shopify

Javy

Creating a JS to Wasm toolchain

Jeff Charles

Senior Developer,
Wasm Foundations

Waterloo, 🇨🇦

Agenda

01	What is Javy?
02	How to write code for Javy
03	Why we made Javy
04	Dive into different ways JS source code is transformed to Wasm
05	How we add functions to Javy

What is Javy?

- A way to convert JS to a Wasm module
- A CLI application and a Wasm library
- CLI takes JS source code in and outputs a Wasm module
- Wasm library can compile JS to QuickJS bytecode and evaluate QuickJS bytecode
- Internals are used by many companies present here
- Targets WASI

Coding for Javy

Wraps top-level evaluation in WASI command pattern

Uses stdin for input, stdout for output, and stderr for logs

Code example

Given an input like `` '{ "foo": 1 }' ``, increment the foo parameter

```
import { STDIO, readFileSync, writeFileSync } from 'javy/fs';

const inputBytes = readFileSync(STDIO.Stdin);
const inputString = new TextDecoder().decode(inputBytes);
const input = JSON.parse(inputString);

input.foo += 1;

const outputString = JSON.stringify(input);
const outputBytes = new TextEncoder().encode(outputString);
writeFileSync(STDIO.Stdout, outputBytes);
```

```
import { STDIO, readFileSync, writeFileSync } from 'javy/fs';

const inputBytes = readFileSync(STDIO.Stdin);
const inputString = new TextDecoder().decode(inputBytes);
const input = JSON.parse(inputString);

input.foo += 1;

const outputString = JSON.stringify(input);
const outputBytes = new TextEncoder().encode(outputString);
writeFileSync(STDIO.Stdout, outputBytes);
```

```
import { STDIO, readFileSync, writeFileSync } from 'javy/fs';

const inputBytes = readFileSync(STDIO.Stdin);
const inputString = new TextDecoder().decode(inputBytes);
const input = JSON.parse(inputString);

input.foo += 1;

const outputString = JSON.stringify(input);
const outputBytes = new TextEncoder().encode(outputString);
writeFileSync(STDIO.Stdout, outputBytes);
```

```
import { STDIO, readFileSync, writeFileSync } from 'javy/fs';

const inputBytes = readFileSync(STDIO.Stdin);
const inputString = new TextDecoder().decode(inputBytes);
const input = JSON.parse(inputString);

input.foo += 1;

const outputString = JSON.stringify(input);
const outputBytes = new TextEncoder().encode(outputString);
writeFileSync(STDIO.Stdout, outputBytes);
```

```
$ esbuild index.mjs --bundle --outfile=dist.js
$ javy compile dist.js -o example.wasm
$ echo '{ "foo": 1 }' | wasmtime example.wasm
{"foo":2}
```

