Using Azure services from WebAssembly modules

Radu Matei

May 11, 2021

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

WAGI, the WebAssembly Gateway Interface[2], is a simple way of writing and executing HTTP response handlers as WebAssembly modules. We recently added new features to WAGI[3], such as pulling modules from OCI registries and outbound HTTP connections from guest modules, which opened the possibility of using Azure services from Wasm modules, and in this article we explore building and running such modules.

WAGI provides a very simple application model for building server-side HTTP response handlers using WebAssembly. Theoretically, any language that compiles to WASI, WebAssembly System Interface[4], can be used build a WAGI handler, since WAGI is modeled after the CGI specification[5] – read the request body from the process' standard input and environment variables and write the response to standard output. Since the request bodies can be read as byte arrays, modules can perform any computation on them, with the advantage of running the modules in the WASI isolation sandbox, together with an experimental HTTP library[6] that allows modules to send outbound HTTP connections.

But besides raw HTTP connections, most real-world scenarios will also regularly need to use external services such blob storage, databases, or message-passing systems, which is why we are experimenting with using a subset of the Azure SDK for Rust from WebAssembly modules – specifically, 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).

The latest work in progress for the Azure SDK compiled to WASI can be tracked here[7], and here is the repository containing WAGI samples that use Azure

[1] https://deislabs.io/posts/using-azure-services-wasi
[2] https://github.com/deislabs/wagi
[3] https://deislabs.io/posts/wagi-updates/
[4] https://wasi.dev
[5] https://en.wikipedia.org/wiki/Common_Gateway_Interface
[6] https://github.com/deislabs/wasi-experimental-http
[7] https://github.com/radu-matei/azure-sdk-for-rust/tree/enable-wasi-experimental-http

services[8]. Keep in mind all the building blocks these features are built on are experimental, so expect breaking changes in the future.

Currently, only modules built in Rust can use the Azure SDK from WebAssembly modules, but we are exploring ways in which to expose some common functionality to other languages that compile to WebAssembly as well. If you are interested in this work, feel free to reach out on GitHub.

Writing a blob to Azure Storage from a WASI module

Let's start from scratch. We will write a new WAGI handler that takes the request body and writes it to a new blob in Azure Storage. We will write it in Rust, compile it to `wasm32-wasi`, then run it in WAGI. To follow along, you will need a built binary of WAGI, and an Azure storage account and its key.

First, let's create a new Cargo project:

```
$ cargo new --bin wagi-azure-blob-storage-sample
```

We need two main dependencies in `Cargo.toml` (here is the complete Cargo.toml file[9]) – one that contains the core Azure SDK functionality, and the other that allows us to work with Azure Storage:

```
[dependencies]
azure_core = {
    git = "https://github.com/radu-matei/azure-sdk-for-rust",
    branch = "enable-wasi-experimental-http",
    features = ["enable_wasi_experimental_http"]
}
azure_storage = {
    git = "https://github.com/radu-matei/azure-sdk-for-rust",
    branch = "enable-wasi-experimental-http"
}
```

Notice that both repositories point to the current work-in-progress branch for the Azure SDK for Rust. We expect that once the project stabilizes, we would release crates that compile to `wasm32-wasi`.

Next, we need a function that, given the storage account and blob, access key, and bytes, creates a new blob:

```
pub async fn write_blob(
    container: String,
    blob: String,
    sa: String,
    key: String,
```

[8] https://github.com/deislabs/wagi-azure-samples
[9] https://github.com/deislabs/wagi-azure-samples/blob/main/Cargo.toml

```
    bytes: Vec<u8>,
    http_client: Arc<Box<dyn HttpClient>>,
) -> Result<()> {
    // create a new client for the given storage
    // account, container, and blob
    let blob_client = StorageAccountClient::new_access_key(
            http_client,
            sa,
            key
        ).as_storage_client()
         .as_container_client(container)
         .as_blob_client(blob);

    println!("Writing {} bytes.", bytes.len());

    // actually write some bytes to the blob.
    // the content type can be changed accordingly.
    blob_client
        .put_block_blob(bytes)
        .content_type("text/plain")
        .execute()
        .await?;

    Ok(())
}
```

This is just using the Rust SDK without any changes. Now we need to get the blob and container names from the request's query string, the actual body to write from the request body, and the access keys from environment variables:

```
pub async fn run() -> Result<()> {
    // read the container and blob names
    // from the request's query string
    let (container, blob) = container_and_blob_from_query()?;
    // read the storage account and key
    // from environment variables
    let (sa, sa_key) = keys_from_env()?;

    // create a new WASI HTTP client.
    // this is the only different part compared
    // to the regular samples for the Rust SDK
    let http_client: Arc<Box<dyn HttpClient>> = Arc::new(
        Box::new(WasiHttpClient {})
    );

    // copy the request body into a new byte array
    let mut buf = Vec::new();
    std::io::copy(&mut std::io::stdin(), &mut buf)?;
    // write the blob
    write_blob(
        container.clone(),
        blob.clone(),
        sa,
        sa_key,
        buf,
        http_client.clone(),
    )
    .await?;

    Ok(())
}
```

As the comment suggests, the only part that is different compared to the samples from the SDK repo[10] is the creation of a new WASI HTTP client, which is then used as argument for all functions that communicate with Azure services.

