cosmonic

Assembling a Bigger Band:

Backstage Plugins in Any Language with WebAssembly

Victor Adossi
Cosmonic
vados-cosmonic

cosmonic

Backstage

Write code.

Click one or more buttons in Backstage.

Computers get hot.

Your code gets deployed.

$CLOUD_PROVIDER bill goes up.

Customers happy (?).

This image was created by ChatGPT. Any likeness to Fred Flinstone, a licensed character of Hanna-Barbera is purely coincidental and only indicative of copyright violations by OpenAI's DALLE Generative AI model.

cosmonic

Why are we here ~~on earth~~ today?

Backstage makes Dev & Ops easier

(This is good)

More plugins = Better Backstage

(Unless you’re doing nothing but writing bugs)

Backstage is extended by writing Javascript

(This is REDACTED)

Sometimes, developers who don’t write Javascript have ideas too.

(Unconfirmed, but likely true)

cosmonic

How can we enable more languages, and get a bigger plugin ecosystem?

1. V8 Isolates?
2. Virtual Machines?
3. Microservices?
4. **WebAssembly**

cosmonic

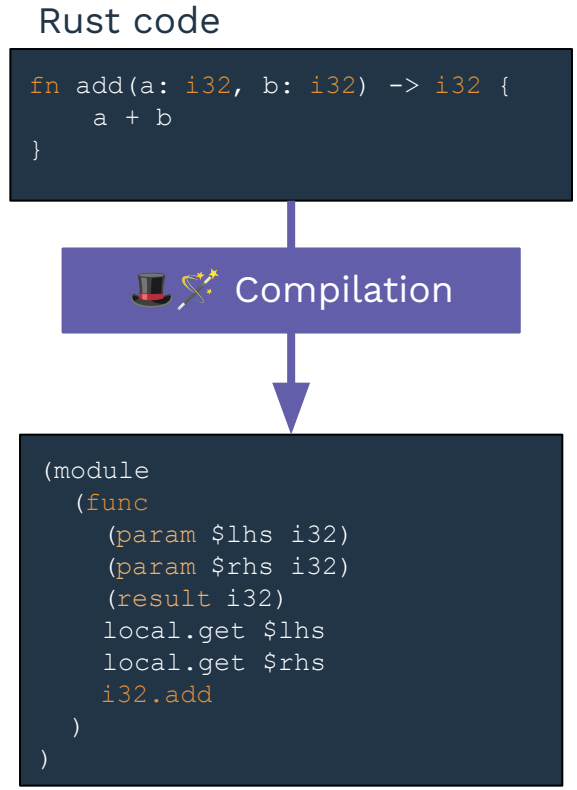
What is WebAssembly?

TL;DR WebAssembly is a compile target.

C code compiles to machine-specific binary

Java code compiles to machine-independent (but JVM/JRE specific) bytecode

Python code compiles to machine-independent (but JVM specific) bytecode



WebAssembly is like a tiny VM

cosmonic

Why does WebAssembly fit?

WebAssembly is well-sandboxed: programs cannot do things like access the filesystem by default.

WebAssembly is performant: the most critical perf research we can find cites an average slowdown of 50% to native code (for reference, Python is ~25x slower than native code).

WebAssembly is cross-platform: building for WebAssembly means running on many platforms easily (for real this time).

Read more @ https://webassembly.org/docs/security

cosmonic

Well, we’re here because we built it. (2x)

We are a WebAssembly company after all.

For Backstage frontend plugins:

- An interface (think gRPC) that represents a frontend plugin
- A CLI that builds & installs a plugin from WASM binary

For Backstage backend plugins:

- An interface that represents a backend plugin
- A CLI that builds, installs & serves requests from WASM binary

cosmonic

TL;DR: WebAssembly Interface Types (“WIT”)

Base WebAssembly can “only” do in computation numbers – `i32`, `f32`, `i64`, `f64`.*

Extensions to WebAssembly build on the “just numbers” base.

