Updates on WAGI, the WebAssembly Gateway Interface

Radu Matei

May 10, 2021

This article originally appeared on the Microsoft DeisLabs blog[1]

A few months ago, our team introduced WAGI[2], or the WebAssembly Gateway Interface, a simple way of writing and executing HTTP response handlers as WebAssembly modules. Since we open sourced the project, the community has been adding critical features that make WAGI one of the easiest ways to build WebAssembly microservices, and in this article we explore some of the new features.

A primer on WAGI

WAGI is for writing HTTP handlers. It uses WASI, the WebAssembly System Interface[3], and exposes HTTP request information to a WebAssembly module as environment variables and through standard input and output. This means that for any language that compiles to WASI, writing an HTTP handler that runs on WAGI is as easy as reading from `stdin` and writing to `stdout`.

Echoing the body of a request can be done by copying the standard input stream directly to standard output (or it can be read as a byte array and further processed), and in Rust it can be done in the following way:

```
use std::io::{copy, stdin, stdout};

fn main() {
    println!("Content-Type: text/plain\n");

    println!("Server received body:");
    copy(&mut stdin(), &mut stdout()).unwrap();
}
```

After compiling the module to `wasm32-wasi`, create a new WAGI configuration

[1] https://deislabs.io/posts/wagi-updates/

[2] https://deislabs.io/posts/introducing-wagi-easiest-way-to-build-webassembly-microservices/

[3] https://wasi.dev

file that sets the `/echo` route to execute the compiled WebAssembly module as an HTTP handler:

```
[[module]]
route = "/echo"
module = "target/wasm32-wasi/release/hello-wagi.wasm"
```

Then start WAGI by running `wagi --config wagi.toml`. At this point, WAGI can receive requests which will be delegated to the built module:

```
$ curl localhost:3000/echo -X POST -d 'Testing WAGI'
Server received body:
Testing WAGI
```

Any language that compiles to WASI can be used to write WAGI handlers, together with any external library for these languages that is compilable to `wasm32-wasi`. So far, our team has been testing with Rust, C/C++, AssemblyScript, Swift, Grain, and Zig, but more and more compiler toolchains are adding support for emitting WebAssembly.

Outbound HTTP connections

Networking in WASI is one scenario that doesn't have a stable API yet. This restricts the types of workloads that can be executed in WASI runtimes for now, which is why we released an experimental outbound HTTP library for WASI[4] that allows modules to create outbound HTTP connections.

Specifically, the published NPM package[5] and crate[6] can be used to send HTTP requests from AssemblyScript and Rust guest modules running in WAGI - for example, making a `GET` request and reading the response body as a string in AssemblyScript:

```
import { Console } from "as-wasi";
import { Method, RequestBuilder } from "@deislabs/wasi-experimental-http";

export function _start(): void {
  Console.write("Content-Type: text/plain\n");

  let res = new RequestBuilder("https://api.brigade.sh/healthz")
    .method(Method.GET)
    .send();
  let str = String.UTF8.decode(res.bodyReadAll().buffer);
  Console.write("Result from request: " + str);
}
```

[4] https://github.com/deislabs/wasi-experimental-http
[5] https://www.npmjs.com/package/@deislabs/wasi-experimental-http
[6] https://crates.io/crates/wasi-experimental-http

Notice how the module will attempt to make an HTTP request to `https://api.brigade.sh/healthz` - before starting WAGI with this module, we have to explicitly allow the guest module to make a request to this host using the `allowed_hosts` field in the WAGI configuration for this module:

```
[[module]]
route = "/as-http"
module = "as/build/optimized.wasm"
allowed_hosts = ["https://api.brigade.sh"]
```

Keep in mind that this library is experimental, and breaking changes could occur in the future.

Pulling modules from OCI registries

Given the widespread adoption of the container ecosystem, a natural choice for distributing WebAssembly modules intended to run outside the browser is OCI registries, and WAGI now supports referencing and pulling modules from supporting OCI registries (for a list of registries that have support for distributing Wasm modules, check the `wasm-to-oci` repository[7]).

We can take any compiled WebAssembly module and push it to one of the compatible registries using `wasm-to-oci`, a tool that leverages the ORAS[8] and OCI artifacts[9] projects - for example, pushing the module we just built to the GitHub Package Registry[10]:

```
$ wasm-to-oci push as/build/optimized.wasm ghcr.io/radu-matei/as-http:v1
INFO[0003] Pushed: ghcr.io/radu-matei/as-http:v1
INFO[0003] Size: 9004
INFO[0003] Digest: sha256:994c8adcef53e93
```

We can now modify the WAGI configuration to point to the newly pushed repository:

```
[[module]]
route = "/as-http"
module = "oci://ghcr.io/radu-matei/as-http:v1"
allowed_hosts = ["https://api.brigade.sh"]
```

After the first pull from the registry, WAGI will cache the module so the instantiation overhead is minimal.

Currently, the repositories have to enable anonymous pulls for WAGI to work,

[7] https://github.com/engineerd/wasm-to-oci
[8] https://github.com/deislabs/oras
[9] https://github.com/opencontainers/artifacts
[10] https://github.com/features/packages

but we are tracking the request to add authentication for registries[11]. As a workaround, `wasm-to-oci pull` can be used before starting WAGI to pull modules from authenticated repositories.

