A simple WebAssembly linker in JavaScript

Radu Matei

November 13, 2020

This article originally appeared on the DeisLabs blog.

Over the last few months, our team has been experimenting with executing WebAssembly modules in various environments, and we released projects such as Krustlet (for executing Wasm modules in Kubernetes), and WAGI (for writing HTTP response handlers in WebAssembly). But while most of our experiments are built in Rust, we are also exploring ways to execute WebAssembly workloads in other environments, such as JavaScript runtimes.

Today, we are releasing a small experimental JavaScript library to help with instantiating WebAssembly modules by providing functionality to link JavaScript objects as module imports, and automatically perform name based resolution for linking entire modules and instances. The library also enables optionally defining asynchronous import functions, by leveraging a set of great open source projects, such as Binaryen and Asyncify.

Given that most of our WebAssembly experiments are built with Wasmtime, the API of the JavaScript linker we are releasing is modeled after the excellent Wasmtime linker, which offers a similar functionality for Rust runtimes, and using the JavaScript linker should be familiar to people using Wasmtime.

Using the JavaScript linker

First, add the NPM package to your project:

```
$ npm install @deislabs/wasm-linker-js
```

Then, using the `Linker` is as easy as importing it using your favorite module system:

```
import { Linker } from "@deislabs/wasm-linker-js";
OR
const { Linker } = require("@deislabs/wasm-linker-js");
```

The next sections contain a few examples of using the JavaScript linker. In order to show the text format of the WebAssembly modules we are trying to use, `binaryen` is also required to run the examples (`npm install binaryen`). This is not necessary in real world scenarios, and the modules can be compiled from their binary representation without additional dependencies. For more examples of using the Linker in both TypeScript and JavaScript, check the linker tests and the Node.js examples from the project's repository on GitHub.

Defining a single import

Let's assume we have a simple WebAssembly module (represented in its text format and contained in the `usingAdd` variable, transformed to its binary representation using Binaryen) that imports a function `add` from a module `calculator`.

In order to instantiate this module, we have to satisfy its import - in this case, provide an implementation for the `add` function from the `calculator` module. We can use the `define` method on the JavaScript linker and provide a JavaScript implementation for the required import:

```
const { Linker } = require("@deislabs/wasm-linker-js");
const { parseText } = require("binaryen");

const usingAdd = `
(module
    (import "calculator" "add" (func $calc_add (param i32 i32) (result i32)))

    (memory 1 1)
    (export "memory" (memory 0))
    (export "add" (func $add))

    (func $add (param i32) (param i32) (result i32)
        (return
            (call $calc_add
                (local.get 0)
                (local.get 1)
            )
        )
    )
)
`;

(async () => {
  var linker = new Linker();

  // The "usingAdd" module imports calculator.add.
  // We define it,  provide a JS implementation, then
  // instantiate it.

  linker.define("calculator", "add", (a, b) => a + b);
  var calc = await linker.instantiate(parseText(usingAdd).emitBinary());
  var result = calc.instance.exports.add(1, 2);
  console.log(result);
})();
```

Once the imports have been defined, the linker also exposes an `instantiate` method which takes care of instantiating the module and passing the correct imports.

> For brevity, next examples will omit importing `@deislabs/wasm-linker-js` and `binaryen`, and will not redeclare previously declared Wasm modules in their text format.

Linking modules, instances, and aliases

But what if we don't want to satisfy imports with JavaScript objects, but with other WebAssembly modules? We would have to instantiate those modules, and manually iterate through their exported items, then construct an import object suitable for the module that is trying to import them. This is the main scenario this package is intended to simplify.

Continuing our example, we want to instantiate our `usingAdd` module (defined above in text format), but satisfy its `calculator add` import through an already compiled WebAssembly module.