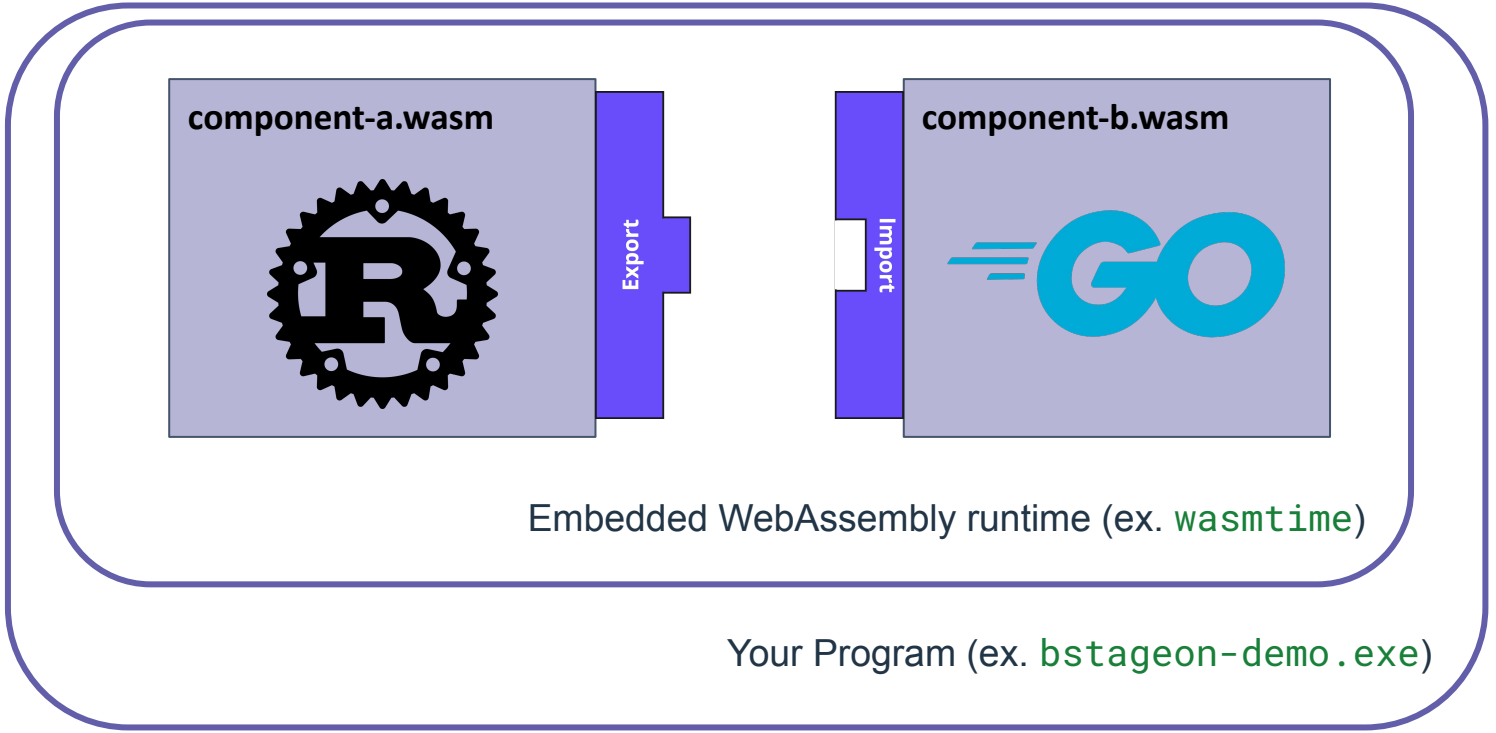
```
package backstagecon:example;

interface logging {
  log: func(msg: string);
}

world logger {
  export logging;
}
```

cosmonic

WebAssembly, WIT, and you(r programs)



cosmonic

How does it work? (Frontend edition)

