Building a Wasmtime host implementation for a WebAssembly interface

Radu Matei, January 2022

The WebAssembly component model defines *a portable, load- and run-time-efficient binary format […] that enables portable, cross-language composition*, effectively enabling a wide range of scenarios for WebAssembly modules built in different programming languages to interoperate. The previous article in this series showcased how to start from a WebAssembly interface, implement it as a WebAssembly component in Rust, then consume the interface and its Rust implementation from other Wasm components built in C++ and Rust.

Satisfying interfaces through other WebAssembly components means they will be executed in the same WebAssembly sandbox, and will not get access to anything outside the sandbox — this means no default access to any host APIs, not hardware acceleration support, or no networking. So how could we build an implementation of an interface, get access to resources that are not available in the WebAssembly sandbox, but still ensure that we only expose the functionality needed for the interface we are implementing?

We could build a host implementation. We could use to all host capabilities (hardware acceleration, networking), and transparently expose them to guest modules through regular Wasm imports. Because we are implementing the same interface, users of it (consumer modules, such as the components from the previous article) don't have to change at all, but the host runtime can be updated, or potentially patched for a security vulnerability without recompiling the consumers of the interface.
Moreover, the host implementation has complete control over how to restrict the guest's access to resources — it can reject outgoing connections based on the domain, or file access based on the relative paths. The choice is up to the host implementation, not the guest module.

There are existing examples for this approach — WASI itself (for enabling host access to things like the filesystem or environment), WASI-NN (for high performance machine learning inference), or WASI experimental HTTP (for performing outbound HTTP requests from Wasm modules).

The current tooling for the component model generates bindings that make it easier to write such host implementations (in Rust and Python, using the Wasmtime API), and this article will explore this topic by extending the project from the previous article to use with a host implementation for a cache interface that stores the key/value pairs in Redis.

Let's start by having another look at the interface. Consumers will use language bindings based on this API to access the caching functionality. The interface contains functions for setting, getting, and deleting values from a cache:

```
// cache.wit
// Type for cache errors.
enum error {
    runtime-error,
    not-found-error,
}
```

```
// Payload for cache values.
type payload = list<u8>
// Set the payload for the given key.
set: function(key: string, value: payload, ttl: option<u32>) -> expected<_, error>
// Get the payload stored in the cache for the given key.
get: function(key: string) -> expected<payload, error>
// Delete the cache entry for the given key.
delete: function(key: string) -> expected<_, error>
```

(A non-trivial example of using the new WIT format can be found here.)
We want to have this host implementation in Rust, so we start with a new library crate:

```
$ cargo new --lib wasmtime-impl-redis
  Created library `wasmtime-impl-redis` package
```

The main dependency needed is `wit-bindgen-wasmtime` — a Bytecode Alliance project that, given a WebAssembly interface (WIT) file, generates Rust bindings that make it easy to implement (or consume) the interface from Wasmtime using its Rust API:

```
// Cargo.toml
[dependencies]
    wit-bindgen-wasmtime = { git = "https://github.com/bytecodealliance/wit-bindgen", rev =
"2e654dc82b7f9331719ba617a36ed5967b2aecb0" }
```

In general, generators from `wit-bindgen` have two operating modes, *import* and *export*:

- import bindings are for *consuming* an interface
- export bindings are for *implementing* an interface.

In this case, because we want to offer a host implementation for the cache interface, we need to generate *export* bindings.

The most important part here is the `wit_bindgen_wasmtime::export!` procedural macro, which takes the interface file as input, and automatically generates Wasmtime-specific bindings to implement the interface in Rust. This is similar to directly using the CLI:

```
$ wit-bindgen wasmtime --export ../cache.wit
Generating "bindings.rs"
```

Inspecting the generated bindings, we can see types for the objects from the interface, a top-level `add_to_linker` function that allows linking all the functions from this implementation, and a trait that contains all functions we must implement. Let's look at the implementation for the `cache::Cache` trait and how to implement it:

```rust
use cache::*;
wit_bindgen_wasmtime::export!("../cache.wit");

struct RedisCache {}

impl cache::Cache for RedisCache {
    fn set(
        &mut self,
        key: &str,
        value: PayloadParam<'_>,
        ttl: Option<u32>,
    ) -> Result<(), Error> {
        todo!()
    }

    fn get(&mut self, key: &str) -> Result<PayloadResult, Error> {
        todo!()
    }
...
}
```

An important thing to note here is that the trait functions have an associated mutable receiver (`&mut self`), which indicates a way of keeping and mutating state with the `RedisCache` structure (that will live throughout the execution of its Wasm caller).

At this point, the respective functions must use a Redis instance to store and retrieve values — this can be achieved by modifying the cache structure to have a reference for the Redis instance address and a client connection:

```rust
/// Redis implementation for the WASI cache interface.
pub struct RedisCache {
    /// The address of the Redis instance.
    pub address: String,

    /// The Redis connection.
    connection: Connection,
}
```

Next, you can implement the required functions by using the Redis crate to store and retrieve data. For example, for the `set` function, we store the value associated with the key, and check whether there is a time-to-live argument passed — if present, set the expiry period in Redis:

```rust
/// Set the payload in Redis using the given key and optional time-to-live (in seconds).
fn set(&mut self, key: &str, value: &[u8], ttl: Option<u32>) -> Result<()> {
    self.connection.set(key, value)?;
    match ttl {
        Some(s) => self.connection.expire(key, s as usize)?,
```

```
        None => {}
    };
    Ok(())
}
```

The big advantage of using the WIT bindgen macro is that it abstracts almost all of the WebAssembly specific boilerplate, and the only remaining task is implementing the actual functionality.

Linking the implementation at runtime

Now that the host implementation is built, it is time to start instantiating the consumer components that were built in the previous part of this series — but this time, instead of satisfying their interface import through another WebAssembly component, we will satisfy it through the host implementation built with Redis.

The entire process for setting up the runtime and creating an instance of a WebAssembly module with Wasmtime can be found on GitHub and in the Wasmtime Rust API documentation, but the relevant part to this article is linking the host implementation to satisfy the cache interface the consumer component is importing:

```
// create an instance of the RedisCache object we just built
let rc = RedisCache::new("redis://localhost:6379")?;

// use the auto-generated add_to_linker method to register the associated functions
// whenever interface functions are executed during this Wasm caller's lifetime
cache::add_to_linker(&mut linker, |ctx| -> &mut RedisCache {
       ctx.runtime_data.as_mut().unwrap()
})?;

...
// create a new instance of the module and call its entrypoint
let instance = linker.instantiate(&mut store, &module)?;

let start = instance.get_func(&mut store, "_start").unwrap();
start.call(&mut store, &[], &mut [])?;
```

The procedural macro we used when implementing the library also generated the `add_to_linker` method, which abstracts the entire process of linking the interface methods to the implementation. After creating a new instance of `RedisCache`, the state is added to the current Wasmtime store, and the module is instantiated.
Finally, the `_start` function from the module (the default entrypoint for WASI modules) is called, and the consumer module is executed.
The complete code for the host implementation and test suite can be found on GitHub.

If you are interested in contributing to a standard set of WebAssembly interfaces, check out this repository from Fermyon.

Conclusion

In this article we explored using Bytecode Alliance tooling to build a host implementation for a WebAssembly interface, and used it to satisfy the imports of a previously built component at runtime. Using this tooling and Wasmtime, we can dynamically instantiate components by using either other Wasm components, or host implementations, based on runtime constraints or environment decisions.

The early stages of the WebAssembly component model show tremendous potential to enable new types of computing workloads, and further developments in the tooling will start unlocking this potential.