Finally, the entrypoint to this program sets the content type of the response, then blocks on the execution of the `run()` function:

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

    block_on(run()).unwrap();
}
```

Now we need to compile the Rust program to WASI, `cargo build --release --target wasm32-wasi`, and at this point, we need to set up the WAGI configuration, `wagi.toml`. For it, we need an Azure storage account, and we need to set three things: the storage account name, an access key with write access to a container, and the URL of the storage account where the SDK will make HTTP requests to:

```
[[module]]
route = "/handler"
module = "target/wasm32-wasi/release/wagi-azure-blob-storage-sample.wasm"
environment = { STORAGE_ACCOUNT = "<sa>", STORAGE_MASTER_KEY = "<sa-key>" }
allowed_hosts = ["https://<sa>.blob.core.windows.net"]
```

Starting WAGI on port 3000 and sending a new request using cURL:

```
$ curl 'localhost:3000/handler?container=<container>&blob=new-article-test' -X
POST -d 'Using Azure services from WebAssembly modules is awesome!'
Writing 57 bytes.
```

[10] https://github.com/Azure/azure-sdk-for-rust

We can check the blob was actually written using the Azure CLI:

```
$ az storage blob list --container <container> --account-name <sa>
Name                Blob Type    Blob Tier    Length    Content Type
-----------------   -----------  -----------  --------  --------------
new-article-test    BlockBlob    Hot          57        text/plain
```

Reading the blob from a WebAssembly module can be done similarly, and a complete example can be found in the samples repository[11].

Using Cosmos DB

Cosmos DB can also be used from WebAssembly modules running in WAGI:

```
// create a new client based on the account and key
let token = AuthorizationToken::primary_from_base64(&key)?;
let client = CosmosClient::new(http_client, account, token)
    .into_database_client(database)
    .into_collection_client(collection);

// create a new document in the collection
client
    .create_document()
    .is_upsert(true)
    .execute(doc)
    .await?;


// query the current collection for items with a given ID
let query = format!("SELECT * FROM c where c.id = '{}' ", some_id);
let res = client.query_documents().execute::<T, _>(&query).await?;
```

Similarly, the account and keys have to be passed as environment variables, and the endpoint for the Cosmos DB account has to be on the list of allowed domains the module can send HTTP requests to.

Check the Azure SDK for Rust[12] for a list of operations currently implemented for Cosmos DB.

Passing messages using EventGrid

There are two aspects to message passing with EventGrid – sending and receiving messages, and with WAGI, we can handle both.

Sending a message is done by using the EventGrid client from the Rust SDK:

[11] https://github.com/deislabs/wagi-azure-samples
[12] https://github.com/Azure/azure-sdk-for-rust

```rust
pub async fn send_message<T: Serialize>(
    host: String,
    key: String,
    events: Vec<Event<T>>,
    http_client: Arc<Box<dyn HttpClient>>,
) -> Result<()> {
    // create a new client for the host
    let client = EventGridClient::new(host.clone(), key, http_client);
    // send an array of messages
    client.publish_events(&events).await?;
    println!("Sent message to host {}", host);
    Ok(())
}
```

We can extend the previous example that writes a blob by sending a message with the information about the blob that was just written:

```rust
    let events = vec![Event::new(
        ...
        EventData { container, blob },
    )];

    send_message(host, host_key, events, http_client).await?;
```

Any event subscription can get the information that a new blob was written in the storage container, and we can configure a new EventGrid webhook subscription to be handled by a WebAssembly module running in WAGI:

First, we have to understand how to validate the new endpoint[13] – when first setting the WAGI endpoint as the subscription webhook, EventGrid will send a validation event, and the handler will reply with the validation response.

Then, for all other event types, we can handle them in any way we need, including using any Azure SDK compilable to WebAssembly.