Why we built Javy

Shopify Functions allows customizing parts of backend logic with Wasm modules

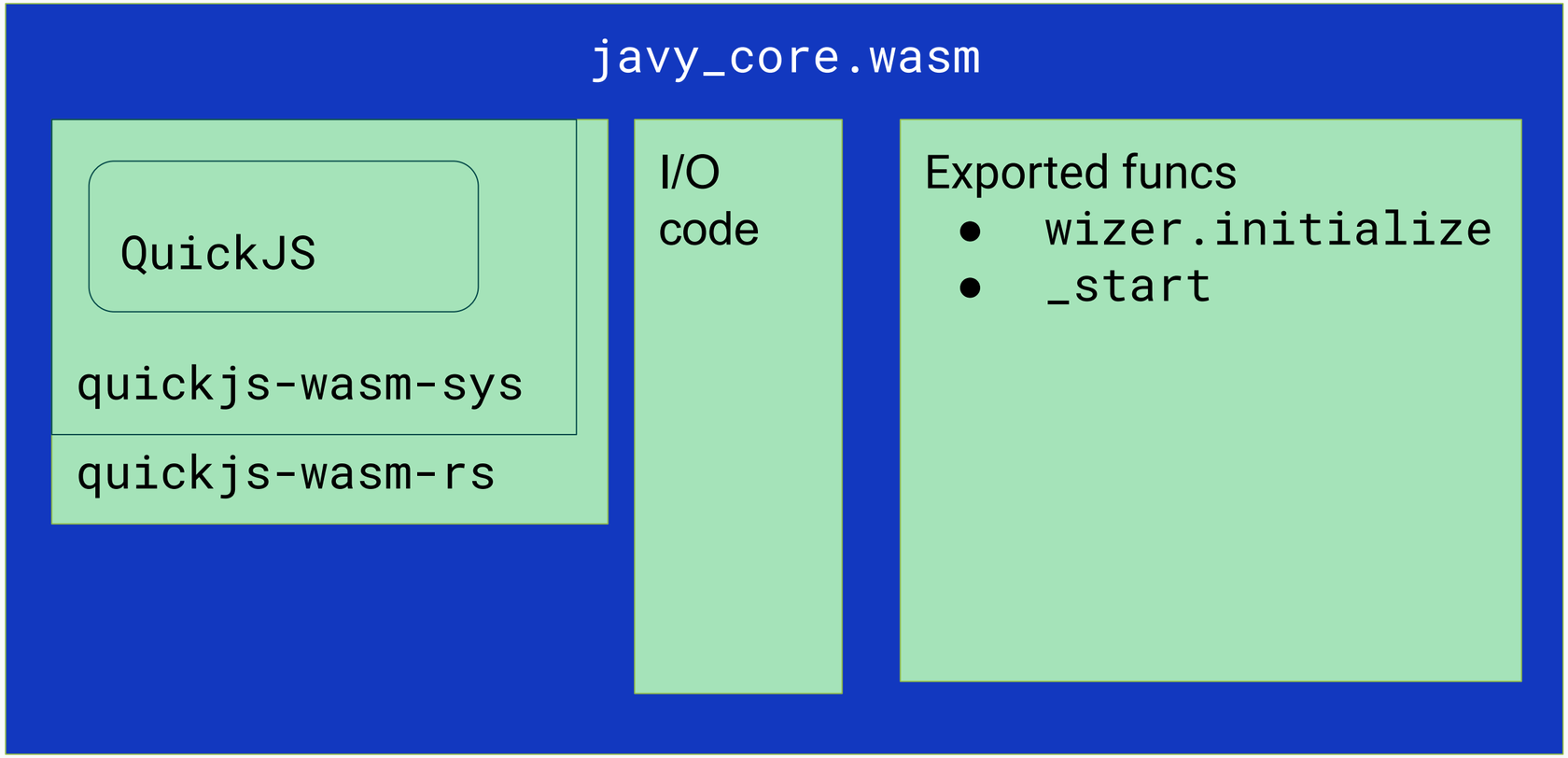
Tight constraints:

