**WebAssembly Simplified: A Beginner’s Guide to Creating Your First WASM Project with Rust**

[[Zhihui](https://medium.com/@imvihv?source=post_page-----458091fdde57--------------------------------)](https://medium.com/@imvihv?source=post_page-----458091fdde57--------------------------------)

[[Stackademic](https://blog.stackademic.com/?source=post_page-----458091fdde57--------------------------------)](https://blog.stackademic.com/?source=post_page-----458091fdde57--------------------------------)

[Zhihui](https://medium.com/@imvihv?source=post_page-----458091fdde57--------------------------------)

·

Follow

Published in

[Stackademic](https://blog.stackademic.com/?source=post_page-----458091fdde57--------------------------------)

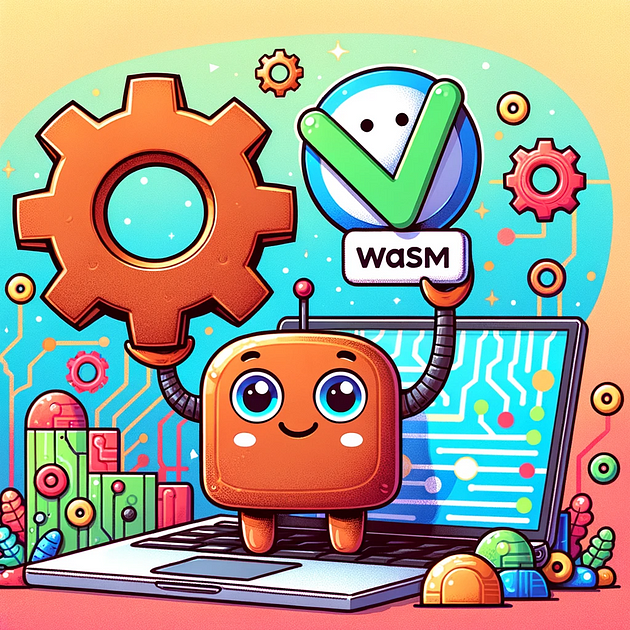
·

8 min read

·

Nov 22

1



Hello everyone, I’m Zhihui, and this is my first article. In it, I want to share how to quickly create a WebAssembly (WASM) project.

You may have heard of wasm and know that Rust can compile into wasm, but have you actually tried it? Today, let’s go through this process together. It’s really very simple. So, without further ado, let’s get started.

Firstly, I’ll assume you’re a Rust developer with Rust already installed on your computer. If not, please install Rust first.

Our core process involves two steps:

1. Create a wasm project using wasm-pack.
2. Create a front-end project with Vite, install the vite-plugin-wasm plugin, and include our wasm project in the dependencies.

**What is wasm-pack? Why do we need it?**

wasm-pack is a convenient toolkit designed to be a one-stop solution for building and utilizing Rust-generated WebAssembly. It handles the tedious tasks like creating a Rust project, compiling to WebAssembly, generating a JavaScript package, exporting functions from WebAssembly, and even publishing to npm.

First, you need to install wasm-pack. It’s very simple. Entering the following command in the terminal:

cargo install wasm-pack

After a bit of compiling, wasm-pack be installed.

By the way, if you have cargo-binstall, you can install the wasm-pack binary directly with the binstall command, avoiding the need for compilation and speeding up the process.

cargo install cargo-binstall # if you don't have cargo-binstall  
cargo binstall wasm-pack

Next, find a clean directory to start creating a wasm project. Name it something like “learn-wasm”, and execute this command:

wasm-pack new learn-wasm

This creates a learn-wasm folder in your current directory. If you cd into it and examine the structure, you'll find:

learn-wasm  
├── Cargo.toml  
├── LICENSE\_APACHE  
├── LICENSE\_MIT  
├── README.md  
├── src  
│ ├── lib.rs  
│ └── utils.rs  
└── tests  
 └── web.rs

It’s a standard Rust lib project, similar to what cargo new --lib learn-wasm would create, but with different content.

Let’s examine the lib.rs file:

mod utils;  
​  
use wasm\_bindgen::prelude::\*;  
​  
#[wasm\_bindgen]  
extern "C" {  
 fn alert(s: &str);  
}  
​  
#[wasm\_bindgen]  
pub fn greet() {  
 alert("Hello, learn-wasm!");  
}

The first line imports the utils module, containing some utility functions.

