cart"

@using EShop.Components;
```

Cart

Cart

class="btn btn-danger" @onclick="Buy">Buy

Value="@_fontSize" Name = "FontSize">

Value="@_fontStyle" Name = "FontStyle">

Cart="@_cart" OnCartItemRemoved="OnCartItemRemoved" />

@code {

private Models.Cart _cart = new Models.Cart("Brian");

private string _fontSize = "font-size: 30px;";

private string _fontStyle = "font-style: italic;";

private void Buy()

{

```
        _cart.Add(new Models.CartItem(new Models.ShopItem("T-shirt",
"One of the tops", 5), 2));

        _cart.Add(new Models.CartItem(new Models.ShopItem("Jacket",
"The most popular", 17), 1));

    }

    private void OnCartItemRemoved(Models.CartItem item)



    {

      _cart.Remove(item);

    }

}
```

In the we expose an EventCallback if CartItem as a parameter, so the page could call the component and assign a method to this callback just like any other parameters we defined before. Later on, we could invoke this callback in the child component at any time, given a CartItem instance as the callback argument. In the method that subscribes, it will be executed while we invoke the callback. In this way, we implemented the removing of CartItem in the page, when we click on the remove button in the child component.

Lifecycle

In a razor component, there are multiple virtual methods that we could choose to override to alter the rendering behavior for this component. These virtual methods are, in the lifecycle order, OnAfterRenderAsync SetParametersAsync will wrap all the parameters set to this component in the ParameterView object, and you can get the parameter by its name. This is a good chance now to retrieve data from the backend API with parameter values. Next, you will call the base in the base method implementations, it invokes OnInitializedasync if this is the first time creating this component instance, and then immediately invokes Till now, you can get the component's state and parameters set.

Whenever you want to explicitly rerender a component, you can call StateHasChanged to trigger the rendering. Blazor will then check with This method returns a boolean value to inform Blazor whether a rendering will take place or not. If you return false in this method, you will prevent Blazor from rerendering this component, to improve users' experience and build up a high-performance application. BuildRenderTree is another virtual method you can override if no markup code has been written yet for this component. In this method, you may define how to render the component specifically by coding. Once Blazor completes the rendering, OnAfterRenderAsync will be invoked, and it will now be safe to call your ref component here. The last method in component lifecycle is Dispose if the component implements IDisposable interface. When the component is removed from the rendering tree, Blazor will invoke this method, and it will be good chance for you to dispose all the resources.

Layout

With three default pages, and the Cart page we developed, there are a total of four pages in our EShop application currently, and they all share the same top head and side navigation bar. This is implemented by a layout in Blazor. If you are building a complex web application, it tends to have more than one page, and it is important to keep a consistent head or navigation menu throughout your application. Layout can be displayed in every page, and it saves you from duplicated code, and it keeps it easy to maintain the shared consistency.

A layout is in fact a special component in Blazor. All the layouts will inherit from the which inherits the same base class that a normal component will inherit. It provides a special Parameter of type Body. In your implemented layout, you can render the @body anywhere. Therefore, it is likely that you will add top head and side navigation menus in the layout, just like the default Blazor example and render the body in an article element:

@inherits LayoutComponentBase

class="page">

class="sidebar">

/>

```
class="top-row px-4">

        href="https://docs.microsoft.com/aspnet/"
target="_blank">About




class="content px-4">

        @Body
```

To wrap contents in your layout, you need to declare where you want to place your components by using the @layout directive.

Layouts can be nested just like normal components have child components. To render a layout inside another layout, you must refer to the parent layout in your child layout and render your content inside the child layout.

Let's create one our own layout then. Under the Shared folder, create a new file named

@inherits LayoutComponentBase

EShop Blazor Application

@Body

We first create a new parent EShop layout, and in the second line of we refer to the EShop layout by adding @layout To verify that we do have 2 layouts together, let's update ./App.razor and change the layout for NotFound from @typeof(MainLayout) to

Now run the application, and you will not miss the extra label element EShop Blazor Application on the top of the web pages. And if you browse to an undeclared route, for example, the navigation menu will disappear with only the EShop Blazor Application label left, which means not found route will be rendered directly inside the new while Cart page is rendered in the nested

Libraries

In the open-source communities, there are a number of popular Blazor components libraries, that you could use in your project.

Fast-blazor

https://github.com/microsoft/fast-blazor

This is a Blazor component library that implements the Microsoft FluentUI. It is lightweight, and compatible with .NET 6 applications.

MatBlazor

https://github.com/SamProf/MatBlazor

This library implements common components following material design specification. It has a complementary demo websites and documentation.

Ant Design Blazor

https://github.com/ant-design-blazor/ant-design-blazor

Ant Design is an enterprise level design language by Alibaba and has a strong ecosystem. This library contributes a Blazor implementation to the Ant Design Community

BootstrapBlazor

https://github.com/dotnetcore/BootstrapBlazor

This is also an enterprise-class library. It is implemented based on the popular Bootstrap styles.

Conclusion

In this chapter, we covered the basic concepts of components and built an interactive shopping cart page with nested components and data binding between components and models. We've learned how to pass data down through multiple levels of hierarchies and handle the events emitted by the component. We also introduced the lifecycle of a component and used a special component, layout, to reuse existing components. Finally, we quickly went through some popular Blazor components library in the open-source community.

In the next chapter, we will discuss more about the mechanism behind the components and layout and take some source code as examples to learn the techniques to improve your applications' performance.

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4

Advanced Techniques for Blazor Component Enhancement

Introduction

In the last chapter, we introduced the razor components as a core building material for a Blazor application. In this chapter, by digging into the source code, we will go through some more advanced ways to enhance your components' features and performances.

Structure

In this chapter, we will discuss the following topics:

How to reference other components

How to preserve elements or components

How to template components

CSS Isolation

Objectives

In this chapter, we will be understanding the advanced topics around Blazor components. For example, components reference and preservation. We will understand the components template and css isolation as well to help you better organize your code structure.

Component Reference

Sometimes, you may want to control other components in your own component. In the world of HTML and JavaScript, you usually add an id attribute to the element that will be manipulated. While in Blazor, there is no easy way to get the element by its id in C# code. One way to solve the problem is to use one of the directive attributes that we introduced in the previous chapter.

@ref directive attribute allows developers to reference another component, and, in this way, we could invoke the method just from the referenced component instance. For example, we would like to check out the items added to the shopping cart, and we can reference the CartComponent and call the method Checkout from it, just like you call a method from any other instance in normal C# code.

```
// some other code

  public void Checkout()

  {

    foreach (var item in Cart.Items)

    {
```

```
    Console.WriteLine($"Checkout {item.Count} {item.Item.Name}, the total is {item.Count * item.Item.Price}");

  }

}
```

First, we add a new Checkout method in the CartComponent source code. This method will iterate through all the items in the cart and print out the log displaying the counts, names and the total prices.

Next we add a @ref directive attribute to the CartComponent in the CarPage.razor and use this component reference as if it is one of the properties of this page class.

```
@page "/cart"

@using EShop.Components;

Cart
```

Cart

class="btn btn-danger" @onclick="Buy">Buy

Value="@_fontSize" Name = "FontSize">

Value="@_fontStyle" Name = "FontStyle">

@ref="cartComponent" Cart="@_cart"
OnCartItemRemoved="OnCartItemRemoved" />

class="btn btn-warning" @onclick="Checkout">Checkout

@code {

private Models.Cart _cart = new Models.Cart("Brian");

private string _fontSize = "font-size: 30px;";

private string _fontStyle = "font-style: italic;";

```
    private CartComponent cartComponent;

    private void Buy()

    {

        _cart.Add(new Models.CartItem(new Models.ShopItem("T-shirt",
"One of the tops", 5), 2));

        _cart.Add(new Models.CartItem(new Models.ShopItem("Jacket",
"The most popular", 17), 1));

    }

    private void OnCartItemRemoved(Models.CartItem item)

    {

      _cart.Remove(item);

    }

    private void Checkout()

    {

      cartComponent.Checkout();
```

```
    }


}
```

Here, we place the business logic, in the razor component, but in practice, you may want to add more interaction logic between users and the application instead of business logic. Usually, business logic belongs to the model layer.

Anyway, in our example, we have a public method in the and we use it in the CartPage to display the cart detail to our customers. Before we add the @ref directive attribute to the component, we merely depend on the capabilities of the CartComponent by itself. Since we use the @ref directive attribute to catch the reference of this component in the razor page, we have more flexibility in controlling the behavior of the component and hence be able to add more functionalities to the cart page. It will be like calling a normal C# instance, and you can call any instance member which is marked publicly accessible.

Components preserving

In a web application, it is very common to show a list of variable kinds of data, or a table. It could be a table of employees' pay slips or last month's sales revenue, or simply a list of forecasted weather for the next week. Sometimes, these lists, or data are static. They remain the same through the users' certain experiences. Sometimes, they are dynamic, and you must keep refreshing the data, as a result, your web application has to keep re-rendering itself, and it could consume a lot of resources if you have a huge amount of data to be refreshed.

For example, you are developing a new feature of wish list. A wish list is a placeholder that customers can save the item they like for later, and not checkout for now. One common scenario in this feature is that customers may frequently move items from cart to their wish list back and force several times for some reasons while they are shopping. In such cases, the cart item or wish list in your application will be re-rendered entirely in conjunction with the frequency that a customer updates it. Especially, when there are a large number of items in the list, there will be performance downgrade, and this will obviously not be welcomed by customers. It is now the developers' responsibility to bring up a solution to boost the performance. Luckily, Blazor provides a directive attribute @key to save your effort, and you won't have to manually control the re-render algorithm to boost the performance.

Here, we create a new file WishListComponent.razor under the Components folder and build a wish list component with the corresponding models as well.

class="list-group">

@foreach (var item in WishList!.Items)

{

- originalValue="@item.Item.Name">

class="list-group-item">

class="d-flex w-100 justify-content-between">

class="mb-1" style="@FontSize">@item.Item.Name

readonly @bind="item.UpdateTime" @bind:format="HH:mm:ss">

class="p-2 bi bi-box-arrow-up" @onclick="()=>MoveToCart(item)">

class="mb-1">@item.Item.Description

}

The corresponding file WishListComponent.razor.cs is created for the code behind the component:

using Microsoft.AspNetCore.Components;

using EShop.Models;

```
namespace EShop.Components

{

    public partial class WishListComponent

    {

        [Parameter]

        public WishList? WishList { get; set; }

        [CascadingParameter(Name = "FontSize")]

        public string? FontSize { get; set; }

        [CascadingParameter(Name = "FontStyle")]

        public string? FontStyle { get; set; }

        [Parameter]


        public EventCallback OnMovingToCart { get; set; }

        private async void MoveToCart(WishListItem item)