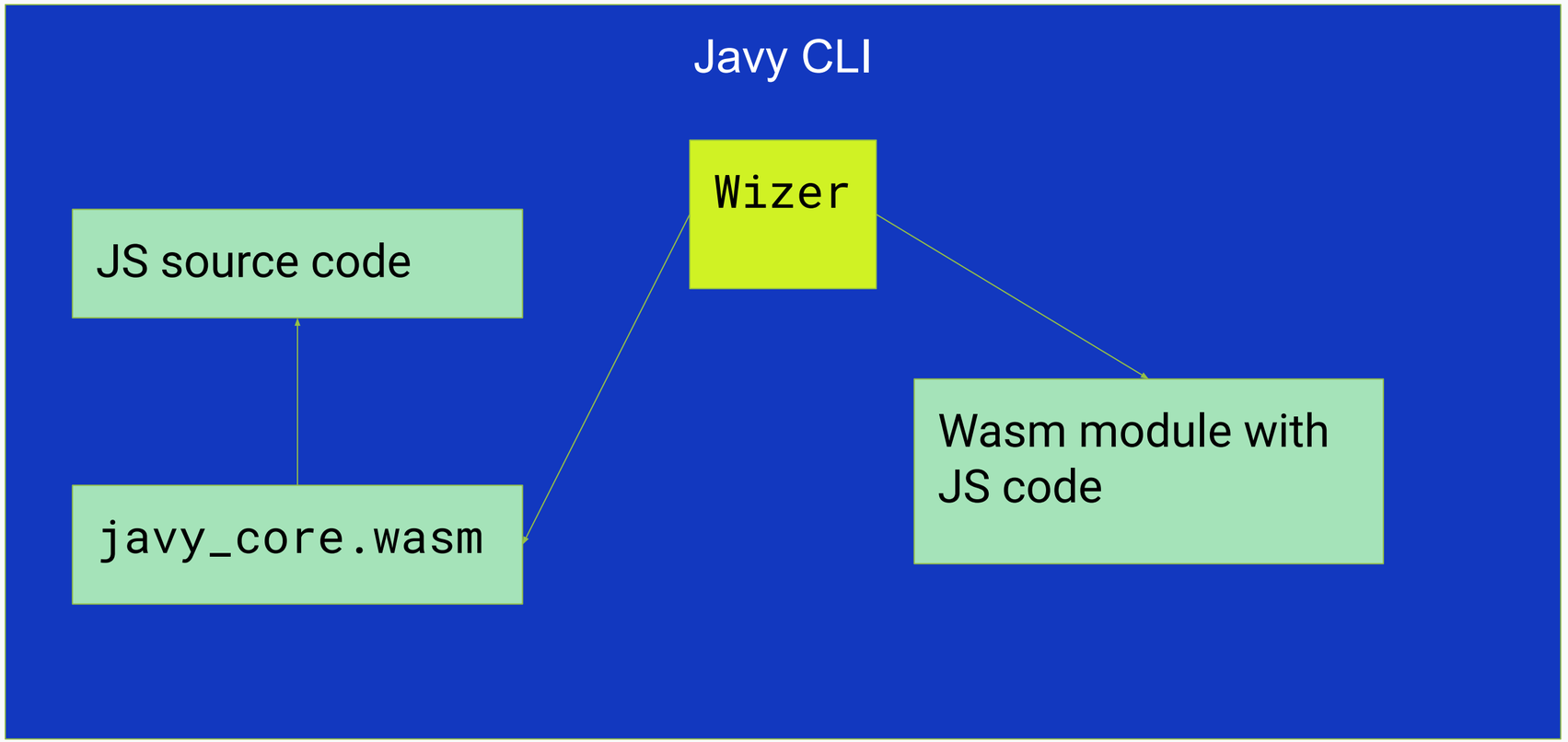
- Need to finish in under 5 ms
- Need to be smaller than 256 kb