On the second line, we see the import of the wasm\_bindgen crate, a very important crate providing functionality for wasm and js interaction, allowing rust and js written code to call each other, acting as a bridge between the two.

The #[wasm\_bindgen] on the third line is a procedural macro that handles all the necessary glue code to ensure correct interaction between functions in WebAssembly and JavaScript. This macro is handy when you want to use js functions in rust, or vice versa.

The extern block that follows is an extern block, commonly known as an external block, used to declare the signatures of external functions.

We all know that JavaScript has an alert function that takes a string argument and pops up a dialog box.

We want to call this function in Rust, but Rust doesn’t know that JavaScript has this function, so we need to declare it here. This way, Rust will know about it. After that, we can use alert in Rust.

The extern block is followed by the "C" keyword, which is an ABI. ABI stands for Application Binary Interface. It defines the calling convention for functions. Here we are using C ABI, which is the calling convention for the C language.

We can ignore the internal principles for now and treat this as a common writing style. When we need to use which JavaScript function, we declare it here.

Next, a function named greet is defined. This function takes no arguments and has no return value. The internal code simply calls the alert function to pop up a dialog box and greet us.

Since the greet function is also annotated with the wasm\_bindgen macro, it will be exported and can be called by JavaScript.

Okay, now that we understand the code here, we can take a look at how this project is compiled to WebAssembly and how JavaScript calls it.

To compile the project using wasm-pack, simply execute the following command:

wasm-pack build

After compilation, you will see many more files in the current folder. We only need to pay attention to the pkg directory, which contains the compiled WebAssembly project. The file with the .wasm suffix is the actual compiled and optimized WebAssembly binary file, and the other files are glue code needed for the WebAssembly to work.

Thanks to wasm-pack, without it, we would have to do all the tedious work of writing glue code ourselves.

pkg  
├── README.md  
├── learn\_wasm.d.ts  
├── learn\_wasm.js  
├── learn\_wasm\_bg.js  
├── learn\_wasm\_bg.wasm  
├── learn\_wasm\_bg.wasm.d.ts  
└── package.json

In fact, when we execute wasm-pack build, wasm-pack first calls cargo to compile the project, the specific command is cargo build --release --target wasm32-unknown-unknown (yes, the default is release level, not debug).

Then it will call wasm-opt to optimize the generated WebAssembly binary file to reduce file size and improve performance.

Finally, it will generate JavaScript binding code, which acts as a bridge between WebAssembly and JavaScript, allowing JavaScript to call functions in WebAssembly.

wasm-pack will also generate a package.json file, which allows the package to be published to npm or imported as a dependency by other JavaScript projects.

Check package.json:

{  
 "name": "learn-wasm",  
 "collaborators": ["jlvihv <imvihv@gmail.com>"],  
 "version": "0.1.0",  
 "files": [  
 "learn\_wasm\_bg.wasm",  
 "learn\_wasm.js",  
 "learn\_wasm\_bg.js",  
 "learn\_wasm.d.ts"  
 ],  
 "module": "learn\_wasm.js",  
 "types": "learn\_wasm.d.ts",  
 "sideEffects": ["./learn\_wasm.js", "./snippets/\*"]  
}

We are particularly interested in these two lines:

"module": "learn\_wasm.js",  
"types": "learn\_wasm.d.ts",

The module field specifies the ES Module (ESM) entry point for this package, and the types field specifies the type definition file for this package.

Take a quick look at the code in these files. We don’t need to worry about the details for now, so the core function is reduced to just one line:

export function greet(): void;

That is, the greet function in WebAssembly is exported.

Okay, now we have a JavaScript package, the core functionality of which is implemented in WebAssembly, although it only has one function, it can be imported by other JavaScript projects.

Next, let’s create a front-end project to use this WebAssembly.

You can use webpack or vite to create a front-end project. Since the example in the official wasm-pack documentation uses webpack, we will use vite to show the difference.

For vue, we execute the following command in the current directory (learn-wasm directory):

pnpm create vue

We’ll name the vue project “web”, select no for all other options, and hit enter all the way to create a vue project. Then, execute cd web, pnpm install, and pnpm run dev in sequence to start the project and see that it has successfully launched.