        {
```

```
            await OnMovingToCart.InvokeAsync(item);

        }

    }

}
```

This time, we separate the component logic into a .razor.cs file. It is a common pattern in developing WPF projects where .xaml file will be mapped with .xaml.cs file. Blazor shares the same spirit here as well. The WishListComponent is in fact a partial class inheriting ComponentBase while partial of the class is UI, another part is pure C# code representing the component logic. The wish list models are very similar to the cart and will not be covered here.

Now, we can use the WishListComponent in the

```
@page "/cart"

@using EShop.Components;

Shopping here!
```

Shopping here!

class="btn btn-danger" @onclick="AddToCart">Add something to the cart

Value="@_fontSize" Name="FontSize">

Value="@_fontStyle" Name="FontStyle">

Cart

@ref="cartComponent" Cart="@_cart"
OnCartItemRemoved="OnCartItemRemoved"
OnMovingToWishList="OnMovingToWishList" />

Wish List

@ref="wishListComponent" WishList="@_wishList" OnMovingToCart="OnMovingToCart" />

class="btn btn-warning" @onclick="Checkout">Checkout

@* some code *@

And add the C# code to implement the item switch logic between cart and wish list:

@* some code *@

@code {

private const string USER = "Brian";

private Models.Cart _cart = new Models.Cart(USER);

```
  private Models.WishList _wishList = new Models.WishList(USER);

  private string _fontSize = "font-size: 30px;";

  private string _fontStyle = "font-style: italic;";

  private CartComponent? cartComponent;

  private WishListComponent? wishListComponent;

  private void AddToCart()

  {

    _cart.Add(new Models.CartItem(new Models.ShopItem("T-shirt", "One of the tops", 5), 2));

    _cart.Add(new Models.CartItem(new Models.ShopItem("Jacket", "The most popular", 17), 1));

    _cart.Add(new Models.CartItem(new Models.ShopItem("Sun glasses", "On sale", 8), 3));

  }

  private void OnCartItemRemoved(Models.CartItem item)

  {
```

```
  _cart.Remove(item);

}

private void Checkout()

{

  cartComponent!.Checkout();

}

private void OnMovingToCart(Models.WishListItem item)

{

  _wishList.Remove(item);

  var cartItem = new Models.CartItem(item.Item, item.Count);

  _cart.Add(cartItem);

}

private void OnMovingToWishList(Models.CartItem item)
```

```
  {
    _cart.Remove(item);
    var wishListItem = new Models.WishListItem(item.Item, item.Count);
    _wishList.Add(wishListItem);
  }

}
```

We also updated the where cascaded style and fonts are shared between the CartComponent and the Users could now click on the small moving to wish list on the right top corner for each cart item and move it to the wish list. Or click on the similar button on the right top corner to reverse it back.

But we have not done any performance improvement yet. If a user moves an item back and forth, performance will be hurt. Next, let's add the @key directive attribute to each cart or wish list item div element to let the Blazor know that these elements should be retained and how the cart items are mapped to them.

```
@* some code *@

@foreach (var item in Cart!.Items)
```

{

class="list-group-item" @key="item.Item.Name">

@* some code *@

We could use any object to key the element or component. It could be a string, a number, or an instance. And Blazor will retain the relationship between the element and the cart item by its name. The best practice is to use @key directive attribute where you are developing a list component and you may use the item ID to key the component. Not only to retain a mapping, don't forget that also marks that the relationship should be disposed when there is an instance replacement, and Blazor will re-render the front-end pages.

Template components

As your application grows, there will be many places where repeated front-end code could be eliminated by using components. In some cases, this would be an easy replacement. For example, when we are developing a CartItemCountComponent is used to save the effort allowing customers to update how many items they would like to buy. Till now, we are dealing with components that are purely developed by writing the HTML code explicitly. While in other scenarios, there might be a chance that the consumer of your components will provide the HTML to be rendered inside your component.

For example, in the you could wrap the contents inside the for each loop to another component, and this new component must have the capability to render the children html in the list group container.

One straight forward way to implement this is to pass the HTML code as string, and the new component will have a string parameter, and render the HTML explicitly as follows:

Here we create GroupContainer.razor under the Shared folder:

class="list-group-item">

class="d-flex w-100 justify-content-between">

class="mb-1" style="@FontSize">@Name

@ChildContent

class="mb-1">@Description

GroupContainer.razor.cs are created here as well:

using Microsoft.AspNetCore.Components;

namespace EShop.Shared

```
{
    public partial class GroupContainer
    {
        [Parameter]
        public string? Name { get; set; }
        [Parameter]
        public string? Description { get; set; }
        [Parameter]
        public string? FontSize { get; set; }
        [Parameter]

        public RenderFragment? ChildContent { get; set; }
    }
}
```

Now we explicitly use this GroupContainer to make sure that every item is rendered in the exact same container:

@using Shared

class="list-group">

@foreach (var item in Cart!.Items)

{

Name=@item.Item.Name Description=@item.Item.Description>

readonly @bind="item.UpdateTime" @bind:format="HH:mm:ss">

class="p-2 bi bi-box-arrow-down" @onclick="()=>MoveToWishList(item)">

class="p-2 bi bi-x-octagon-fill" @onclick="()=>Remove(item)">

@* @bind-Count="item.Count" @bind-Count:event="CountChanged" /> *@

}

Another way to implement a HTML wrapper would be as shown in the codes. It is a convention that when you define a property of type RenderFragment with the name everything filled inside the component will be rendered where you place the @ChildContent in the razor component. In our example, @ChildContent is placed in a div element, and when another team or some other developers utilize your GroupContainer component, they add an input element and two icons in it. These three added elements will be rendered exactly inside the div element.

RenderFragment is a delegate that use a RenderTreeBuilder to build the UI content, as it is defined as follows:

public delegate void RenderFragment(RenderTreeBuilder builder);

The preceding code shows the definition of which represents a method with a RenderTreeBuilder instance as parameter to customize the content, and there are a few common methods made public to accomplish this objective.

Now that you know RenderFragment is a delegate, so we could try writing some HTML elements in pure C# code.

First, we add a RenderFragment field in

using Microsoft.AspNetCore.Components;

namespace EShop.Shared

```
{
    public partial class GroupContainer
    {
        [Parameter]
        public string? Name { get; set; }
        [Parameter]
        public string? Description { get; set; }
        [Parameter]
        public string? FontSize { get; set; }
        [Parameter]
        public RenderFragment? ChildContent { get; set; }
        private RenderFragment _descriptionRF = (b) =>
        {
            b.OpenElement(0, "p");
            b.AddAttribute(1, "class", "mb-1");
```

```
            b.AddContent(2, "this is a description.");

            b.CloseElement();


        };

    }

}
```

Then we use this RenderFragment in GroupContainer.razor to determine where it will be rendered:

```
class="list-group-item">


class="d-flex w-100 justify-content-between">


class="mb-1" style="@FontSize">@Name
```

@ChildContent

@_descriptionRF

In the preceding example, we first render @_descriptionRF to replace the original p element showing the item description. It looks just like the And yes, we created a RenderFragment in the corresponding *.razor.cs file. Next to the ChildContent property, we add a private RenderFragment called exactly the name in the razor file. Given a lambda expression, b is the

In the OpenElement comes first, the same as you are writing the HTML code, you would type

first. Next, we call AddAttribute to give the p element a class attribute, just like you type class='mb-1' in HTML. The third line, obviously, gives the p element a content to display. Finally, we end the lambda expression by calling the same when you type the closing label

It is all very straightforward, but there is an important aspect that we missed here, the integer parameter of these first three methods. The integer here in fact plays an important role in the rendering tree. An

interesting thing is, when we re-number these integers in the lambda expression, it still compiles and runs:

```
// some code

private RenderFragment _descriptionRF = (b) =>

{

    b.OpenElement(-3, "p");

    b.AddAttribute(-2, "class", "mb-1");

    b.AddContent(-1, "this is a description.");

    b.CloseElement();

};


// some code
```

Figure Description RenderFragment

So, you see, the sequence can start from any number, as long as they are incremental. One common mistake that developers may make is to use auto incremented index:

```
int index = 0

b.OpenElement(index++, "p");

b.AddAttribute(index++, "class", "mb-1");

b.AddContent(index++, "this is a description.");

b.CloseElement();
```

It seems more reasonable to use an integer tracking the sequence since we are all developers. But the difference between Blazor and other UI frameworks is that it uses this sequence to calculate the tree diff with a linear algorithm. And it may work fine in this trivial example. Once you have a more complicated scenario. Such as if branch, or loops, auto generated sequence numbers may hurt your performance. The key here is the algorithm Blazor takes and compares the old and new rendering with the same sequence. It means for every same sequence number, if the data comes with that number changes, Blazor is convinced that it will re-render that data. In a loop for a list, it is very common that some items will be removed or added in the middle based on your business logic, and there will be re-rendering for every data comes after that

removing or adding, even though most of them remain the same, because the corresponding sequence is changed if you use auto increment numbers.

Another tip here is always remembered to call AddAttribute right after the call of OpenElement or otherwise you will encounter the following runtime exception:

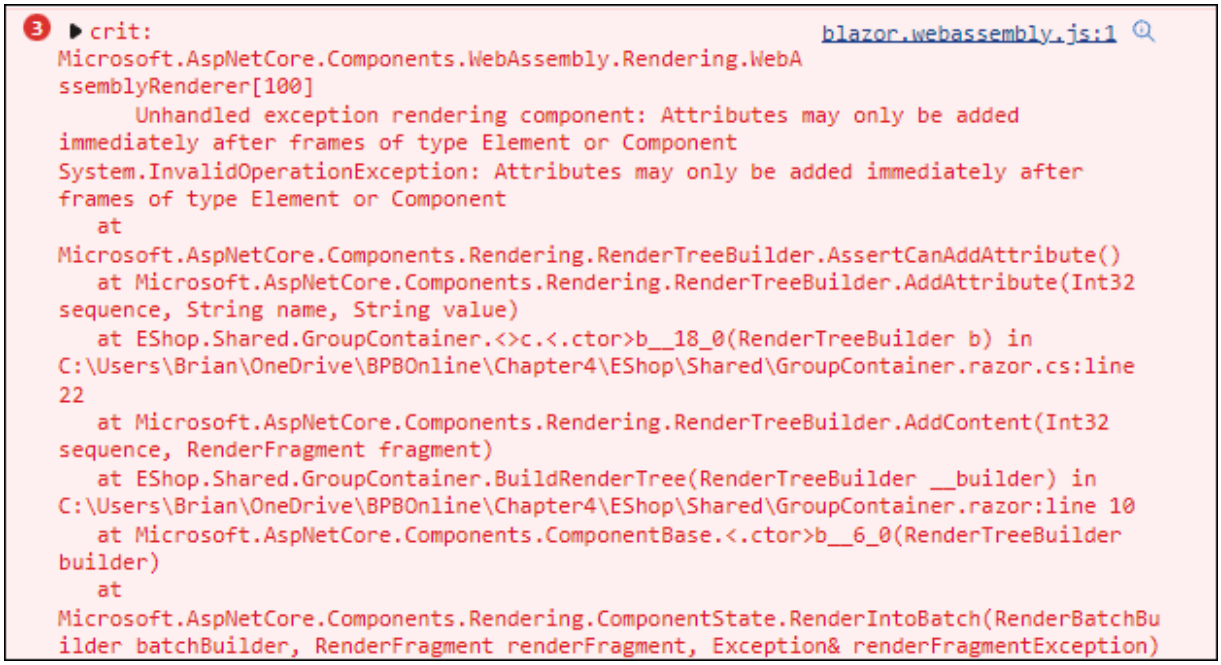

Figure AddAttribute exception

Currently, this RenderFragment is static, and always renders the same description. But we are required to display specific descriptions for each item. So, a generic version of RenderFragment can be used here:

```
// some code

private RenderFragment _descriptionRF = (desc) => (b) =>
```

```
        {
            b.OpenElement(-3, "p");
            b.AddAttribute(-2, "class", "mb-1");
            b.AddContent(-1, desc);
            b.CloseElement();
        };
        // some code
```

And we pass the Description as a parameter to the

```
class="list-group-item">

class="d-flex w-100 justify-content-between">

class="mb-1" style="@FontSize">@Name
```

@ChildContent

@_descriptionRF(Description!)

Generic RenderFragment is a delegate as well, and it is defined to return a normal It allows developers to render more dynamic content. This generic argument provides a typed parameter to the RenderFragment delegate and can be passed to the Inside the normal RenderFragment lambda expression, developers can customize the output with this parameter.

RenderFragment provides a way to manually generate HTML elements, but it must be used with caution, otherwise it will easily overcome the benefits it brings. It is recommended not to use RenderFragment unless you have a very strong reason to do so.

CSS Isolation

CSS style is widely used throughout web applications. However, there are some common challenging that developers are facing nowadays. When your application grows larger and more complex, it will be barely easy to maintain the style dependencies. It will be hard to locate the style related bug on deeply nested elements.

Blazor CSS isolation aims to avoid these problems by defining a style dedicated to only one component. And it is more than easy to implement an isolated CSS style. You only need to name your css file prefixed with the name of corresponding razor file and put them under the same path.

For example, in we have a

element:

class="list-group-item">

class="d-flex w-100 justify-content-between">

@Name

@ChildContent

@_descriptionRF(Description!)

And we simply create GroupContainer.razor.css under the Shared folder with style for

element:

```
h2 {

    color: orange;

}
```

In this way, all the h2 elements in GroupContainer component will have the same orange color. Super easy, right? If you inspect the generated web page html, you may notice that:

b-xk4f1vd9ci>T-shirt

The h2 element was added as an attribute by the compiler in the format of b-{10 In our case, it is and this attribute makes the selector more specific. To prove this, we can bring up the DevTools and you will find h2[b-xk4f1vd9ci] style in All the isolated css style will be compiled into this file together:

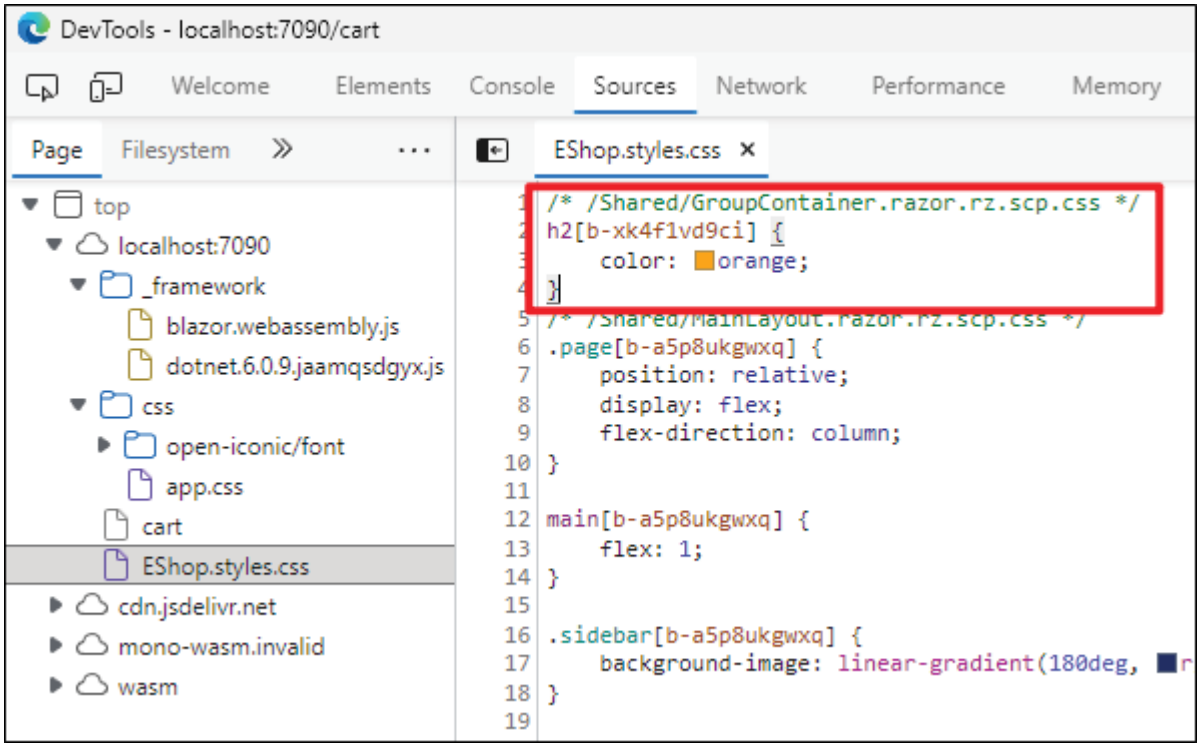

Figure Compiled isolated CSS style

Since the css is isolated, it will not be applied to the child elements or components. To allow children inheriting the same css style, you can use ::deep to indicate that this style will apply to its children.

Let's conduct a small experiment here, adding both a parent header and a child header:

```
@* some code *@

Name=@item.Item.Name Description=@item.Item.Description>
```

child header

readonly @bind="item.UpdateTime" @bind:format="HH:mm:ss">

class="p-2 bi bi-box-arrow-down" @onclick="()=>MoveToWishList(item)">

class="p-2 bi bi-x-octagon-fill" @onclick="()=>Remove(item)">

@* some code *@

In we add a h2 element, since it is wrapped inside the this will be the child header, in add the code

parent header

to the first line, as a parent header. Finally update CSS selector from h2 to ::deep Now, run the application, and you will find that the child header is orange, but the parent header is still black, CSS style does not work!

Figure Inherited css

That's because the ::deep selector will not match those elements at the root. So, you either wrap the parent header in a or add another h2 selector to the css file:

```
h2, ::deep h2 {

    color: orange;
```

```
}
```

This time, all the headers will be orange.

Conclusion

In this chapter, we explored more advanced topics of Blazor components. Starting with components reference, we showed that other components can be referenced just like a variable in plain C# code by adding the @ref directive attribute to the target component. Next, we discussed the technics to preserve the relationship between components with its model. @key directive attribute helps to retain the mapping and is encouraged to be used inside a list or a loop.

We also discussed how to manually generate front-end elements in pure C# code. Remember to use RenderFragment only when necessary. Finally, we talked about CSS isolation to avoid dependency chaos and how this isolation is implemented by the compiler.

In the next chapter, we will build a file uploading components with the knowledge we have learned and discuss how to handle files from or to customers. Some HTTP related topics will be covered as well, through our way.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 5

File Uploading in Blazor

Introduction

In the last chapter, we discussed some advanced topics about Blazor components, including components reference, components preserving, and component template. We also introduced the CSS isolation technique to avoid overlap issues. In this chapter, we will demo how you can interact with users through file uploading and downloading for both Blazor Server and Blazor WebAssembly.

Structure

In this chapter, we will discuss the following topics:

File transfer on the internet

Upload a file

Upload files to other service providers

Security considerations and best practices

Objectives

In this chapter, first, we will be introduced to the protocols that are commonly used to transfer files between clients through the internet, and we will understand the differences between these two protocols, HTTP and FTP. Next, we will understand the InputFile component that is used in the Blazor Framework to allow users to upload their files to the servers. We will be explained the detail through a EShop feature where customers will upload photos taken from the bought item and leave comments for that item.

Build comments for EShop

File sharing can be a verify useful way to communicate between users and your web application or between users. If it is a social media application, you might want to allow customers to upload their favorite photos, so that other users can leave a comment or share that photo. If it is a business intelligence application, customers may want to download the data as an Excel file or a PDF report. In previous chapters, we built an EShop application, and it would make our product more competitive if the seller could sign in and post a selected picture for the products on sale or customize a header picture for the sellers' home page.

File transfer

There are two common ways to transfer files on the internet, HTTP and FTP. HTTP stands for HyperText Transfer Protocol. As the name suggests, HTTP deals with HyperText, which means, the client or web browser will send a request to retrieve the HyperText from a server. HyperText was introduced by Ted Nelson in 1956, and it is still text, but a text with more information, linking other HyperTexts by cross-referencing each other. These links between texts are like edges in one huge graph in the world of internet with texts as the nodes. These nodes are connected with each other, more or less through the links. So that while you are at one node, you can almost travel to any other node directly, or with only a few steps, unlike a bus route, where you must keep traveling to the next stop linearly until your destination, even though the stops in between are not valuable to you at all.

And files are obviously part of the HyperText in our internet world. HTTP is a one-way communication protocol as we discussed in Chapter Choose Hosting for Blazor, and a client will initiate a request to get the desired file from the HTTP server. Authentication is not required to get the file, by its nature, through HTTP protocol, although nowadays, developers are adding more and more sophisticated signing-in mechanisms. But all these authentication logics are not required by HTTP. They come from the business.

On the other hand, FTP is born with authentication. FTP stands for File Transfer Protocol. You may tell from its name that this protocol is born to share files. Users usually first provide credentials to log into the server and get permission to upload or download files. Unlike HTTP, FTP is more appropriate for large file transferring, and it is widely used in large enterprises, universities, and institutions.

Nowadays, both HTTP and FTP get their security extensions to HTTPS and FTPS, supporting Transport Layer Security. By enhancing data transfer through encryption algorithms, HTTPS and FTPS have become more and more popular, or even a new must-have requirement in the internet world. For example, Google Chrome will default to HTTPS starting from version 90. In this chapter, we will focus on file transfer through HTTPS for the web application.

File upload

Both in Blazor Server and Blazor WebAssembly, we use an InputFile component to catch the file from the user's upload. By default, this component will be rendered into an input element. In our EShop application, the product owner would like to add a new feature that allows customers to upload their comments with photos for shopping items.

First, let's build a shop-item page with customized components using the techniques we learned from previous chapters.

First, we create a new file Comments.cs under the Models folder:

```
namespace EShop.Models

{

    public class Comment

    {

        public string Content { get; set; }

        public string ImageUrl { get; set; }
```

```
        public DateTime CreatedTime { get; set; }

        public Comment(string content, string imageUrl)

        {

            Content = content;



            // ImageUrl = imageUrl;

            ImageUrl = "https://ts1.cn.mm.bing.net/th/id/R-
C.699defcec77365c2dcd1bac50a789a46?
rik=sk9qvtl%2fpa%2f5eA&riu=http%3a%2f%2fcdn57.picsart.com%
2f179951678001202.jpg&ehk=SEq86wyqKmauSBKOyess7Qp6gtM
56dWGBQia7SwkJHg%3d&risl=&pid=ImgRaw&r=0";

            CreatedTime = DateTime.Now;

        }

    }

}
```

In this Comment model, we designed three properties and for now we will use a picture from Bing Image to represent the shop item:

```
namespace EShop.Models

{

    public class ShopItem

    {

        public string Name { get; set; }

        public string Description { get; set; }

        public string ImageUrl { get; set; }

        public double Price { get; set; }



        public List Comments { get; }

        public ShopItem(string name, string description, double price)

        {

            Name = name;

            Description = description;
```

```
            Price = price;

            Comments = new List();

        }

        public void AddComment(Comment comment)

        {

            Comments.Add(comment);

        }

    }

}
```

We added a new property of the comments list in the ShopItem model, and a method to add a comment for this shop item. The comment model has three properties. One is the content of this comment. Another is the image for this shop item that a buyer bought from our EShop website, and this photo taken from the buyer can be included with this comment. For now, we use a random photograph from the Bing Image to display in the application. Later, we will replace this default image with user uploaded one. The last one is an

auto-generated time of the comment creation to take note of when the buyer left this comment.

Next, we create a new Blazor component file Comment.razor under the Shared folder:

```
class="Container">



class="Content">@CommentModel.Content


    @if(!string.IsNullOrWhiteSpace(CommentModel.ImageUrl))

    {

        class="Image" src="@CommentModel.ImageUrl" />

    }



class="Time">@CommentModel.CreatedTime.ToString("yyyy-mm-dd HH:MM:ss")
```

This component will display the content, image, and the created date of a comment.

Then we create the corresponding C# code behind file Comment.razor.cs to hold the model for this component:

```
using Microsoft.AspNetCore.Components;



namespace EShop.Shared

{

    public partial class Comment

    {

        [Parameter]

        public Models.Comment? CommentModel { get; set; }

    }

}
```

And a corresponding style file Comment.razor.css to give the component a nicer look:

```
.Container {
```

```
    border-width: 0 0 1px 0;

    border-style: solid;

    border-color: black;

    margin-bottom: 15px;

}

.Content {

    margin: 1px;

}


.Image {

    width: 300px;

    margin-bottom: 15px;

}

.Time {

    font-size: 12px;
```

```
    color: gray;

    margin: 1px;

}
```

We build a shared component for the comment model here. It will display the content of the comment, and the photo user uploaded, together with when the comment was left. So that other users who are interested in this shopping item can view the real photo taken by a real buyer, instead of the photo that could be highly augmented by the seller.

Then we will construct a new page to display the shop item and the comments of it. Create a new file ShopItem.razor under the Pages folder.@page

```
@using EShop.Shared;

class="ItemContainer">
```

@Item.Name

$ @Item.Price

@Item.Description

class="ItemDisplay">

src="@Item.ImageUrl" />

class="NewComment">

type="text" @bind="Content"/>

class="btn btn-primary NewCommentBtn" @onclick="AddComment">Leave a comment

class="Comments">

Comments

```
@foreach(var comment in Item.Comments)
{
    CommentModel="@comment" />
}
```

Customers will be able to see the name, price, and description of the selected shop item on this new page, and we use the newly created Comment component in a foreach loop to list all the comments for this item.

In the corresponding C# code behind file a model of the ShopItem will be held and we provide a method AddComment so that customers can click on the button to add their own comments:

```
namespace EShop.Pages

{

    public partial class ShopItem

    {

        public Models.ShopItem Item { get; }

        public string? Content { get; set; }

        public ShopItem()

        {

            Item = new Models.ShopItem("T-shirt", "The best ever with lower price!", 19.9);

            Item.ImageUrl = "https://ts1.cn.mm.bing.net/th/id/R-C.614bdee2065be0f1976bdf839c725e26?rik=EJ2vSWnKs9a9vQ&riu=http%3a%2f%2fclipart-library.com%2fimg%2f828773.png&ehk=avi5QwUJFS0v4Qtu8ggI5Ariopp4uJwf7r5QlOnJQ0o%3d&risl=&pid=ImgRaw&r=0";

        }
```

```
        private void AddComment()
        {
            System.Console.WriteLine("AddComment");
            if (string.IsNullOrWhiteSpace(Content))
            {
                return;
            }
            var comment = new Models.Comment(Content!, "");
            Item.AddComment(comment);
            Console.WriteLine(Content);
        }
    }
}
```

And a CSS file ShopItem.razor.css to make the page looks nicer and make the layout more concise:

```
.ItemContainer {
    display: flex;
    flex-direction: column;

    align-items: center;
}
.ItemDisplay {
    display: flex;
    justify-content: space-around;
}
.NewComment {
    display: flex;
    flex-direction: column;
    width: 300px;
    margin-top: 20px;
```

```
    align-items: flex-start;

}

.NewCommentBtn {

    margin-top: 10px;

}

.Comments{

    margin-top: 20px;



}
```

Next, we build a new page for displaying shopping items. It shows the shopping item title with its price and description. On the left, you will see a graph provided by the seller, and on the right-hand side, customers can leave a comment on the item. For now, we do not implement an authentication mechanism, and we obviously do not verify the user to be the one that bought this T-shirt before. So basically, anyone can leave that comment. We will build an authentication and authorization system in a later chapter. We will just skip that part and focus on this chapter's topics here. Refer to the following figure:

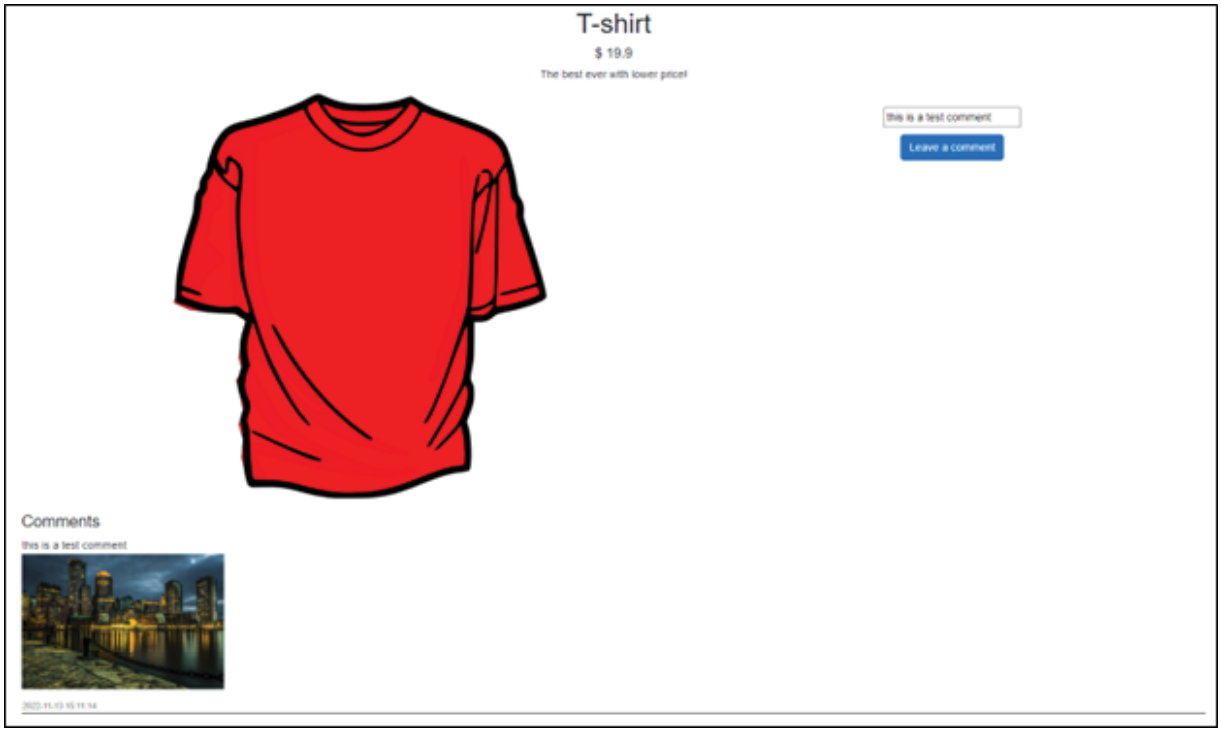

Figure ShopItem page

And finally, we added the new page to the navigation menu in NavMenu.razor under the Shared folder so that users can navigate the showing page with an easy click on the left navigation bar:

@* some code *@

class="nav-item px-3">

class="nav-link" href="cart">

class="oi oi-list-rich" aria-hidden="true"> Cart

class="nav-item px-3">

class="nav-link" href="shop-item">

class="oi oi-list-rich" aria-hidden="true"> ShopItem

@* some code *@

The skeleton of a shopping item displaying page is almost done. Except for the image uploading part. We will implement that now and see how the InputFile component that comes with Blazor can help us to achieve that.

We will remove the hard coded ImageUrl value and use the parameter from the constructor in

// some code

public Comment(string content, string imageUrl)

```
        {
            Content = content;
            ImageUrl = imageUrl;
            CreatedTime = DateTime.Now;
        }
// some code
```

Add the InputFile component in ShopItem.razor page and assign the method UploadImageAsync to handle the OnChange event of the InputFile component:

```
        @* some code *@
        type="text" @bind="Content"/>
        OnChange="@UploadImageAsync" />

class="btn btn-primary NewCommentBtn" @onclick="AddComment">Leave a comment

        @* some code *@
```

Write the method UploadImageAsync in Developers can read the file stream through the InputFileChangeEventArgs parameter and convert it to a Base64 string:

```
using Microsoft.AspNetCore.Components.Forms;

namespace EShop.Pages



{

    public partial class ShopItem

    {

        public Models.ShopItem Item { get; }

        public string? Content { get; set; }

        private string _base64Image = string.Empty;

        public ShopItem()

        {

            Item = new Models.ShopItem("T-shirt", "The best ever with lower price!", 19.9);
```

```
        Item.ImageUrl = "https://ts1.cn.mm.bing.net/th/id/R-
C.614bdee2065be0f1976bdf839c725e26?
rik=EJ2vSWnKs9a9vQ&riu=http%3a%2f%2fclipart-
library.com%2fimg%2f828773.png&ehk=avi5QwUJFS0v4Qtu8ggI5Ar
iopp4uJwf7r5QlOnJQ0o%3d&risl=&pid=ImgRaw&r=0";

    }

    private void AddComment()

    {

        System.Console.WriteLine("AddComment");

        if (string.IsNullOrWhiteSpace(Content))



        {

            return;

        }

        var comment = new Models.Comment(Content!,
_base64Image);

        Item.AddComment(comment);

        Console.WriteLine(Content);
```

```
        }

        private async void UploadImageAsync(InputFileChangeEventArgs
args)

        {

            using var stream = args.File.OpenReadStream();

            byte[] buffer = new byte[stream.Length];

            await stream.ReadAsync(buffer, 0, buffer.Length);

            _base64Image =

        }

    }

}
```

Now, run the application, go to the shop-item page and open the DevTools. In the Elements tab you will see HTML similar to Figure

Figure InputFile render to HTML

InputFile component will be rendered to an input element with the type attribute of To allow users to upload more than one file at once, you may add the attribute multiple to it. OnChange of the InputFile component was assigned a method with the parameter of type InputFileChangeEventArgs to handle the file(s) users uploaded.

We first call OpenReadStream to read from the uploaded image stream. By default, an exception will be thrown if the file is larger than 500KB. To override this behavior, you may provide a number to maxAllowedSize parameter of OpenReadStream method explicitly. In fact, it is not recommended to read the uploaded stream into memory directly, but for demo purposes, we are not following that advice here. But you should avoid that in your production code. And since we will cover saving data through EF core in the future chapter, we are only converting the uploaded image into a static base64 string to display on the web page. ContentType we used in the example, are the standard MIME type that is widely used in the browser. We rely on this type to convert the stream to a correct bas64 string. When uploading multiple files, you may iterate through the result of the method GetMultipleFiles from Each of them can be handled the same as the File property.

Tips

When you are expecting an image uploaded from users, just like the comment example above, you may find another method helpful.

Modify the method UploadImageAsync in ShopItem.razor.cs to use a new method RequestImageFileAsync on the IBrowserFile type:

```
    @* some code *@

    private async void
UploadImageAsync(InputFileChangeEventArgs args)

    {

        var file = await
args.File.RequestImageFileAsync(args.File.ContentType, 300, 500);

        using var stream = file.OpenReadStream();

        byte[] buffer = new byte[stream.Length];

        await stream.ReadAsync(buffer, 0, buffer.Length);
```

```
            _base64Image = $"data:{file.ContentType};base64,
{Convert.ToBase64String(buffer)}";

        }

        @* some code *@
```

RequestImageFileAsync may help you to resize the image from uploading or converting to another format when you provide a specific format. However, it should be noted that there is no guarantee that the converted IBrowserFile will be an image, or even that the conversion can be processed. Since the conversion is running in the JavaScript runtime, you may find it most suitable to call this method when you are building a Blazor WebAssembly application.

To avoid malicious attacks, first, scan, check, and validate all the files that are not provided by yourself before you work on them. In addition, you may also want to upload the customers' files from the client browser directly to a trusted third-party storage service. For example, Azure Storage is most appropriate for these situations, so that your application can safely process the file as simply a proxy and call the external storage services when you are required to show or use the files. The detail on how to upload to third parties will not be covered in this book. But since you are writing C# code, which is the benefit when you choose to build your web application with Blazor, you can investigate the third-party documents for backend code, and we hope they have a C# example. And for Azure, you are promised that they have one.

Finally, it is worth limiting the number of files that a user can upload at once. To achieve that, simply pass an explicit maximum when you are calling The default number is 10 but remember to change it when you have a less maximum in production.

Conclusion

In this chapter, we first introduced two popular protocols, HTTP and FTP, that we could leverage to share files between your server and clients. And then we went through a detailed example of how to upload files in Blazor and discussed a few recommendations and considerations that you may want to think about when dealing with file uploading.

In the next chapter, we will discuss the other way around, how to provide files to your customers in Blazor.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6

Serving and Securing Files in Blazor

Introduction

In the previous chapter, we introduced how a user can upload files to Blazor by using the InputFile razor component and discussed how to protect your application from cybersecurity attacks. Serving files to clients is crucial to your application, especially for those applications that are facing business customers. The requirements often include sales data files generated periodically, statistics downloading to understand more insights, or business intelligence reports. In this chapter, we will talk about how you can provide users with files, download files from your applications, and the security considerations the same when users upload files.

Structure

In this chapter, we will discuss the following topics:

Middlewares in ASP.NET Core

Serve Static Files

Serve Dynamic Files

Security Advice

Objectives

In this chapter, we will understand how the requests are handled in ASP.NET Core. The design of is the most important piece to get a whole picture of the ASP.NET Core running mechanism. Next, we will be introduced how to serve static files or dynamic files in Blazor. Static files are simply static assets saved on the local hard drive while dynamic files require to be generated runtime. We will then be discussing some basic security rules to protect servers from attacks when providing files to the clients.

Middleware

In Blazor Server, you could choose the static file middleware provided by ASP.NET Core to server static files to your users. Middleware is a new concept that was introduced into ASP.NET Core. It is a pipeline system that is like an assembly line in a manufacturing factory. All the requests from the clients will be fed into this pipeline system. Each middleware works as a worker in the assembly line to create, remove or modify requests and responses. Unlike an assembly line, the middleware pipelines not only pass the requests coming to the next middleware but also return the response to the previous one when it reaches the last middleware in the pipeline systems. In addition, each middleware may decide by itself whether or not to pass the requests to the next one. While in a factory, a worker generally cannot control who to work on the products next or where the products go in the assembly line. Figure 6.1 shows the workflow of a request in the middleware system:

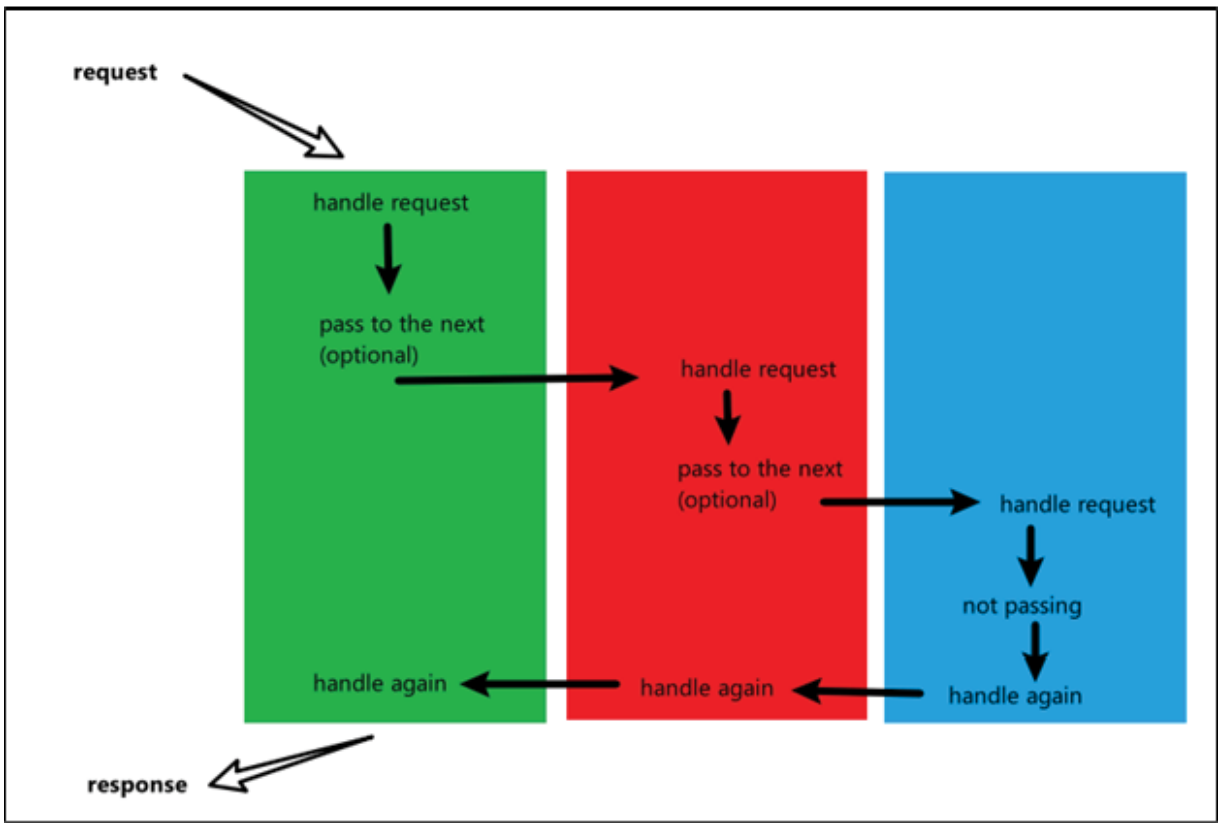

Figure ASP.NET Core middleware system

ASP.NET Core comes with a lot of default middlewares, for example, Static File Middleware is one of those. Other common middlewares include Redirection, routing, authentication, authorization, and many more. Even the API controller is implemented by middlewares. All the middlewares follow the very same processing rule illustrated by Figure These middlewares together synthesize the robust ASP.NET Core application.

To learn more about the middlewares, let's get hands dirty by implementing one of those. First, we need to create another EShop project based on Blazor Server, and we moved the EShop with Blazor WebAssembly to a subfolder WebAssembly. At the same time, we create another subfolder, Server, which will contain the code for EShop

in Blazor Server mode. To create a Blazor Server EShop, change the directory to the Server folder and run the command dotnet new blazorserver -n EShop.Server -o . (don't miss out on the dot sign). Now your code structure will look like the following code:

```
EShop

├———EShop.sln

└———.gitignore

└———.vscode

└———EShop.Server

├———Pages

├———Shared


├———wwwroot

├———EShop.Server.csproj

├———other folders and files

└———EShop.WebAssembly

├———Pages
```

```
├──Shared

├──wwwroot

├──EShop.WebAssembly.csproj

├──other folders and files
```

A Middleware can be implemented in two ways. The first and simplest way is to code an inline middleware. If you want to log every request coming from clients, you could easily use an inline middleware to achieve that. We can add the inline middleware in Program.cs file under the EShop.Server project to log every request:

```
using EShop.Server.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddServerSideBlazor();

builder.Services.AddSingleton();

var app = builder.Build();
```

```
app.Use(async (context, next) =>

{
    string log = $"
{context.Request.Scheme}://{context.Request.Host.Value}
{context.Request.Path.Value}";
    Console.WriteLine($"client requests: {log}");
    await next.Invoke();
});
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
```

```
app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.MapBlazorHub();



app.MapFallbackToPage("/_Host");

app.Run();
```

In the preceding code, we called the Use extension This method is used to add an inline middleware delegate to the pipeline system, and we passed a delegate to the method. The delegate has two parameters. The first is a context of the type which contains all the information related to the request, and the response as well. Another parameter is called next, which is the next delegate that will be executed in the pipeline system. It in fact points to the next middleware. So, all the middlewares are chained like a linked list only that this list is bi-directional, requests coming in and going out.

We print out the URLs that the clients requested in the debug console. Showed in Figure apparently, the client first visited our application with the http schema and was directed to port 5001 with a more secure https schema. Thanks to the it adds a middleware to redirect http requests to https and forced the clients to use https when they are visiting our

EShop application. After that, all the required static files and blazor components were loaded:

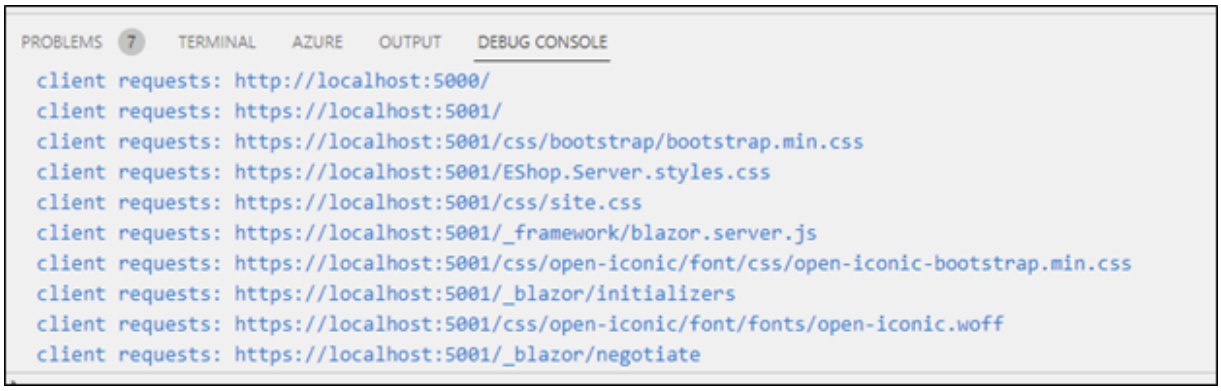

Figure 6.2: Inline log middleware

If you have a more complicated middleware, you might want a dedicated class to act as a middleware in the pipeline system. In fact, there is no such middleware base class provided by the asp.net core framework as long as a class meets two requirements. The first is a public constructor of which the first parameter has the type of The second is a public Invoke or InvokeAsync method of which the first parameter has the type of For example, we could move our logger logic into a class like the following code:

```
namespace EShop.Server;

public class LoggerMiddleware

{

    private readonly RequestDelegate _next;
```

```
    public LoggerMiddleware(RequestDelegate next)

    {

        _next = next;

    }

    public async Task Invoke(HttpContext context)

    {

        string log = $"
{context.Request.Scheme}://{context.Request.Host.Value}
{context.Request.Path.Value}";

        Console.WriteLine($"client requests: {log}");



        await context.Response.WriteAsync("hello world!");

    }

}
```

The preceding code shows an example of You may have more parameters for the constructor or Invoke method, but make sure that RequestDelegate or HttpContext comes first. This time in the middleware class, we are not calling next.Invoke() to short circuit other

middlewares followed behind. In this way, we revert the request handling direction and start to compose the response returned to the client. And the middlewares before this LoggerMiddleware may choose to alter the response when they see it necessary. Finally, the response will be received by the client from the very first middleware that handles the request.

To add this class of middleware into the pipeline system, we call the UseMiddleware extension method in It will register a middleware to the pipeline. Notice that the order we register a middleware is very important here because if a particular middleware chooses to short-circuit the request, all the middlewares registered after it will be ignored and will not handle the request:

```
using EShop.Server;

using EShop.Server.Data;

var builder = WebApplication.CreateBuilder(args);



// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddServerSideBlazor();

builder.Services.AddSingleton();

var app = builder.Build();
```

```
app.UseMiddleware();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

    app.UseExceptionHandler("/Error");

    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

    app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.MapBlazorHub();



app.MapFallbackToPage("/_Host");
```

app.Run();

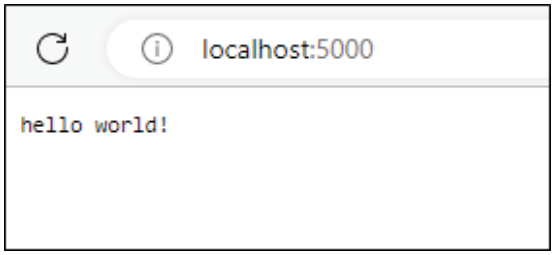

Figure Middleware short circuit

Figure 6.3 is a screenshot of our application when a user browses Since the LoggerMiddleware short circuit the request pipeline, this time, it will not show the familiar Blazor App home page with menus on the left. Instead, a simple plain text response is returned from the

If we move the LoggerMiddleware registration after:

```
// some other code

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

    app.UseExceptionHandler("/Error");
```

```
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.

    app.UseHsts();

}

app.UseHttpsRedirection();

app.UseMiddleware();

app.UseStaticFiles();

// some other code
```

Figure 6.4: Reorder LoggerMiddleware

Since LoggerMiddleware comes after when we browse to we are redirected to the corresponding https schema endpoint,

Serve Static Files

By default, when you create a new Blazor Server or Blazor WebAssembly application, serving static files is supported. But since we short-circuit all the rest middlewares, let's allow the request to move down through the pipeline system. We update the LoggerMiddleware.cs to remove the short-circuit:

```
namespace EShop.Server;

public class LoggerMiddleware

{

    private readonly RequestDelegate _next;

    public LoggerMiddleware(RequestDelegate next)

    {

        _next = next;

    }

    public async Task Invoke(HttpContext context)
```

```
    {

        string log = $"
{context.Request.Scheme}://{context.Request.Host.Value}
{context.Request.Path.Value}";

        Console.WriteLine($"client requests: {log}");

        // await context.Response.WriteAsync("hello world!");

        await _next.Invoke(context);



    }

}
```

So, we call the Invoke method on the RequestDelegate to pass the request to the next middleware:

world!

-- this is a static file.

And we add a text file hello.txt under the wwwroot folder with the content above. All the files under wwwroot will be served, and you

may already find out that all the CSS files are placed under this folder as well:

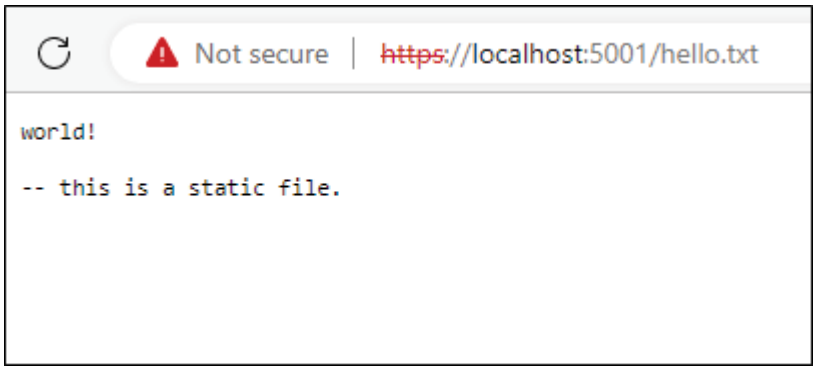

Figure Serve static files under wwwroot

Now you start the server and browse to and the web browser will display the content from the file Similarly, if you navigate to a CSS file, it will display the style configuration as well.

Sometimes, your files may not be under this default wwwroot files. For example, you have a dedicated assets folder to keep all the graphs or logos together. And the browser will complain that such a file resource cannot be found, which is a 404-error status code.

To support the files under any customized directory, you may config the UseStaticFiles middleware in Program.cs to achieve that:

```
using EShop.Server;

using EShop.Server.Data;
```

```
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddServerSideBlazor();

builder.Services.AddSingleton();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

    app.UseExceptionHandler("/Error");



    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

    app.UseHsts();
```

```
}

app.UseHttpsRedirection();

app.UseMiddleware();

app.UseStaticFiles(new StaticFileOptions

{

    FileProvider = new PhysicalFileProvider(

        Path.Combine(builder.Environment.ContentRootPath,
"Assets")),

    RequestPath = "/assets-url"

});

app.UseRouting();

app.MapBlazorHub();

app.MapFallbackToPage("/_Host");