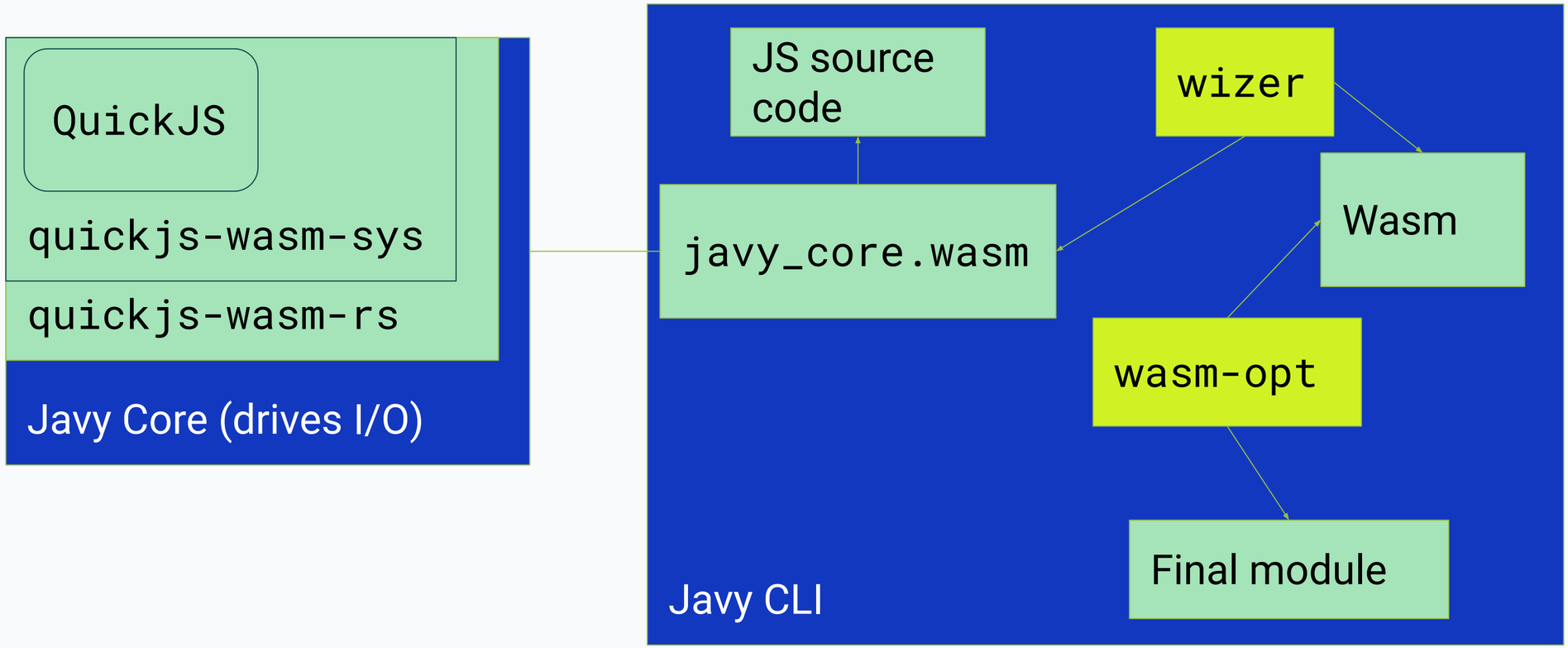
Developers want to write their Function code in JS/TS

No non-Wasm runtime environment to run JS

How we create Wasm that runs JS


javy_core.wasm
QuickJS
quickjs-wasm-sys
quickjs-wasm-rs
I/O
code
Exported funcs
wizer.initialize
_start


Javy CLI
Wizer
JS source code
javy_core.wasm
Wasm module with
JS code


QuickJS
quickjs-wasm-sys
quickjs-wasm-rs
Javy Core (drives I/O)
JS source code
wizer
javy_core.wasm
Wasm
wasm-opt
Final module
Javy CLI

Quick tour of quickjs-wasm-rs

- `Context`
- `Value`
  - Includes `serde` support

Adding JS APIs

For example, `TextDecoder`

```
(function () {
    const __javy_decodeUtf8BufferToString =
        globalThis.__javy_decodeUtf8BufferToString;
    class TextDecoder {
        ...
        decode(input, options = {}) {
            ...
            return __javy_decodeUtf8BufferToString(input,
                byteOffset, byteLength, this.fatal, this.ignoreBOM);
        }
        ...
    }
    globalThis.TextDecoder = TextDecoder;
    Reflect.deleteProperty(
        globalThis, "__javy_decodeUtf8BufferToString");
})();
```

```rust
fn ...(context: &Context) -> Result<()> {
    ...
    context.eval_global(
        "text-encoding.js",
        include_str!("../prelude/text-encoding.js"),
    )?;
    ...
}
```

```rust
fn ...(..., context: &Context, ...) {
    let global = context.global_object()?;
    global.set_property(
        "__javy_decodeUtf8BufferToString",
        context.wrap_callback(decode_utf8_buffer_to_js_string())?,
    )?;
}

fn decode_utf8_buffer_to_js_string(
) -> impl FnMut(&Context, &Value, &[Value]) -> anyhow::Result<Value> {
    move |ctx: &Context, _this: &Value, args: &[Value]| {
        …
    }
}
```

```rust
fn ...(..., context: &Context, ...) {
    let global = context.global_object()?;
    global.set_property(
        "__javy_decodeUtf8BufferToString",
        context.wrap_callback(decode_utf8_buffer_to_js_string())?,
    )?;
}

fn decode_utf8_buffer_to_js_string(
) -> impl FnMut(&Context, &Value, &[Value]) -> anyhow::Result<Value> {
    move |ctx: &Context, _this: &Value, args: &[Value]| {
        …
    }
}
```