Next, we need to install the vite-plugin-wasm plugin, which allows us to use WebAssembly in a vite project.

pnpm add -D vite-plugin-wasm

Add the plugin in vite.config.js (or vite.config.ts, if you are using ts):

import wasm from "vite-plugin-wasm";  
​  
export default defineConfig({  
 plugins: [wasm()],  
});

Then, we manually add a dependency in package.json, in the dependencies block, adding our WebAssembly package:

"dependencies": {  
 "learn-wasm": "file:../pkg"  
 },

Note that the path here is relative to package.json,so if your WebAssembly package is in a different directory, you need to modify the path.

After adding the dependency, we need to run pnpm install again.

After completing these tasks, we can call the function in wasm. Remember the greet function that pops up a dialog box to say hello?

Just call the wasm package like any other package, for instance, in App.vue, call the greet function:

import \* as wasm from "learn-wasm";  
wasm.greet();

Restart the project using pnpm run dev. You should see the dialog box pop up in the browser. If not, check if you skipped any of the steps above.

So far, we have completed the entire process of creating and using a wasm project. Congratulations, you’ve learned something new, and it’s really very simple.

You might say this project is too simple, with only one function calling alert, lacking technical depth. You may want a more complex project, like a calculator or a todo list, to better understand the use of wasm.

Don’t rush, a journey of a thousand miles begins with a single step. Let’s take it one step at a time and make progress bit by bit.

Next, we modify the wasm module, replacing alert with console.log.

It’s not difficult. We only need to declare console.log in the extern block of lib.rs, and then call it in the greet function:

However, it’s important to note that console in console.log is actually a global object in JavaScript, and log is a method of this object. Therefore, we need to change the signature of console.log to this in Rust:

#[wasm\_bindgen]  
extern "C" {  
 #[wasm\_bindgen(js\_namespace = console)]  
 fn log(s: &str);  
}

When we call it in Rust, we don’t use console.log(), but log() instead:

#[wasm\_bindgen]  
pub fn greet() {  
 log("Hello, learn-wasm!");  
}

We can also output the value of variables in the log() function, just as we do in Rust:

#[wasm\_bindgen]  
pub fn greet(name: &str) {  
 log(&format!("Hello, {}!", name));  
}

It’s important to note that the format!() macro returns a String type. To pass this to the log() function, we need to use & to create a reference.

In interactions between Rust and JavaScript, it’s preferable to use the &str type instead of String. This is because wasm\_bindgen can directly map &str from Rust to strings in JavaScript, and &str is more convenient as it avoids the ownership issues present with String.

However, sometimes, for example, when defining structures, you must use String. This is because &str can lead to lifetime issues — a common but sweet problem in Rust.

Alright, once we’ve modified lib.rs, let's recompile the wasm package:

wasm-pack build

Next, we need to clear the vite cache to ensure our recompiled wasm package is effective and not hindered by vite's cacheing.

rm -rf node\_modules/.vite  
pnpm install

Before running pnpm run dev, we also need to make some changes to App.vue, as the greet function now requires an argument.

import \* as wasm from "learn-wasm";  
wasm.greet("world!");

Now everything is set. Start the project, and use shift + F5 to forcefully refresh the page. You should see Hello, world! output in the browser console.

If you find this process a bit tedious, we can write a script for the above steps. This way, every time you modify the wasm module, you only need to execute a command to recompile the wasm package, clear the Vite cache, and restart the project.

Let’s modify the package.json file in the web project, changing the dev command in the scripts block to this:

"scripts": {  
 "dev": "wasm-pack build && rm -rf node\_modules/.vite && pnpm install && vite",  
 },

That’s it. Make some small changes to wasm, and give it a try.

That’s all for this article. If you found it helpful, please give me a like or share it with your friends. I will continue to share more content about Rust and wasm. If you’re interested, feel free to follow me. Thank you.

**Stackademic**

*Thank you for reading until the end. Before you go:*

* *Please consider****clapping****and****following****the writer! 👏*
* *Follow us on*[***Twitter(X)***](https://twitter.com/stackademichq)*,*[***LinkedIn***](https://www.linkedin.com/company/stackademic)*, and*[***YouTube***](https://www.youtube.com/c/stackademic)***.***
* *Visit*[***Stackademic.com***](http://stackademic.com/)*to find out more about how we are democratizing free programming education around the world.*