Introduction to WebAssembly components

Radu Matei, December 2021

WebAssembly (or Wasm) is a W3C specification for a portable binary format for distributing and running code that has been implemented in the four major browser engines since 2017. In familiar terms, Wasm is used as a compilation target for various programming languages, generating a compact binary that can run at near-native speeds in the browser. This brought existing languages such as Rust, C and C++, Go, or C# (and new languages like Grain) to the web, and enabled porting extremely complex applications such as Google Earth or Photoshop to the browser.

Despite the name, however, nothing in WebAssembly is specific to browsers — and in fact, the same benefits that make it a compelling execution environment for browsers (such as near-native speed, compact binary format, or sandbox isolation) make it well-suited for scenarios outside the browser, in datacenters, clouds, or on the edge. The WASI project, or the WebAssembly System Interface, is a proposal that aims to standardize the execution of Wasm outside the browser and to provide a common (platform agnostic) layer and set of primitives that guest modules can use to interact with the underlying runtime, while maintaining the secure sandbox promised by WebAssembly. (Lin Clark's initial post announcing the Bytecode Alliance does a fantastic job at explaining the goals of WASI.)

WebAssembly and WASI show great promise for the future of computing outside the browser — but attempting to write any non-trivial WebAssembly application that tries to interoperate across runtime or language boundaries requires significant effort today, and exchanging any non-fundamental data types (such as strings or structures) involves pointer arithmetic and low-level memory manipulation.

The component model proposal aims to solve this issue, and this article will explain the goals of the proposal and will showcase how to use the current tooling from the Bytecode Alliance to build and execute such components in Rust and C++. (Note: The demo components, the implementations, the tools used, and the developer experience showed here represent very early attempts to solve this, and future tooling will improve it. This is shown for educational purposes, and should not be considered stable.)

The WebAssembly component model

Using an operating system analogy, WebAssembly allows the execution of low-level CPU instructions, while WASI is a way to model input/output interfaces. From this perspective, the need for a "process model" that defines how processes are started and how they interact with each other is starting to emerge — this is what the WebAssembly component model proposal is trying to address.

The first stated goal of the component model is *to define a portable, load- and run-time-efficient binary format […] that enables portable, cross-language composition* – effectively, addressing how multiple components can interact with each other, and the use cases describe a wide range of scenarios for embedding components, composition, and dynamic linking.

The main use case this article addresses is the following — defining an API layer as a WebAssembly interface, implementing it as a WebAssembly component, then consuming it from other components by passing arguments and return values. There are numerous other topics to explore in this area such as transitive dependencies, distribution, developer experience, or building specialized host runtimes for a given interface, all of which will be addressed in future articles.

Defining and implementing WebAssembly components

The goal is to build a component that can be imported from other WebAssembly modules, written in potentially other programming languages, and the first step is defining its interface — what is the public API this component will implement? This is done using WIT (WebAssembly Interface), an experimental textual format used for defining Wasm interfaces. It is the next iteration of WITX, which itself is based on the standard text format. (A non-trivial example of using the new WIT format can be found here.)

The component is going to be a simple key/value cache layer that gets, stores, and deletes arbitrary payloads:

```
// 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>
```

Let's implement this interface in Rust, using the file system as storage for the cache:

```
$ cargo new --lib rust-wasi-impl
  Created library `rust-wasi-impl` package
```

Next, the only dependency needed is `wit-bindgen-rust` – a Bytecode Alliance project that generates Rust bindings given a WIT interface:

```
// Cargo.toml
[lib]
    crate-type = [ "cdylib" ]

[dependencies]
    wit-bindgen-rust = { git = "https://github.com/bytecodealliance/wit-bindgen", rev =
"32e63116d469d8046727fae3c1333a7d35d0c5d3" }
```

The next section contains a simplified version of the actual implementation (note that the complete implementation for all components can be found on GitHub). A very important part here is the `wit_bindgen_rust::export!` procedural macro — it takes the interface file as input, and it automatically generates Rust bindings for all the objects defined in the interface, bindings necessary to implement the interface.