This all works great! Except **the smallest modules are 870 kb** and we have a size limit of 256 kb. 😔

Enter dynamic linking

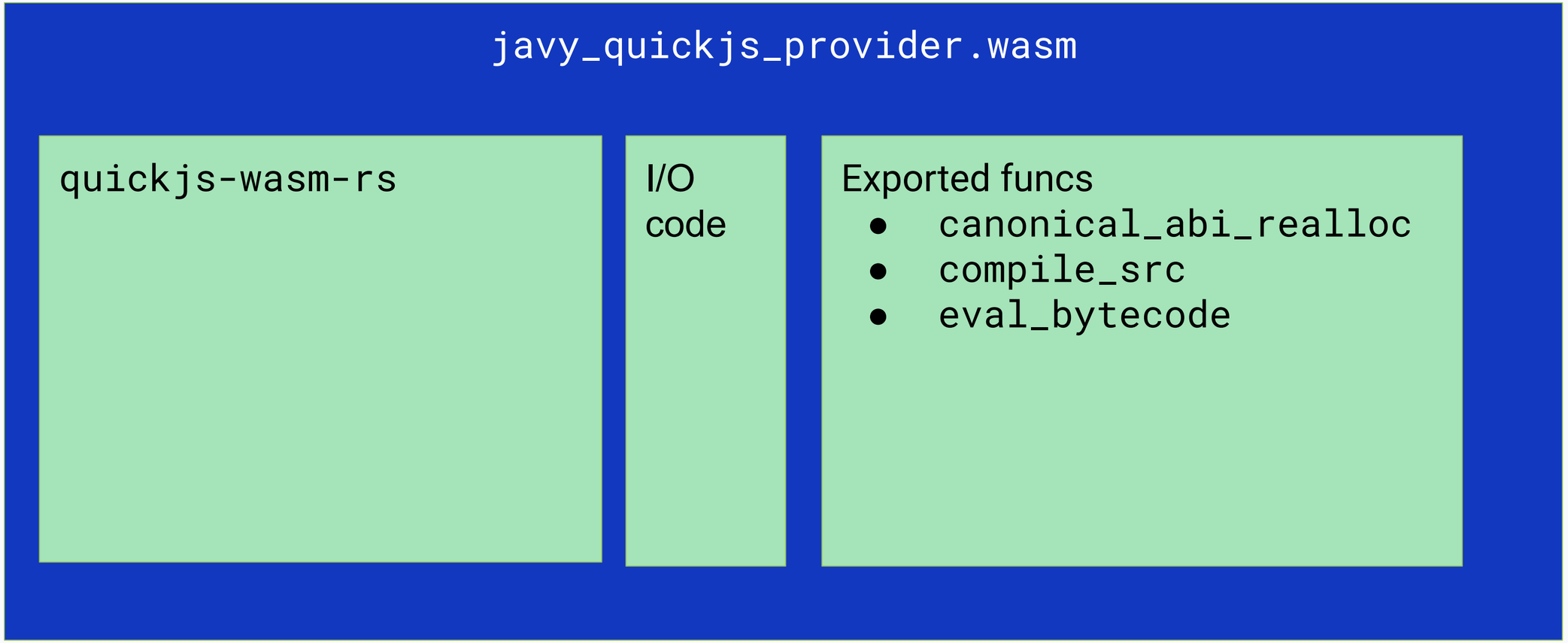
Dynamic linking

Separate Wasm modules for JS bytecode and interpreter (we call it the provider)

Uses linker in Wasmtime

JS code module imports memory and functions from provider module

Uses memory allocation function defined in the component model


javy_quickjs_provider.wasm
quickjs-wasm-rs
I/O
code
Exported funcs
canonical_abi_realloc
compile_src
eval_bytecode