Besides using the local filesystem and OCI registries as sources for modules, WAGI can also pull artifacts from Bindle[12], although support for it is still early[13].

Declaring sub-routes in the module

WAGI now also allows modules to define their own sub-routes - specifically, in cases when we want a single module to handle more than one sub-route, by implementing and exporting a `_routes()` function in the module that maps routes to custom handler functions:

```
fn main() {
    println!("Content-Type: text/plain\n\n Hello from main()");
}

// Use no_mangle so we can call this
// from WAGI or other external tools.
#[no_mangle]
/// A provider function that can be
/// called directly
pub fn hello() {
    println!("Content-Type: text/plain\n\n Hello")
}

#[no_mangle]
/// Another provider function that can
/// be called directly.
pub fn goodbye() {
    println!("Content-Type: text/plain\n\n Goodbye")
}

// This maps a few routes:
// '/hello' will result in the `hello()`
// function being called.
// '/goodbye' and all subpaths of '/goodbye'
// will call the `goodbye()` function.
//
// Note that when compiled, the `main` function
// is named `_start()`. So if you want
// to map to that function, it is `/main _start`.
#[no_mangle]
pub fn _routes() {
    println!("/hello hello");
```

[11] https://github.com/deislabs/wagi/issues/41
[12] https://deislabs.io/posts/introducing-bindle/
[13] https://github.com/deislabs/wagi/pull/31

```
    println!("/goodbye/... goodbye");
    println!("/main _start");
}
```

Then, defining the top-level route for the module as `/example` in the WAGI configuration file:

```
[[module]]
route = "/example"
module = "/PATH/TO/hello_wagi.wasm"
```

After WAGI starts and build the routes, this module will map `/example` to `_start()`, `/example/hello` to `hello()`, and `/example/goodbye/...` to `goodbye()` (including wildcards).

You can find a full routing example here[14].

Using Azure services from Rust modules

Built on top of the outbound HTTP support, we have been experimenting with using various Azure services from WASI modules running in WAGI, using the Azure SDK for Rust[15] - specifically, using Azure Blob Storage (reading and writing blobs), Cosmos DB (reading and writing documents and collections), and EventGrid (sending messages and handling event subscriptions through HTTP webhooks).

We are excited by the prospect of having fully portable modules running in a Wasm sandbox that use Azure services, packaged at less than 3 MB (with room for further optimization). An initial set of samples for WAGI modules using Azure services can be found on GitHub[16], and we would love to hear your feedback about building and running them.

Optimizing compiled modules and caching We highly recommend using optimization tools to shrink the size of the resulting binary or speed-up the instantiation time. The following represents a non-exhaustive list of tools and techniques our team has been experimenting with:

- `wasm-opt` from Binaryen[17] - by removing debug symbols and unused module elements, together with an entire suite of instruction optimizers, `wasm-opt` is perhaps the most widespread tool, and depending on the toolchain used to compile modules, it can have exceptional results in shrinking the module size.
- `wizer` from the Bytecode Alliance[18] is a tool that pre-initializes WebAssembly modules - it *executes their initialization function, and then snapshots*

[14] https://github.com/technosophos/hello-wagi
[15] https://github.com/Azure/azure-sdk-for-rust
[16] https://github.com/deislabs/wagi-azure-samples
[17] https://github.com/WebAssembly/binaryen
[18] https://github.com/bytecodealliance/wizer

the initialized state out into a new WebAssembly module. If a module spends significant time initializing, it might be a good candidate for Wizer, and for specific types of workloads, it can yield startup times of up to 6x faster.

- Wasmtime caching[19] in WAGI - by default, Wasmtime, the runtime used by WAGI, will compile a WebAssembly module just-in-time (JIT) for the current architecture it is running on. Wasmtime caching writes the compiled module to a temporary cache and when instantiating a module, it checks whether an already compiled version of it already exists in the cache, resulting in significant startup speed improvements for large modules. WAGI provides a `--cache` flag pointing to a Wasmtime cache file, but keep in mind that for small modules, the overhead introduced by the filesystem access might increase the instantiation time.
- ahead-of-time compilation (AOT) with `wasmtime compile`[20] - as opposed to JIT compilation, with AOT compilation, Wasmtime creates a new module that is already compiled for the current architecture (or cross-compile). This means that before instantiating, Wasmtime will not have to perform a JIT compilation, nor access the filesystem cache to check if an already compiled version exists. This requires an extra step after pulling the module, but for large modules, the performance improvements are significant.

These are just a few potential optimization techniques that can be performed with WAGI, and real world usage indicates that, depending on the workload, language, and runtime, a combination of these could improve the binary size or startup time for a given module. If there are other optimization techniques we can enable for WAGI, let us know about them in WAGI's issue queue[21].

We would love to hear your feedback

WAGI is still in its early stages, but we are excited about the ways it could be used. We welcome all contributions that adhere to our code of conduct[22], and we are looking forward to your pull requests, issues, and suggestion.

[19] https://docs.wasmtime.dev/cli-cache.html
[20] https://github.com/bytecodealliance/wasmtime/pull/2791
[21] https://github.com/deislabs/wagi
[22] https://opensource.microsoft.com/codeofconduct/