This is equivalent to using the `wit-bindgen` CLI to manually generate the bindings (to check in to source control, or inspect):

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

Inspecting the generated bindings, we can see the low-level code (that until now had to be manually written) to handle passing non-fundamental data types between modules, with the canonical ABI described in the interface types proposal.

Rust's excellent macro support means the bindings can be dynamically generated from the interface at build time. Regardless of how the bindings are generated, the main piece to implement here is a Rust trait that models the API from the interface:

```
// lib.rs
wit_bindgen_rust::export!("../cache.wit");

struct Cache {}
impl cache::Cache for Cache {
    fn set(key: String, value: Payload, _: Option<u32>) -> Result<(), Error> {
        let mut file = File::create(path(&key)?)?;
        file.write_all(&value)?;
        Ok(())
    }

    fn get(key: String) -> Result<Payload, Error> {
        let mut file = File::open(path(&key)?)?;
        let mut buf = Vec::new();
        file.read_to_end(&mut buf)?;
        Ok(buf)
    }
 ...
}
```

(Note that at the time of writing this article, the convention for Rust implementations is that the struct implementing the interface trait must have the same name (accommodating for `snake_case`) as the interface file, hence `struct Cache {}`. There are also a few error handling specific parts omitted from the snippet above, see complete implementation.)

The actual implementation is straightforward – store and retrieve keys/value pairs as files in the file system (assuming this component has the capability to write to a filesystem).

At this point, the Wasm module can be built using the Rust toolchain. Then, using the translator from the binary format to the text format (from this repo), we can see the module exports the three methods from the interface, together with functions to adapt the arguments passed between boundaries (as described by the canonical ABI):

```
$ cargo build --target wasm32-wasi --release
$ wasm2wat-rs target/wasm32-wasi/release/rust_wasi_impl.wasm | grep export
  (export "set" (func $set.command_export))
  (export "get" (func $get.command_export))
  (export "delete" (func $delete.command_export))
  (export "canonical_abi_realloc" (func $canonical_abi_realloc.command_export))
  (export "canonical_abi_free" (func $canonical_abi_free.command_export))
```

Importing WebAssembly interfaces in Rust and C++

The previous section defined an interface using WIT, then implemented it in Rust using the convenient macros provided by `wit-bindgen`. This section will create two new components, in Rust and C++, which will import the interface.

First, a new Rust executable with the same Cargo dependency as the previous component:

```
$ cargo new --bin rust-consumer
```

As in the previous Rust component, the main aspect here is the use of the `wit_bindgen_rust::import` procedural macro — same as before, the macro takes the interface and generates Rust bindings, but crucially, because this component *imports* the interface, the bindings will be different (they can be inspected by executing `wit-bindgen rust-wasm --import ../cache.wit`):

```
wit_bindgen_rust::import!("../cache.wit");

fn main() {
    let key = "five-good-emperors";
    let value = "Nerva, Trajan, Hadrian, Pius, and Marcus Aurelius";

    cache::set(key, value.as_bytes(), None).unwrap();
    let ret = cache::get(key).unwrap();
    assert_eq!(ret, value.as_bytes());
}
```

The generated import bindings can be used in a very idiomatic way to set and retrieve information.

The important thing to note here is that the program above only needs the *interface* in order to compile, as the generated WebAssembly module will contain imports for the cache functionality:

```
$ cargo build --target wasm32-wasi --release

$ wasm2wat-rs target/wasm32-wasi/release/rust-consumer.wasm | grep import
```

```
  (import "cache" "set" (func $rust_consumer_cache_set_wit_import (type 8)))
  (import "cache" "get" (func $rust_consumer_cache_get_wit_import (type 9)))
  (import "wasi_snapshot_preview1" "fd_write" (func $wasi_wasi_fd_write (type 10)))
```

Before actually linking and executing the main module above, it is worth exploring how to build another consumer, this time in C++.

