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 chapters**. 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 Introduction- 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 Model- 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 Components- 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 Enhancement- 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 Blazor- 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 Blazor- 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 Forms- 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 Application- 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 Interop- 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 HTTP- 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 Core- 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 Identity- 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 Kubernetes- 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 **https://github.com/bpbpublications/Building-Modern-Web-Applications-with-ASP.NET-Core-Blazor**. 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 **https://github.com/bpbpublications**. 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.

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 **www.bpbonline.com**. 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 **www.bpbonline.com**.

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

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 *Assembly*, we might get a better understanding of WebAssembly. *Web*, 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 *Amazon.com*, watch videos on *Youtube.com* and check out how friends' life is on *Facebook.com*.

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 *Java*. 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 **`my-hello-world-demo`**:

```
#include <stdio.h>


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.html**, **hello.js** and **hello.wasm**, 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 **https://www.python.org**. Now, open your command line or terminal and move to **my-hello-world-demo** and enter **python -m http.server**, python will start a server listening on port **8000**. Open your browser and navigate to **http://localhost:8000**, it will show the files under **my-hello-world-demo,** then click **hello.html**, a default frontend page provided by **emscripten** will show. Refer to *Figure 1.1*:

***Figure 1.1**: 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 <emscripten.h>

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 **`http://localhost:8000/function.html`**. 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 **`function.js`**:

```
<script type='text/javascript'>
    var statusElement = document.getElementById('status');
    var progressElement = document.getElementById('progress');
    var spinnerElement = document.getElementById('spinner');

    var Module = {
        preRun: [],
        postRun: [],
        print: (function() {
            var element = document.getElementById('output');
            if (element) element.value = ''; // clear browser cache
            return function(text) {
                if (arguments.length > 1) text = Array.prototype.slice.
call(arguments).join(' ');
                console.log(text);
                if (element) {
                    element.value += text + "\n";
                    element.scrollTop = element.scrollHeight; // focus
on bottom
                }
            };
        })(),
        canvas: (function() {
           // draw canvas
        })(),
        setStatus: function(text) {
```

```
            // set status
        },
        totalDependencies: 0,
        // some code here
    };
    Module.setStatus('Downloading...');
    // some code here
</script>
<script async type="text/javascript" src="function.js"></script>
```

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, **`setStatus`**. 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 **Downloading...**. And you might already guess it. It is downloading the WebAssembly file, **`function.wasm`**. 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 **function.js**, 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, **function.wasm**.

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 200**. Refer to *Figure 1.2*:

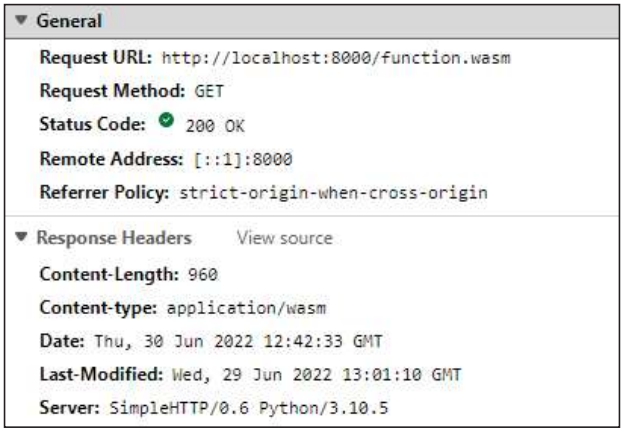

Figure 1.2: fetch function.wasm

`WebAssembly.instantiateStreaming()` will be responsible for compiling and initiating the WebAssembly module, and it will be more efficient than load wasm code directly by **`WebAssembly.instantiate()`**. 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 **`instantiateArrayBuffer()`**, and the name of the function indicates that it is load the binary format of **`.wasm`** 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 **`createExportWrapper()`**, and it will find the exported functions by name in **`Module['asm']`**. 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 **`function.wasm`**? Can we manually load it?

Introducing .WASM binary format

Let us try another example.

```
#include <emscripten.h>

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                                                  ..
```

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 9th 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.

Id	Section
0x00	custom section
0x01	type section
0x02	import section
0x03	function section
0x04	table section
0x05	memory section
0x06	global section
0x07	export section
0x08	start section
0x09	element section