```rust
async fn run() -> Result<()> {
    // Event Grid sends the events to subscribers
    // in an array that has a single event.
    // https://docs.microsoft.com/en-us/azure/event-grid/event-schema
    let event: Vec<Value> = serde_json::from_reader(&mut stdin())?;
    let event = event[0].clone();

    match get_event_type(event.clone())? {
        EventType::Validation(code) => {
            let val = json!({
                "validationResponse": code,
            });
```

[13] https://docs.microsoft.com/en-us/azure/event-grid/webhook-event-delivery

```
                println!("{}", val);
                return Ok(());
            }
            EventType::BlobCreated => return handle_blob_created_event(event).await,
            EventType::Custom(ev) => {
                panic!("unknown event {}", ev)
            }
        }
}
```

Size, distribution, caching

First, let's have a look at the size of the module:

```
$ ls target/wasm32-wasi/release/wagi-azure-blob-storage-sample.wasm
2.8M  target/wasm32-wasi/release/wagi-azure-blob-storage-sample.wasm
```

This is the module generated by the Rust compiler. We can further optimize it by running `wasm-opt` from Binaryen (this article explains a few optimization techniques[14] that should be helpful when running WebAssembly modules.):

```
$ wasm-opt wagi-azure-blob-storage-sample.wasm -O4 -o mod.wasm
$ ls
2.3M mod.wasm
2.8M wagi-azure-blob-storage-sample.wasm
```

We can obtain a 2.3 MB module just by running `wasm-opt`, and since WAGI supports pulling from OCI registries, we can push the optimized module to a supporting registry using `wasm-to-oci`[15], and pull it from there at startup:

```
$ wasm-to-oci push mod.wasm ghcr.io/radu-matei/write-azure-blob-wasi:v1
INFO[0006] Pushed: ghcr.io/radu-matei/write-azure-blob-wasi:v1
INFO[0006] Size: 2251461
INFO[0006] Digest: sha256:62b44dc8e4e6
```

Then change the WAGI configuration accordingly:

```
[[module]]
route = "/handler"
module = "oci://ghcr.io/radu-matei/write-azure-blob-wasi:v1"
...
```

Let's have a look at the total response time for a request:

```
$ time curl '<endpoint>/handler?container=<>&blob=<>' -X POST -d '...'
```

[14]https://deislabs.io/posts/wagi-updates/#optimizing-compiled-modules-and-caching
[15]https://github.com/engineerd/wasm-to-oci

```
Writing 57 bytes.
0.00s user 0.00s system 1% cpu 0.423 total
```

For this particular example, on my hardware, I get an average response time of around 400 ms. Where is that coming from? The majority of that is coming from actually executing the WebAssembly module, and from making external HTTP requests to Azure, and depending on the datacenter location and service used, that latency can also be improved. But there is another significant source of latency – the instantiation time for the WebAssembly module:

```
wagi::runtime] instantiation time for module
oci://ghcr.io/radu-matei/write-azure-blob-wasi:v1: 90.618ms
```

Particularly, when instantiating a new module, Wasmtime will compile the module just-in-time (JIT) for the current architecture and platform, but we can use a cache of the compiled module by creating a Wasmtime cache[16] file and configuring WAGI to use it:

```
[cache]
enabled = true
directory = "<absolute-path-to-a-temporary-cache>"
# optional
# see more details at https://docs.wasmtime.dev/cli-cache.html
cleanup-interval = "1d"
files-total-size-soft-limit = "10Gi"
```

Now we can test the endpoint again, and we can see that the total response time dropped significantly:

```
$ time curl '<endopoint>/handler?container=<>&blob=<>' -X POST -d '...'
Writing 57 bytes.
0.00s user 0.00s system 1% cpu 0.216 total
```

Checking the WAGI logs, we see the instantiation time of the WebAssembly module went from 90 ms to 10 ms:

```
wagi::runtime] instantiation time for module
oci://ghcr.io/radu-matei/write-azure-blob-wasi:v1: 10.5992ms
```

For context, a simple “hello world” module written in Rust, with the same optimization and caching settings as before is instantiated in around 2.5 ms:

```
wagi::runtime] instantiation time for module
test.wasm: 2.5489ms
```

[16]https://docs.wasmtime.dev/cli-cache.html

Depending on the workload, services used, and hardware, you will see different numbers, but the optimization and caching technique should help reduce the overall latency. Our team is exploring ways in which we can continue to reduce the instantiation times, and we are extremely excited about new proposed Wasmtime API[17], particularly in the context of multi-threading[18].

Conclusion

In this article we saw how to get started using Azure services from WebAssembly modules. This is currently experimental, but we are working towards adding more and more services to the compatible SDK, and we would love to hear from you.

Moreover, we managed to build, a module less than 3 MB in size that uses Azure services, and is run in an isolation sandbox, with an instantiation time of around 10 ms. WAGI is still in its early stages, but we are very excited about the ways it can be used to build and run microservices, and being able to use Azure services opens up a lot of interesting opportunities.

[17] https://github.com/bytecodealliance/rfcs/pull/11
[18] https://docs.wasmtime.dev/examples-rust-multithreading.html