1. Write frontend code.
2. Build a WASM binary from any language that supports WebAssembly.
3. If the WASM binary matches this WIT interface, we'll turn it into a Backstage plugin with no extra work.*

```
interface backstage-frontend-plugin {
  /// A representation of a node dependency
  ///  (one that might be installed via `npm install`/`yarn add`)
  record node-dependency {
    /// Name of the dependency (ex. 'react')
    name: string,
    /// Version of the dependency
    version: option,
    /// Whether this is a development-only dependency
    dev: bool,
 }

  // Get node dependencies that need to be installed
  get-node-deps: func() -> result<list<node-dependency>, string>;

  /// Files that make up React components
  /// to be used in in the frontend plugin
  record component-file {
    /// (snip)
  }

  /// Get the files that make up the frontend of the component
  get-component-files: func() -> result<list<component-file>, string>;
}
```

cosmonic

Example (Rust) - Build setup

```toml
[package]
name = "backstage-rust-frontend-plugin"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib"]

[dependencies]
wit-bindgen = { version = "0.22.0", default-features = false, features = ["realloc"] }
```

cargo.toml

```toml
[build]
target = "wasm32-wasi"
```

.cargo/config.toml (optional)

cosmonic

Example (Rust) - Imports

```rust
mod bindings;

use crate::bindings::exports::component::backstage_rust_plugin::backstage_frontend_plugin::{
    ComponentFile, Guest, NodeDependency,
};

/// The implementations of the contract will
/// hang off of the struct below
struct Component;
```

src/lib.rs (top)

cosmonic

Example (Rust) - Specifying NodeJS deps

```rust
impl Guest for Component {
    fn get_node_deps() -> Result<Vec<NodeDependency>, String> {
        return Ok(vec![
            NodeDependency {
                name: "react".into(),
                version: None,
                dev: false,
            },
            // ... (snip)
        ]);
    }

    // ... (snip)
}
```

src/lib.rs

cosmonic

Example (Rust) - Bundling frontend files

```rust
impl Guest for Component {
    // ... (snip)

    fn get_component_files() -> Result<Vec<ComponentFile>, String> {
        // NOTE: these paths are available *under* `src/components`
        // in the generated plugin
        return Ok(vec![
            ComponentFile {
                path: "ExampleComponent/ExampleComponent.tsx".into(),
                is_root: true,
                component_class_name: Some("ExampleComponent".into()),
                contents: String::from(include_str!(
                    "../public/components/ExampleComponent/ExampleComponent.tsx"
                )),
            },
            // ... (snip)
        ]);
    }
}
```

src/lib.rs

(((Frontend 🦀 Demo)))

cosmonic

How does it work? (Backend edition)

1. Implement the WIT contract
2. Build a WASM binary from any language that supports WebAssembly.
3. If the WASM binary matches this WIT interface, we'll turn it into a Backstage plugin with no extra work.*

```
interface backstage-backend-plugin {
  /// A representation of a node dependency
  ///  (one that might be installed via `npm install`/`yarn add`)
  record endpoint {
    /// Name of the dependency (ex. 'react')
    path: string,
    /// Version of the dependency
    method: string,
 }

  // Get endpoints that should be served
  get-endpoints: func() -> result<list<node-dependency>, string>;
}

world plugin {
  export backstage-backend-plugin;
  export wasi:http/incoming-handler;
}
```

cosmonic

What's `wasi:http/incoming-handler`?

"Vanilla" WebAssembly only deals with numbers (`i32`, `f64`, ...)

