BYTECODE ALLIANCE

JCO and Browsers

Victor Adossi

WebAssembly Plumber's Summit 2026

Slides

Slides

Quick overview: The JS Wasm ecosystem

There is a wide span of projects in the JS ecosystem:

- StarlingMonkey
  - 🟦 SpiderMonkey
- ComponentizeJS
  - 🦀 spidermonkey-embedding-splicer
- Jco
  - ☕ @bytecodealliance/jco
  - ☕ @bytecodealliance/jco-std
  - ☕ @bytecodealliance/jco-transpile
  - 🦀 js-component-bindgen
  - 🦀 wasm-tools-component (internal)
  - ☕ preview2-shim
  - ☕ preview3-shim

🟦 C++
🦀 Rust
☕ Javascript

Slides

Quick overview: JS -> Component (now)

Here's a very quick overview of how JS code becomes a component

1. Build SpiderMonkey into Wasm w/ P2 bindings via StarlingMonkey
2. Take your input JS and build it into StarlingMonkey Wasm
3. Use componetize-js to splice expected interface(s) in/out of the resulting Wasm binary
   a. componentize-js depends on a `jco transpile`'d build of `js-component-bindgen`
4. Run Wizer to initialize and snapshot
5. Pass the component to a runtime (`wasmtime`, V8 via `jco transpile`)

How JS code will become a component (soon^{tm})

Slides

Just some of the work currently under way 🎉

P3 host bindings

Rust bindings/harness for StarlingMonkey (Thanks Till!)

Easier build, development flow, easier to write “safe” code

ComponentizeJS being rewritten to target wit-dylib (Thanks Joel!)

No more component splicer <-> js-component-bindgen dependency hell

Alternative runtimes like componentize-qjs (Thanks Tomasz!)

More points in the continuum between size and sophistication of JS components

P3 guest bindings (Thanks Till!)

Builds on StarlingMonkey’s Rust bindings/harness

Slides

How a component runs in jco transpile (now)

Here’s how a Jco component runs in WebAssembly supporting JS engines:

jco transpile essentially wasm-tools unbundle s your component, then...

- 🔌 P2 imports are wired to JS-side trampolines (i.e. wasmtime_environ::Trampoline)
- 👟 Wasm instructions walked, used in generation (i.e. wit_bindgen_core::Instruction)
- 🏭 Code that supports P1/P2 builtins is generated (e.g. resources, tables, lifts/lowers)
- 📦 Code that supports imports is imported/connected (e.g. [method]descriptor.stat)
- ⏱️ JSPI is used (i.e. WebAssembly.suspending, thanks Calvin!)

P3 components run in `jco transpile` (now)

Slides

jco transpile essentially wasm-tools unbundle s your component

- 🔌 P3 imports are wired to JS-side trampolines (i.e. wasmtime_environ::Trampoline)
  - **E.g. [async-lower]*, [stream-new-<ordinal>]*, etc**
- 👟 Wasm instructions walked, used in generation (i.e. wit_bindgen_core::Instruction)
  - **The "return" of a function distinct, new implicit async callback driver**
- 🏭 **Code that supports P3 builtins is generated (e.g. tables, metadata, flat lift/lower)**
- 📦 Code that supports imports is imported/connected (e.g. [method]descriptor.stat)
  - **Ergonomics of specifying async imports/exports has changed**
- ⏲️ JSPI is used (e.g. WebAssembly.suspending)
  - **Deep threading is now sometimes necessary (e.g. async export that calls async import, tunneled through a generated patch-up component's func table)**

What’s done? What’s left?

Slides

1. **Done***
    a. Async Tasks, Subtasks
    b. Waitables
    c. Backpressure
    d. Context
    e. Streams**
2. **Left**
    a. Futures
    b. P3-shim integration
3. **Future**
    a. Threads (currently, we pretend to have *one* thread)
4. **Nice to have / Far future**
    a. Implementation polish (eye towards perf, more dead code elimination, ergonomics, etc)
    b. Debugging improvements (a new JCO_DEBUG flag was added, but it can be improved)
    c. Improving Documentation
    d. Fuzzing/Property testing

* Implies likely spec-compliance and unlikely to be obviously buggy as confirmed by test-suite, not perfect code, of course.
** Long tail of ensuring lift/lowers and interactions across types and containers are properly working. No fuzzing/property testing yet

What does the code look like?

Slides

With WIT like this:

```
package jco:test-components;

interface get-stream-async {
    get-stream-u32: async func(vals: list<u32>) -> result<stream<u32>, string>;
}

world stream-tx {
  export get-stream-async;
}
```

What does the code look like?

Slides

And Rust guest code like this:

```rust
mod bindings {
    use super::Component;
    wit_bindgen::generate!({ world: "stream-tx", });
    export!(Component);
}

use bindings::wit_stream;
use wit_bindgen::StreamReader;
use bindings::exports::jco::test_components::get_stream_async::Guest;

struct Component;

impl Guest for Component {
    async fn get_stream_u32(vals: Vec<u32>) -> Result<StreamReader<u32>, String> {
        let (mut tx, rx) = wit_stream::new();
        wit_bindgen::spawn(async move {
            tx.write_all(vals).await;
        });
        Ok(rx)
    }
}
```

What does the code look like?

Slides

Use jco transpile'd output like this:

```
import esModule from "./transpiled-output";

const { WASIShim } = await import("@bytecodealliance/preview2-shim/instantiation");
const instance = await esModule.instantiate(
    undefined,
    new WASIShim().getImportObject(),
);

const vals = [11, 22, 33];
const stream = await instance["jco:test-components/get-stream-async"].getStreamU32(vals);

const first = await stream.next(); // 11
const second = await stream.next(); // 22
const third = await stream.next(); // 33
```

Slides

When can you play with this? Right now!

Progress has been going into successive Jco versions

(there have been few regressions given the separation/lack of interaction of P3 features and existing test suite coverage)

The test suite grows every week and consists of upstream and local components under test.

1.17.0 is the latest release (12 days ago)

1.18.0 is likely to be out this week (latest Streams impl and tests)

Unfortunately, you will likely still encounter bugs

Thanks in advance for filing issues

P3 Shims

Slides

Thanks to Tomasz for his excellent work on P3 shims

Independent test suite confirms functionality against Web APIs & NodeJS

Implementations have been updated to the latest 0.3.0 RC interfaces

Browser support is still “experimental” in Jco, deferred for post 0.3.0 launch high priority after NodeJS stabilize

Browser support is still “Experimental”? 😠

Slides

We’re missing a few key P2 implementations for robust Browser support

Community members have stepped up to help fill gaps:

feat(preview2-shim): implement browser HTTP fetch support via JSPI

(thanks @danbopes!)

We’re missing much more P3 implementations

(much implementation is shared, so this isn’t as bad as it sounds)

🙏 We’d love help with our P2 and P3 browser shims

Slides

🌇 Fin.

Questions?

🤝 **Consider contributing to the JS Wasm ecosystem!**
The codebases are now *much* easier to approach than they were in the past.

🙇‍♂️ **Thanks to all the contributors** to StarlingMonkey, ComponentizeJS, Jco and upstream ecosystem crates (wasmtime, wasm-tools, orca/wirm, etc) who make the JS ecosystem possible.