For this, we define another module (represented in its text format and contained in the `add` variable) that exports a function called `add` which satisfies the import for our initial `usingAdd` module. We use the linker's `module` method to automatically link the exported items from the `add` module and use them as imports for instantiating `usingAdd`:

```
const add = `
(module
  (memory 1 1)
  (export "memory" (memory 0))
  (export "add" (func $add))

  (func $add (param i32) (param i32) (result i32)
      (return
          (i32.add
              (local.get 0)
              (local.get 1)
          )
      )
  )
)
`;

(async () => {
  var linker = new Linker();

  // The "usingAdd" module above imports calculator.add.
  // We link a module that exports the functionality
  // required, then instantiate the module that uses it.

  await linker.module(
    "calculator",
    new WebAssembly.Module(parseText(add).emitBinary())
  );
  var calc = await linker.instantiate(parseText(usingAdd).emitBinary());
  var result = calc.instance.exports.add(1, 2);
  console.log(result);
})();
```

The `module` method takes a compiled WebAssembly module, instantiates it using the items already defined in the linker, then adds its exported items to the linker so they can be used when instantiating other modules using the `instantiate` method. The linker also allows adding an already instantiated module, through the `instance` method, as well as aliasing a module under a new name, through the `alias` method (Node.js examples for all the public linker methods can be found in the project repository).

The linker methods we have seen so far (`define`, `module`, `instance`, `alias`, and `instantiate`) all have correspondents in the Wasmtime linker, and the APIs are purposefully similar.

Defining asynchronous imports

One of the most popular scenarios in JavaScript applications is waiting for network or I/O operations. Unfortunately, the current WebAssembly MVP does not have a way of waiting for the execution of asynchronous imports (see this issue). To enable this functionality, Binaryen has a pass that transforms a Wasm module and allows it to pause and resume by unwiding and rewinding the call stack.

When enabled, this library can use the JavaScript wrapper of Asyncify and define asynchronous import functions for WebAssembly modules (note that the Asyncify pass must have been applied to the module before instantiating using the linker). This allows us to use satisfy the import of a WebAssembly module by providing a JavaScript function that performs network or file access operations.

Let's satisfy the `calculator add` import of our initial `usingAdd` module by providing a JavaScript function that sleeps for 1.5 seconds before returning the result. First, we have to explicitly enable the Asyncify functionality in our linker (by passing a true boolean to its constructor).

We can now call `define` on the linker, and pass and `async` function.

The main aspect to remember when using Asyncify is that modules instantiated using this method must be transformed by running the Binaryen `asyncify` pass - which we do next using the `runPasses(["asyncify"])`. Then we can continue to use both the linker and instances created with the linker as before:

```
var useAsyncify = true;
var linker = new Linker(useAsyncify);

// Notice how we define an asynchronous import, which
// will wait for 1.5s before returning the result.
var sleep = function (ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
};
linker.define("calculator", "add", async (a, b) => {
  await sleep(1500);
  return a + b;
});

let bytes = parseText(usingAdd);

// we perform the asyncify compiler pass from Binaryen
bytes.runPasses(["asyncify"]);
var calc = await linker.instantiate(bytes.emitBinary());

var result = await calc.instance.exports.add(1, 2);
console.log(result);
```

> Be sure to read Alon Zakai's blog post to better understand how Asyncify works and the overhead it introduces.
>
> Ingvar Stepanyan has a great blog post about building a WebAssembly shell with a real filesystem access in a browser using WASI and Asyncify.

Using the linker in the browser

Currently, this library has limited support for running in a browser. The easiest way to use it is to pull the library through UNPKG, and you can check a complete example of this library in the browser in the repository:

```
<script
  type="module"
  src="https://unpkg.com/@deislabs/wasm-linker-js/dist/wasm-linker.js"
></script>
```

However, keep in mind the library does not currently have support for `instantiateStreaming`, which is the preferred way of instantiating Wasm modules in browsers. The module system for generating a browser compatible library is also far from ideal - check this issue if you want to help make it better!

We want to thank the WebAssembly community for building great tools, and we hope this little library will be helpful for people working with WebAssembly from JavaScript runtimes!