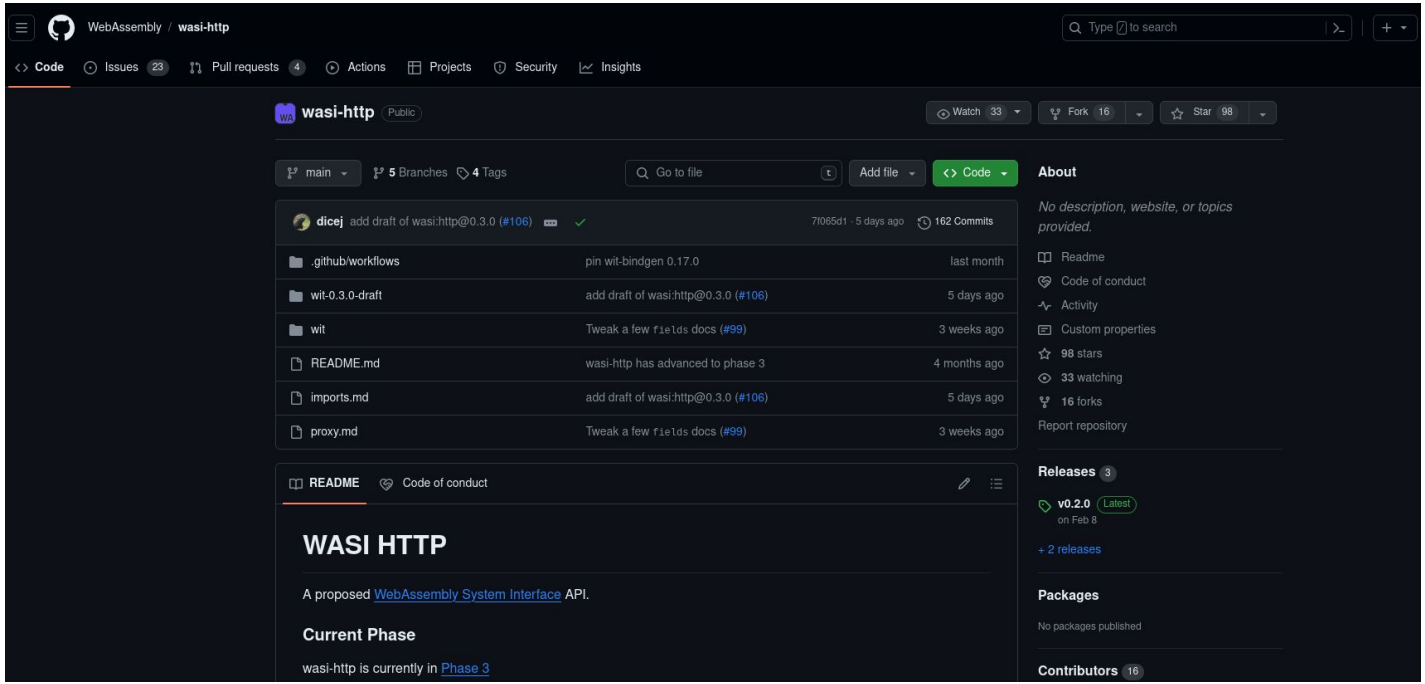
WebAssembly Interface Types ("WIT") eases the use of complex types (see also: the Component Model)

WebAssembly System Interfaces ("WASI") encapsulate things we do with software every day, reusably.

`wasi:http` is a reusable abstraction layer for fundamental HTTP functionality software depends on.

cosmonic

Don’t take my word for it!



https://github.com/WebAssembly/wasi-http

cosmonic

Example (Rust) - Specifying endpoints

```rust
impl Guest for Component {
    fn get_routes() -> Result<Vec<Route>, String> {
        return Ok(vec![Route {
            method: "GET".into(),
            path: "/demo".into(),
        }]);
    }
}
```

src/lib.rs

cosmonic

Example (Rust) - Implementing the HTTP handler

```rust
impl incoming_handler::Guest for Component {
    fn handle(request: IncomingRequest, response_out: ResponseOutparam) -> () {
        // Read the request path, stripping query if present
        let path = request
            .path_with_query()
            .expect("failed to read incoming request path");
        // ... (snip)

        // Dispatch the request
        match (request.method(), path) {
            (Method::Get, "demo") => {
                response_body_w
                    .blocking_write_and_flush(
                        r#"{"status": "success", "message": "hello Kubecon 2024!"}"#.as_bytes(),
                    )
                    .expect("blocking write and flush failed");
            }
            // ... (snip)
        }
        // ... (snip)
    }
}
```

src/lib.rs

(((Backend 🦀 Demo)))

cosmonic

What could be better?