Because C++ doesn't have the same macro system as Rust, the bindings need to be on disk at compile time — using `wit-bindgen`,(and generating *import* bindings, as the C++ component will import the interface), they are written into a `bindings/` directory:

```
# Makefile
bindgen:
    $(WIT_BINDGEN) c --import ../cache.wit --out-dir bindings

build:
    $(WASI_CC) -I . -I ./bindings -c -o cache.o bindings/cache.c
    $(WASI_CC) main.cpp cache.o -o cpp_consumer.wasm
```

At this point, the implementation is a C++ main program that uses the header file defined in `bindings/cache.h` and calls the functions to get and set key/value pairs:

```
#include "bindings/cache.h"

int main(int argc, char **argv)
{
    char *key = "almost-consul";
    char *value = "Caligula's horse, Incitatus";
    printf("Writing contents `%s` in storage `%s`", value, key);

    cache_string_t *skey;
    skey->len = strlen(key);
    skey->ptr = key;

    cache_payload_t *svalue;
    svalue->len = strlen(value);
    svalue->ptr = (uint8_t *)value;

    cache_set(skey, svalue, NULL);

    cache_payload_t *ret;
    cache_get(skey, ret);
    printf("Retrieved from `%s`: `%s`", key, (char *)ret->ptr);
    assert(svalue->len == ret->len);
}
```

The rest of the implementation is adapting the character arrays where the key and value are stored into the types expected by the interface. Finally, this can be compiled, and exploring the resulting module's imports, the same imports from a cache module can be seen:

```
$ make bindgen build
$ wasm2wat-rs cpp_consumer.wasm | grep import
  (import "wasi_snapshot_preview1" "proc_exit" (func $__wasi_proc_exit (type 2)))
  (import "cache" "set" (func $__wasm_import_cache_set (type 3)))
  (import "cache" "get" (func $__wasm_import_cache_get (type 4)))
```

Linking and executing components

The previous sections defined the interface, built an implementation for it, then imported the interface in two Rust and C++ programs, resulting in two WebAssembly modules with imports that must be satisfied before they can be instantiated. This section will link them with the actual component implementation using `wasmlink`, a CLI that allows us to statically link *a module and its dependencies using module linking and the Canonical Interface Types ABI*.

Starting with the C++ module that imports the interface, before executing it, its cache imports must be satisfied — this is currently done manually using `wasmlink`:

```
link:
    $(WASMLINK) cpp_consumer.wasm \
        --interface cache=../cache.wit \
        --profile wasmtime \
        --module cache=../rust-wasi-impl/target/wasm32-wasi/release/rust_wasi_impl.wasm \
        --output linked.wasm

run:
    $(WASMTIME) --enable-module-linking --enable-multi-memory --mapdir=/cache::. linked.wasm
```

The link target uses the `wasmlink` CLI to supply the Rust implementation of the interface whenever the `cpp_consumer.wasm` imports anything from the `cache` module. The linker is also generating WebAssembly code responsible for adapting the data between the linear memories of each component, meaning that no component can access another component's memory directly, ensuring a "shared-nothing" approach.

The output of this target is a statically linked module that contains an inline copy of the Rust implementation for the cache interface. (The various imports and exports of the final linked module can be explored using `wasm2wat-rs`.)

Finally, this can be run using Wasmtime (with support for the module linking and multi memory proposals enabled, and with granting the module the ability to use the filesystem, where the cache implementation stores its data):

```
$ make link run
wasmtime --enable-module-linking --enable-multi-memory --mapdir=/cache::. linked.wasm
Retrieved from `almost-consul`: `Caligula's horse, Incitatus`
```

The same commands can be run for the Rust consumer:

```
$ make link run
wasmtime --enable-module-linking --enable-multi-memory --mapdir=/cache::. linked.wasm
Retrieved from five-good-emperors: Nerva, Trajan, Hadrian, Pius, and Marcus Aurelius
```

Right now, linking is a manual operation, but as the tooling and language support evolves, this will be significantly improved.

Conclusion

This article explored the new WebAssembly component model proposal and demonstrated a very early way of using interfaces, building Rust and C++ components, linking, and running them with Wasmtime. There is a great opportunity for improving the developer experience for building, consuming, and linking Wasm component, and future articles will showcase the improvements done together with the community, as well as other areas such as language toolchain integration, or the distribution of components.

As more programming languages add WebAssembly as a compilation target, and as tooling is built that automatically generates bindings for those programming languages, the component model will enable true potable and cross-language composition for software.