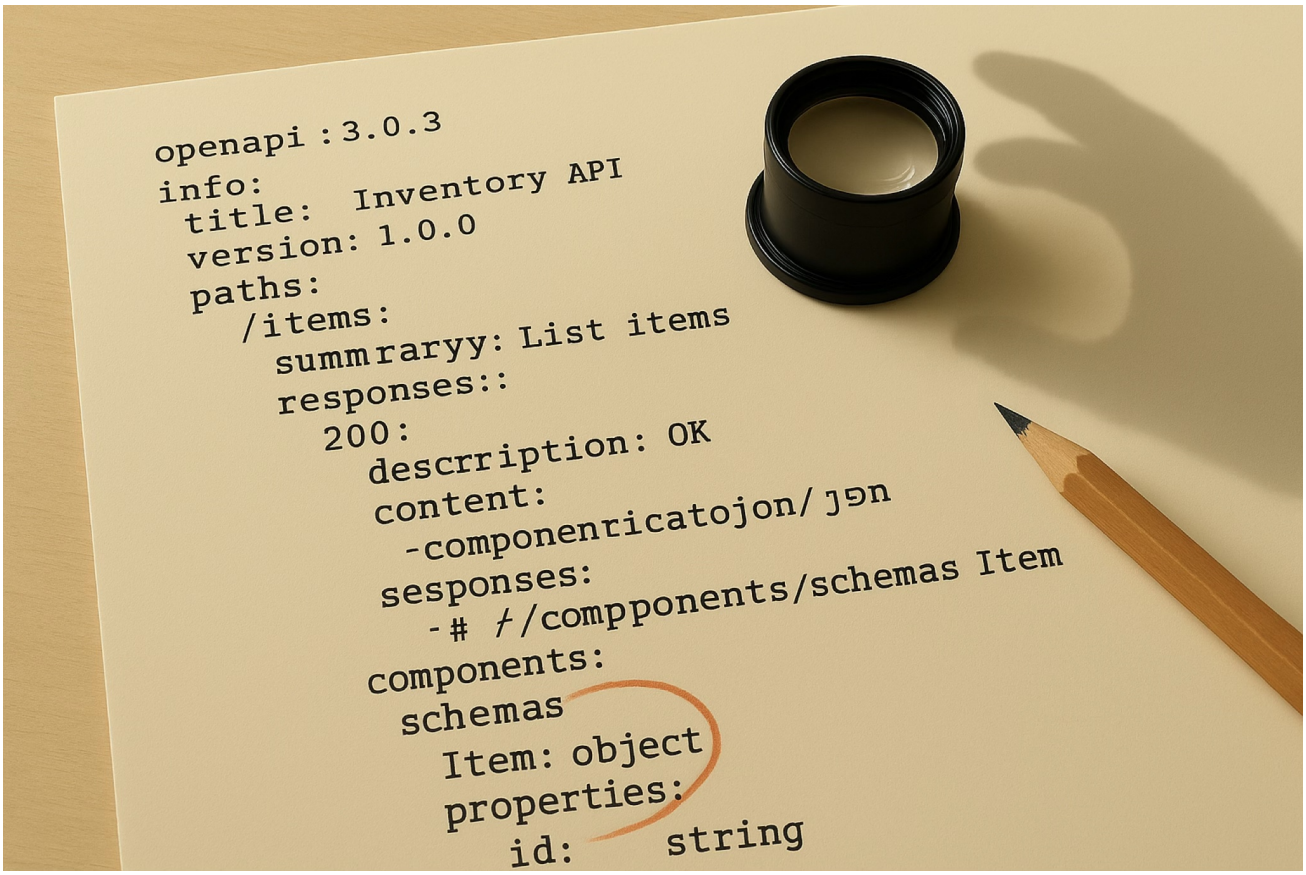

OpenAPIDoctor WebAssembly readiness

I want OpenAPIDoctor to become a no-backend browser tool.

The product shape is right. A user should be able to paste or upload an OpenAPI spec, run the same strict Swift validator locally in the browser, see structured diagnostics, repair the safe cases, and download the fixed spec. No upload. No server. No account.

The only honest way to test that idea is to ask the compiler.

What I tested

Native build first:

```
swift build
```

That passed with Apple Swift 6.2.

Then I tested the installed WASI SDK:

```
swift build \
  --target OpenAPIDoctor \
  --swift-sdk 6.2-RELEASE-wasm32-unknown-wasip1 \
  --scratch-path /tmp/openapidoctor-wasm-library
```

And the CLI shape:

```
swift build \
  --product openapi-doctor \
  --swift-sdk 6.2-RELEASE-wasm32-unknown-wasip1 \
  --scratch-path /tmp/openapidoctor-wasm-cli
```

Both failed before OpenAPIDoctor itself compiled.

The blocker

The first blocker is `Yams`, specifically its embedded `CYaml` C target:

```
error: unable to create target:
'No available targets are compatible with triple "wasm32-unknown-wasip1"'
```

I confirmed it directly by building `Yams` for the same SDK. It fails in the C parser files: `parser.c`, `reader.c`, `writer.c`, `api.c`, `emitter.c`, and `scanner.c`.

I also tried a tiny C compile with the same target:

```
clang --target=wasm32-unknown-wasip1
```

That failed with the same target-support error. So the current local toolchain cannot compile the C YAML dependency to WASI.

What this means

OpenAPIDoctor is not WebAssembly-ready as it exists today.

That does not kill the idea. It narrows the work.

The core shape is still good:

- `Validator.validate(yaml:)` already accepts an in-memory string.
- `Repairer.repair(yaml:)` already returns repaired YAML in memory.
- The native CLI is just a composition layer around those APIs.
- The right browser boundary is a WASI command: `stdin` in, diagnostics or files out.

The weak point is YAML infrastructure.

OpenAPIDoctor imports `Yams` in the validator, repairer, and synthesizer. It also uses `Stitcher` for multi-file refs, and Stitcher depends on Yams too. That means a browser build needs a YAML path that can cross to WASI.

The next path

There are three practical options.

1. Test again inside the Linux SwiftWasm toolchain we use for the TileDown playground builds. If that environment can compile C targets to WASI, then the next blocker will be runtime behavior.
2. Replace or isolate the YAML layer behind a WebAssembly-friendly interface. Single-file diagnosis could start with a smaller path before multi-file repair.
3. Add an OpenAPIDoctor WASI wrapper only after the dependency question is settled. The wrapper should read the spec from `stdin` and write JSON diagnostics to `stdout`.

The browser product should not use custom JavaScript exports. The better pattern is the same one that already works for the PDF playground:

```
stdin + argv + preopened files/directories -> stdout or output files
```

That keeps the Swift code shaped like a normal command-line program and lets the browser provide an in-memory filesystem.

Current verdict

OpenAPIDoctor is product-ready as a local-first browser idea, but not yet compiler-ready for WebAssembly.

The first concrete task is not UI. It is a dependency build proof for `OpenAPIKit`, `Yams`, and `Stitcher` under WASI. Only after that should the web tool be wrapped as a TileDown WASM tile or a standalone site.