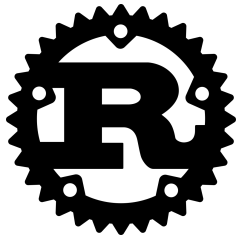
There are a few things that we'd like to improve:

- Better frontend integration (can we convert React components to WebAssembly more generally?)
- Typescript AST-based manipulation of Backstage code, not strings
- Faster developer loop for the `bingband` CLI, HMR?
- Wiring in of Backstage resources like AuthN and the database (i.e. more WIT)

What can *you* build with the thing we built?

Here's your personal invitation to build frontend and backend plugins with this toolkit.

The following language toolchains work well with WebAssembly:

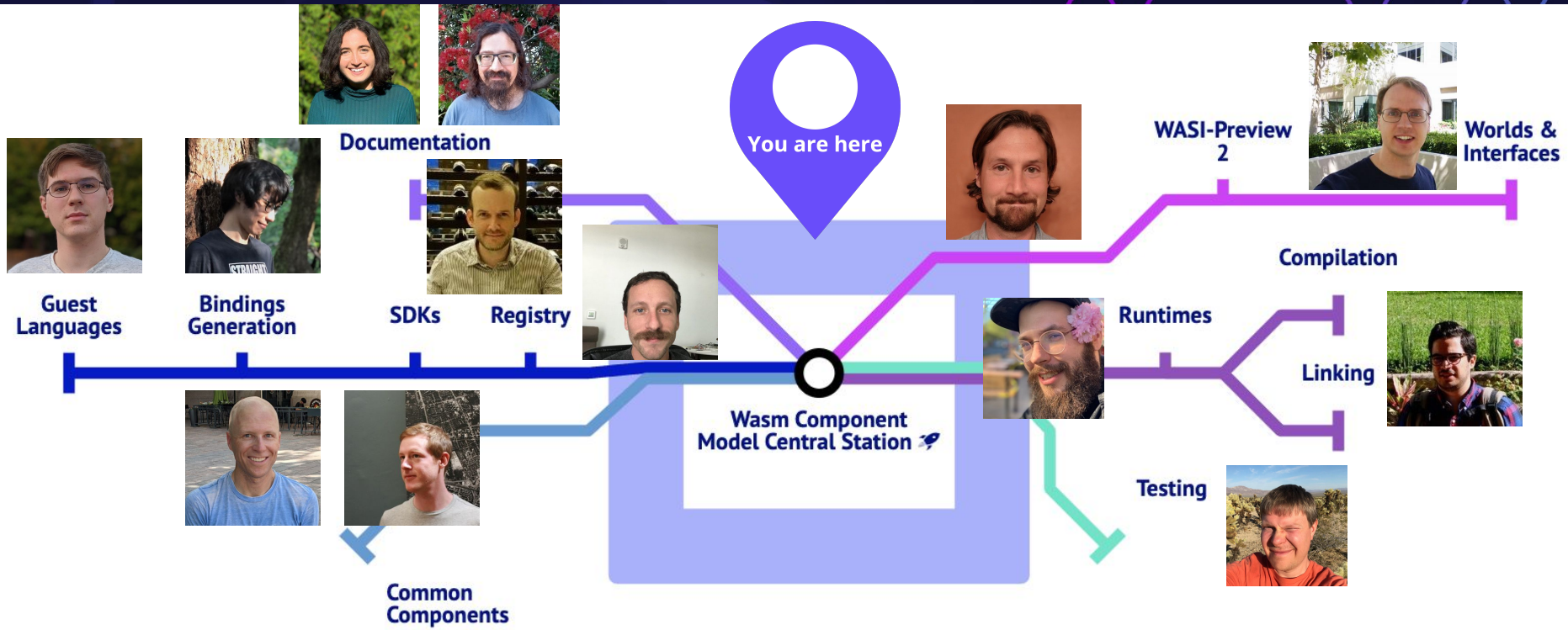


Who is building WebAssembly?

WebAssembly is awesome because of work done by many individuals, under the guidance of the Bytecode Alliance.