app.Run();
```

In the preceding code, we provide the static files middleware a StaticFileOptions instance. It tells the middleware where we put the static files and how a client can request them. The first assets string in PhysicalFileProvider identifies the physical path of the files folder. And the second assets string assigned to RequestPath represents the URL path that a user should navigate to get the file. In our example, the user would call https://localhost:5001/assets-url/Logo_Square.png to get the logo placed under the Assets folder:

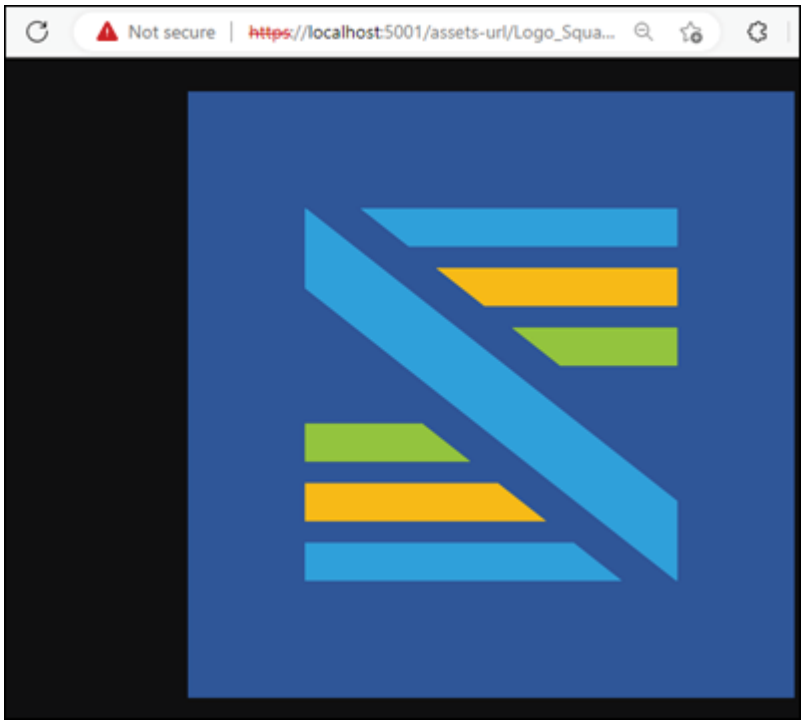

Figure 6.6: Serve files under another directory

It works now that we can place files under any folders we desired. However, if you browse the home page, you will find out that it is a plain text page, with no color, no font size, and no style. This is because moving the static files directory from wwwroot to and the web application cannot find those CSS and JavaScript files under the Assets folder anymore:

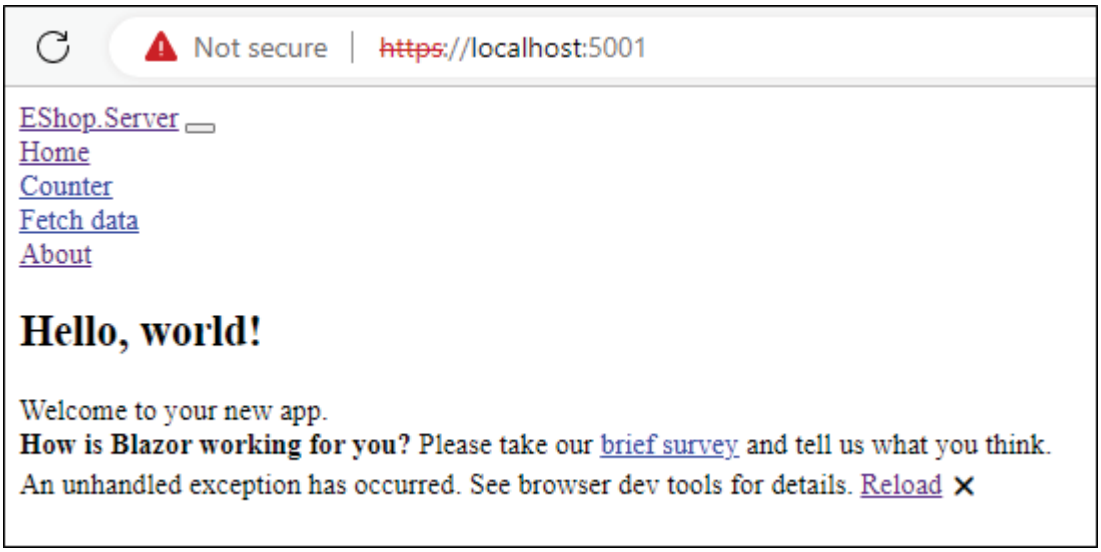

Figure Missing CSS and JavaScript files

But we can easily fix it. Leave the default UseStaticFiles method and call it again in the next line with our customized files directory. In this way, you can serve any many directories as you want.

Serve Dynamic Files

In the last section, we discussed how to serve static files to customers. Static files are files that will not change, for example, website logo, or website license claim. However, when your customer requires a statistical report for the last month, it is impossible for developers to prepare such a file in advance and put it under a certain directory. These are dynamic files, and they are generated only when it was requested, or at least they will change over time. To serve a dynamic file, we need help from JavaScript. Add a new eshop.js under wwwroot folder with the following code:

```
function download() {

  const current = new Date();

  const day = current.getDate()

  const month = current.getMonth() + 1

  const year = current.getFullYear()

  const time = year + "/" + month + "/" + day + " " +
current.getHours() + ":" + current.getMinutes() + ":" +
current.getSeconds();
```

```
  const data = 'hello world!' + "\n" + time;

  const blob = new Blob([data]);

  const url = URL.createObjectURL(blob);

  const anchorElement = document.createElement('a');



  anchorElement.href = url;

  anchorElement.download = 'hello.txt';

  anchorElement.click();

  anchorElement.remove();

  URL.revokeObjectURL(url);

}
```

In the download function, we first get the current date and time from the Date object. Then combined with "hello every time a client requests this document, we can generate the content based on the current time. The rest of the function creates an anchor element, which will navigate to the URL and represent the file data through a Blob object. Finally, we invoke the click event on the anchor

element, remove the element from the DOM and revoke the URL so that the browser will not keep the reference any longer:

```
html>

lang="en">




id="app">Loading...


id="blazor-error-ui">
    An unhandled error has occurred.
    href="" class="reload">Reload
```

Next, we reference this JavaScript file globally in the index.html so that every page in our EShop application can call this download function just like what you would normally do in any JavaScript project:

@page "/shop-item"

@using EShop.WebAssembly.Shared;

class="ItemContainer">

@Item.Name

$ @Item.Price

@Item.Description

class="ItemDisplay">

src="@Item.ImageUrl" />

class="NewComment">

type="text" @bind="Content"/>

OnChange="@UploadImageAsync" />

class="btn btn-primary NewCommentBtn"
@onclick="AddComment">Leave a comment

onclick="download()">Download

class="Comments">

Comments

```
    @foreach(var comment in Item.Comments)
    {
        CommentModel="@comment" />
    }
```

On the ShopItem page, we add a button to download this hello world file with dynamic date and time. In a production application, JavaScript may depend on the user's input or data from other sources to generate the dynamic file. This would require the Blazor application to pass the input or data to JavaScript. And it will be

accomplished by the interop between Blazor and JavaScript which will be covered in a later chapter.

The previous example represents downloading from a stream that is recommended to serve a file not larger than 250MB. If you are providing a file larger than 250MB to the customers, a better way is to download it from a URL. The steps are basically the same as before, but you will provide the file name and URL directly to the JavaScript. The file could in fact from an external source, or, if you are implementing a micro-service architecture, from another service in your clusters.

Security Advice

When you are serving files, it is recommended that files downloaded are located separately from the system files, or code files on the server, as a hacker may acquire the execute permissions on the directory. Placing all the files together prohibits you from applying different security control policies on different types of files.

Don't forget to revoke the URL; otherwise there may be memory leaks to the clients. And make sure to perform security checks on the server when you are interacting with customers. Even when you have applied validation or checks on the client side, say, in JavaScript, keep in mind that you need to double-check again as client validations can easily break through.

If you are providing files through any external sources, remember to run a security scan before you pass them to the customers. In general, the more you don't trust other partners, the more security you will get.

Conclusion

In this chapter, we first went through the design of the middlewares pipeline system as a streamline to handle clients' requests. And we understand that UseStaticFiles is the default middleware to serve static files to clients, and it can be configured to serve files in different locations. After that, we covered how to serve dynamic files as not all required data can be generated before your application goes online. Last, we discussed how to make your server more secure to protect your applications.

In the next chapter, we will cover one of the most important components or elements that you will regularly interact with customers, forms. We will introduce how to collect data from users with forms, and how to validate the input from users.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7

Collecting User Input with Forms

Introduction

In this chapter, we will introduce how a user could provide information to our application and how we will validate the user's input with customized rules. The form has always been a classic element for web UI, and it is a bridge that connects users and applications with two-direction communications.

Structure

In this chapter, we will discuss the following topics:

EditForm

InputBase

Validation

Custom validation

Form submission

EditContext

Form state

Objectives

In this chapter, we will understand how to create forms with Blazor and how to validate users' data with default or customized rules and prompt validation errors if data do not satisfy the rules. We will also be understanding the key concepts in blazor forms, including submission, context, and state.

Forms

In a web application, forms are used to collect users' input. A typical example will be the register/login page, where a user types in his or her username and password to authenticate, and almost all the web applications in the world have this feature. A native HTML element works as a container for all kinds of html input elements, including text input, radio buttons, or even files.

In Blazor, you could always use the native element to manage users' input data as you did without Blazor. However, it is recommended to use the EditForm comes with Blazor with more convenient and advanced data management.

In our EShop application, when there are no clothes of a customer's size, the customer can request his or her size by filling out a request form, and our EShop staff will refill the stocks. Let's first see how to implement this feature with native HTML elements.

First, we create a new page file RequestItem.razor, under the Pages folder of EShop.WebAssembly project. Then we add the newly created page to the navigation menu. Find NavMenu.razor under the Shared folder in the project and update the element:

@* some code *@

class="flex-column">

class="nav-item px-3">

class="nav-link" href="" Match="NavLinkMatch.All">

class="oi oi-home" aria-hidden="true"> Home

class="nav-item px-3">

class="nav-link" href="counter">

class="oi oi-plus" aria-hidden="true"> Counter

class="nav-item px-3">

class="nav-link" href="fetchdata">

class="oi oi-list-rich" aria-hidden="true"> Fetch data

class="nav-item px-3">

class="nav-link" href="cart">

class="oi oi-list-rich" aria-hidden="true"> Cart

class="nav-item px-3">

class="nav-link" href="shop-item">

class="oi oi-list-rich" aria-hidden="true"> ShopItem

class="nav-item px-3">

class="nav-link" href="request-item">

class="oi oi-list-rich" aria-hidden="true"> RequestItem

@* some code *@

We added another
element navigating to the request-item form page. Now let's complete the new page RequestItem.razor with the native element:

@page "/request-item"

Request Item

Please fill in the request form below if you do not find desired item.

Native form element

for="item">Item:

type='text' name='item' id='item' required>

for="size">Size:

value="s">Small

for="count">Count:

type='number' name='count' id='count' required min="1" max="10">

type='submit' />

We are using a plain form here, without any customized style, so the form would not be that attractive. But it is good enough to demonstrate the core feature provided by the native element. For example, some fields are marked as required, and the form will not be submitted if these fields are empty. We also add a min and max validation to verify that the count value must be between 1 and 10. Refer to the following figure:

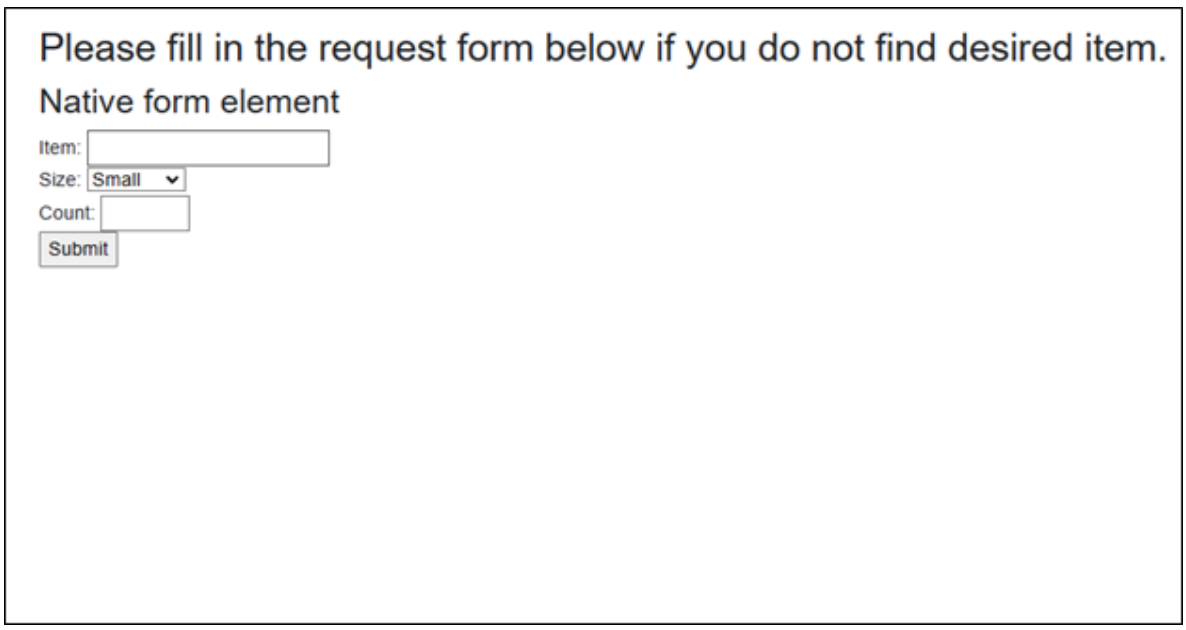

Figure Native HTML form

Next, we will compare it with a form generated by the EditForm component that comes with the Blazor framework.

EditForm

The source code of EditForm is located at and we will show some of the most important sections here:

```
public class EditForm : ComponentBase

{

    private EditContext? _editContext;

    [Parameter]

    public EditContext? EditContext

    {

        get => _editContext;

        set

        {

            _editContext = value;
```

```
        _hasSetEditContextExplicitly = value != null;
    }
}
[Parameter] public object? Model { get; set; }

[Parameter] public EventCallback OnSubmit { get; set; }
[Parameter] public EventCallback OnValidSubmit { get; set; }
[Parameter] public EventCallback OnInvalidSubmit { get; set; }
protected override void OnParametersSet()
{
    if (Model != null && Model != _editContext?.Model)
    {
        _editContext = new EditContext(Model!);
    }
}
```

```
    protected override void BuildRenderTree(RenderTreeBuilder
builder)

    {

        Debug.Assert(_editContext != null);

        builder.OpenRegion(_editContext.GetHashCode());

        builder.OpenElement(0, "form");

        builder.AddMultipleAttributes(1, AdditionalAttributes);



        builder.AddAttribute(2, "onsubmit", _handleSubmitDelegate);

        builder.OpenComponent>(3);

        builder.AddAttribute(4, "IsFixed", true);

        builder.AddAttribute(5, "Value", _editContext);

        builder.AddAttribute(6, "ChildContent",
ChildContent?.Invoke(_editContext));

        builder.CloseComponent();

        builder.CloseElement();

        builder.CloseRegion();
```

```
}

private async Task HandleSubmitAsync()

{

    Debug.Assert(_editContext != null);

    if (OnSubmit.HasDelegate)

    {

        await OnSubmit.InvokeAsync(_editContext);

    }

    else



    {

        var isValid = _editContext.Validate();

        if (isValid && OnValidSubmit.HasDelegate)

        {

            await OnValidSubmit.InvokeAsync(_editContext);
```

```
            }

            if (!isValid && OnInvalidSubmit.HasDelegate)

            {

                await OnInvalidSubmit.InvokeAsync(_editContext);

            }

        }

    }

}
```

We first learned, that the EditForm inherits ComponentBase as many other Blazor components do. In the method (we introduced this method in previous chapter about the advanced Blazor component), EditForm uses the element under the hood as well. Then it defined more attributes on the form and cascading the EditContext to its child components.

The most important parameters of this EditForm component are the Model and the The Model could be of any types, while EditContext has a dedicated EditContext type. In general, we will only one of these two parameters. If we assigned a model to the Model parameter, a new EditContext will be generated with a simple EditContext constructor that requires one parameter of the model. Otherwise, we

are required to assign an EditContext instance directly to the EditContext parameter. In both ways, the EditForm component will have an EditContext parameter that is not null and cascade it to the children.

Now, let us see how to replace the native element with the Blazor EditForm component. By default, the namespace Microsoft.AspNetCore.Components.Forms will be imported in the _Imports.razor file, or you could specify this namespace in your razor file with the @using directive.

First, we create a new file RequstItem.cs, under the Models folder for the RequestItem model:

```
namespace EShop.WebAssembly.Models;

public enum RequestItemSize

{

    Small,

    Medium,

    Large

}



public class RequestItem
```

```
{

    public string EShopItemName { get; set; }

    public RequestItemSize Size { get; set; }

    public int Count { get; set; }

}
```

Then we will use the EditForm component in the RequestItem.razor page, adding the following code to the page:

EditForm component

```
Model="@_requestItem">




            Item:

            id="editform-item" @bind-
Value="_requestItem.EShopItemName" />
```

Size:

id="editform-size" @bind-Value="_requestItem.Size">

value=@RequestItemSize.Small>@RequestItemSize.Small

value=@RequestItemSize.Medium>@RequestItemSize.Medium

value=@RequestItemSize.Large>@RequestItemSize.Large

Count:

id="editform-count" @bind-Value="_requestItem.Count" />

type="submit">Submit

Refer to the following figure:

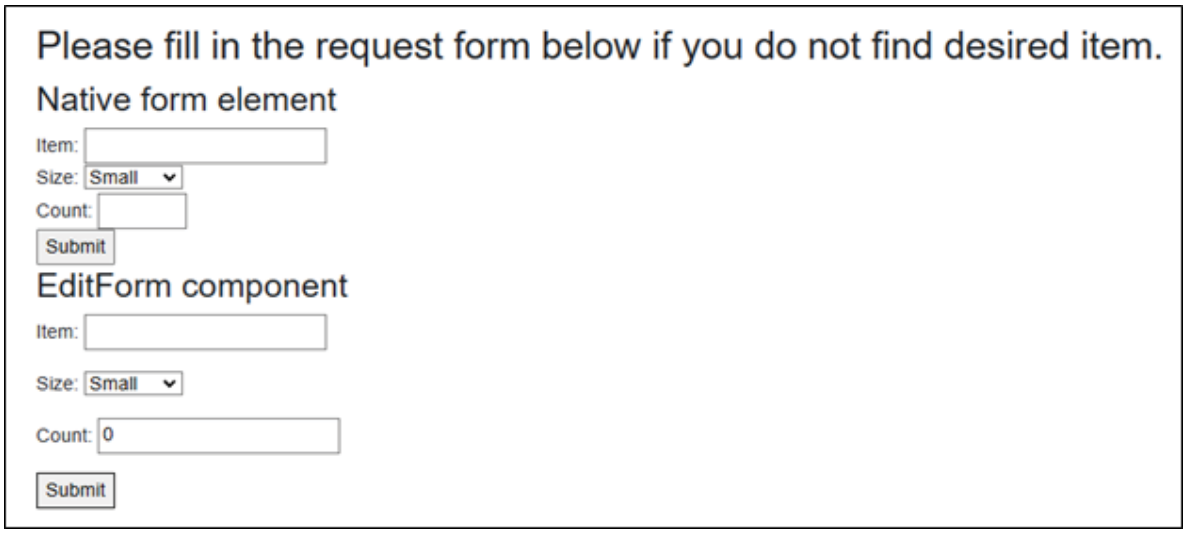

Figure EditForm

We are not validating the user's input here, as it will be covered in later sections. In general, it is recommended to use the EditForm component because it provides more features than native element. And in the following sections, we will cover these features.

InputBase

In the last simple example, you may notice that Blazor not only provides the EditForm component to enhance the development experience, but also comes with a few InputXXX components to replace different types of the native element. For example, we used InputSelect and InputNumber to replace  type='text'  and  type='number' These are all components inheriting

InputBase is an abstract class sits within the same namespace with It has a private property of type EditContext to receive the cascaded parameter from its parent Other than that, it has a Value parameter of generic type TValue to be used as the value of this input, along with a ValueChanged callback that will update the bound value. It is intended to use the Value parameter with two-way binding, and you may refer to the previous chapter where we covered this topic.

InputSelect component is used to provide a few options to the application users. In our EShop example, people will choose their desired size of the clothes with this component. But the code we wrote here does not really follow best practice design principles. Once we are required to extend the size options, both the code of the enumeration and the form page with InputSelect

component must be modified. There is a good chance that we might miss one of the modifications here when you are facing a complex feature and so the inconsistency comes.

To fix that, we can build a customized component inheriting InputBase as well that all it needs is the type of your enum and it will automatically show all the available options come from that enum type.

We create a new file InputEnum.cs under the Components folder:

```
using System.Diagnostics.CodeAnalysis;

using System.Globalization;

using Microsoft.AspNetCore.Components;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Components.Rendering;

namespace EShop.WebAssembly.Components;

public class InputEnum : InputBase

{
```

```
    protected override bool TryParseValueFromString(string?
Value, [MaybeNullWhen(false)] out Tenum result,
[NotNullWhen(false)] out string? validationErrorMessage)

    {

        if (string.IsNullOrWhiteSpace(value))

        {

            result = default;



            validationErrorMessage = $"{nameof(value)} cannot be
null";

            return false;

        }

        if (Enum.TryParse(typeof(Tenum), value, out object
convertedEnum))

        {

            result = (Tenum)convertedEnum!;
```

```
            validationErrorMessage = null;

            return true;

        }

        result = default;

        validationErrorMessage = $"{nameof(value)} is not valid";

        return false;

    }

    protected override void BuildRenderTree(RenderTreeBuilder
builder)

    {

        builder.OpenElement(0, "select");



        builder.AddMultipleAttributes(1, AdditionalAttributes);

        builder.AddAttribute(2, "onchange",
EventCallback.Factory.CreateBinder(this, value =>
CurrentValueAsString = value, CurrentValueAsString, null));
```

```
        // Add an option element per enum value
        foreach (var value in Enum.GetValues(typeof(Tenum)))
        {
            builder.OpenElement(3, "option");
            builder.AddAttribute(4, "value", value.ToString());
            builder.AddContent(5, value.ToString());
            builder.CloseElement();
        }
        builder.CloseElement(); // close the select element
    }
}
```

First, we override the abstract method which will parse the input value to an enumeration instance. Next, we also override the method BuildRenderTree to define how the component is rendered. In fact, InputEnum wraps the native