CLI

`javy_quickjs_provider` is embedded as a library

1. Call `compile_src`
    - Uses Wasmtime to instantiate provider and invoke function
2. Use `wasm-encoder` to write module

```
(module
    (func $realloc
        (import "javy_quickjs_provider_v1" "canonical_abi_realloc")
        (param i32 i32 i32 i32)
        (result i32))
    (func $eval_bytecode
        (import "javy_quickjs_provider_v1" "eval_bytecode")
        (param i32 i32))

    (memory
        (import "javy_quickjs_provider_v1" "memory") 0)

    (func (export "_start") ...)
    (data $bytecode "<bytecode>"))
```

```
(module
    (func $realloc
        (import "javy_quickjs_provider_v1" "canonical_abi_realloc")
        (param i32 i32 i32 i32)
        (result i32))
    (func $eval_bytecode
        (import "javy_quickjs_provider_v1" "eval_bytecode")
        (param i32 i32))

    (memory
        (import "javy_quickjs_provider_v1" "memory") 0)

    (func (export "_start") ...)
    (data $bytecode "<bytecode>"))
```

```
(module
    (func $realloc
        (import "javy_quickjs_provider_v1" "canonical_abi_realloc")
        (param i32 i32 i32 i32)
        (result i32))
    (func $eval_bytecode
        (import "javy_quickjs_provider_v1" "eval_bytecode")
        (param i32 i32))

    (memory
        (import "javy_quickjs_provider_v1" "memory") 0)

    (func (export "_start") ...)
    (data $bytecode "<bytecode>"))
```

```
(func (export "_start") (local $buffer_ptr i32).
    (local.set $buffer_ptr
        (call $realloc
            (i32.const 0)                       ;; Original ptr
            (i32.const 0)                       ;; Original length
            (i32.const 1)                       ;; Alignment
            (i32.const <bytecode length>))) ;; Length
    (memory.init $bytecode
        (local.get $buffer_ptr)         ;; Memory destination
        (i32.const 0)                   ;; Start index
        (i32.const <bytecode length>) ;; Length
    (data.drop $bytecode)
    (call $eval_bytecode
        (local.get $buffer_ptr)
        (i32.const <bytecode length>)))
(data $bytecode "<bytecode>")
```

```
(func (export "_start") (local $buffer_ptr i32).
    (local.set $buffer_ptr
        (call $realloc
            (i32.const 0)                       ;; Original ptr
            (i32.const 0)                       ;; Original length
            (i32.const 1)                       ;; Alignment
            (i32.const <bytecode length>))) ;; Length
    (memory.init $bytecode
        (local.get $buffer_ptr)         ;; Memory destination
        (i32.const 0)                   ;; Start index
        (i32.const <bytecode length>) ;; Length
    (data.drop $bytecode)
    (call $eval_bytecode
        (local.get $buffer_ptr)
        (i32.const <bytecode length>)))
(data $bytecode "<bytecode>")
```

```
(func (export "_start") (local $buffer_ptr i32).
    (local.set $buffer_ptr
        (call $realloc
            (i32.const 0)                       ;; Original ptr
            (i32.const 0)                       ;; Original length
            (i32.const 1)                       ;; Alignment
            (i32.const <bytecode length>))) ;; Length
    (memory.init $bytecode
        (local.get $buffer_ptr)         ;; Memory destination
        (i32.const 0)                   ;; Start index
        (i32.const <bytecode length>) ;; Length
    (data.drop $bytecode)
    (call $eval_bytecode
        (local.get $buffer_ptr)
        (i32.const <bytecode length>)))
(data $bytecode "<bytecode>")
```

```
$ echo 'console.log("hello world!");' > my_code.js
$ javy compile my_code.js -o example.wasm -d
$ javy emit-provider -o provider.wasm
$ wasmtime run --preload javy_quickjs_provider_v1=provider.wasm example.wasm
hello world!
```

```
$ echo 'console.log("hello world!");' > my_code.js
$ javy compile my_code.js -o example.wasm -d
$ javy emit-provider -o provider.wasm
$ wasmtime run --preload javy_quickjs_provider_v1=provider.wasm example.wasm
hello world!
```

What we covered

01 — What Javy is and how to write code for it

02 — Create modules with QuickJS embedded

03 — quickjs-wasm-rs and adding JS functions

04 — Dynamically linked modules

Q&A

Shopify engineering blog:

https://shopify.engineering