Learn more @ bytecodealliance.org
Watch the meetings on YouTube: @bytecodealliance
Say “Hi” 👋 on Zulip: https://bytecodealliance.zulipchat.com

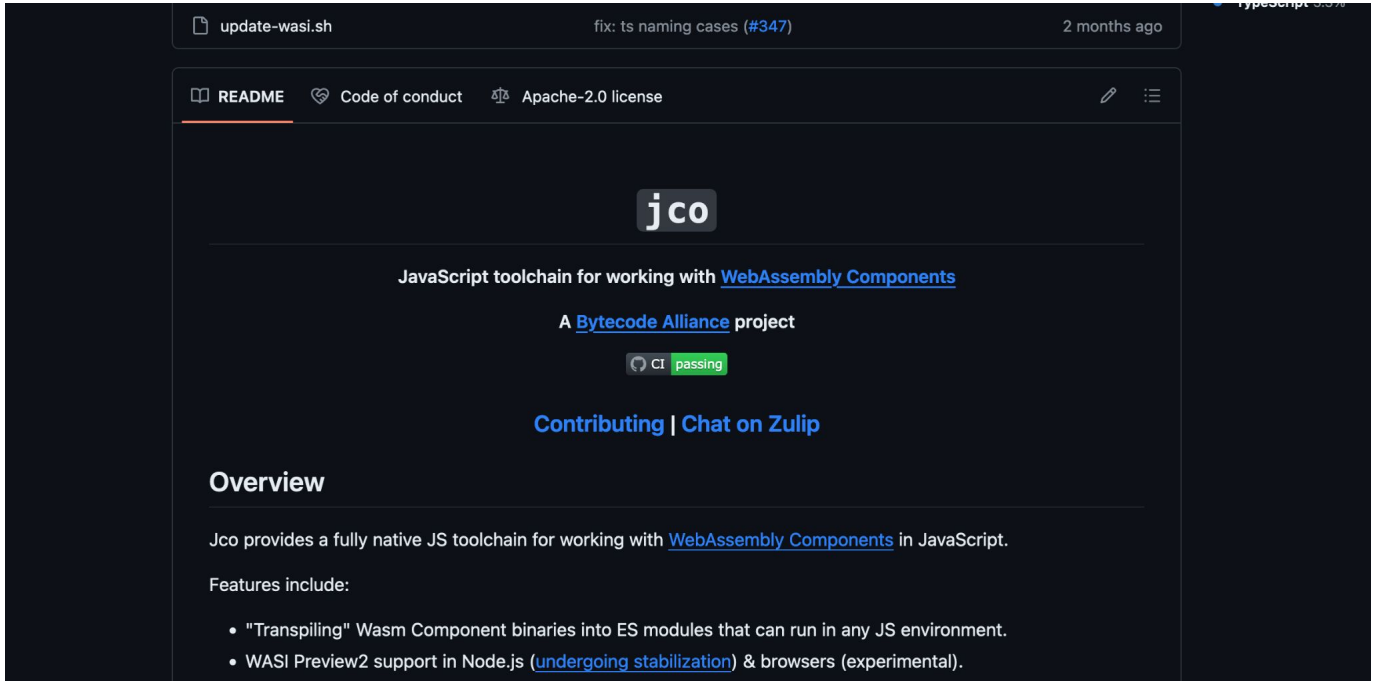
Just a few more people making WebAssembly happen



Thanks to Bailey Hayes (@ricochet) for putting this slide together

cosmonic

Special thanks to `jco` maintainers



github.com/bytecodealliance/jco

cosmonic

Why we love Backstage

We're building an application platform so both Dev and Ops can ship faster, and **Backstage is a natural fit.**

We think it's simple:

We ship our platform with Backstage support.

You ship your infrastructure with Backstage.

Our platform ships your applications.

Your applications delight users.

cosmonic

Thanks for listening.
Here's where you can find the code

For Zoomers

For non-Zoomers and/or the security-conscious:

github.com/cosmonic-labs/backstage-bigband

Questions?

github.com/cosmonic-labs/backstage-bigband