WASM COOKING WITH GOLANG

by Philippe Charrière | @k33g_org

Bots.Garden Editions

Wasm Cooking with GoLang

Philippe Charrière

To everyone aspiring to have fun when writing software.

Acknowledgments

🚧 This is a work in progress

Copyrights

1. Recipe 17: Wasmtime CLI

What are you going to learn?

These are our first steps to "free" the wasm programs from the JavaScript hosts:

- How to execute a wasm application from the command line
- How to invoke a wasm function from the command line

Wasmtime is a project hosted by the Bytecode Alliance. **Wasmtime** notably allows you to run wasm programs with a CLI and write your own CLI or to call functions present in your wasm libraries. **Wasmtime** offers SDKs to run wasm applications or call functions from Rust, Python, Go, .Net languages.

You can find more information about wasmtime on:

- https://wasmtime.dev/
- https://github.com/bytecodealliance/wasmtime

1.1. Preamble

If you're using the Gitpod project, create a new project from the template https://gitlab.com/wasmcooking/gitpod-golang-project-template as explained at the start of this ebook.

1.1.1. Wasmtime installation

First, we need to install the Wasmtime CLI. On Linux and MacOS (or inside of the Gitpod workspace), use the below commands:

```
curl https://wasmtime.dev/install.sh -sSf | bash
source $HOME/.bashrc
```

You can download installers on https://github.com/bytecodealliance/wasmtime/releases and you'll find a Windows version.

Then create a new project and name it `wasmtime-cli` for this new recipe and add this files:

- `go.mod`
- `main.go`

1.2. `go.mod`

In the `go.mod` file, change the name of the module:

go.mod

```
module hello-cli

go 1.17
```

1.3. `main.go`

We already used a similar sample in a previous recipe:

main.go

```
package main

import (
    "fmt"
    "os" ①
)

func main() {

    fmt.Println("Hello World from Go")
    args := os.Args ②
    argsWithoutCaller := os.Args[1:] ③

    fmt.Println(args)
    fmt.Println(argsWithoutCaller)

}
```

① **os** package provides an array named **Args**

② The **os.Args** array contains all the passed command-line arguments. The first item of the array is the name of the program itself.

③ Get all the arguments, except the program name

1.4. Build and run

We have a problem with the Go compiler and its WebAssembly support: it does not support WASI (WebAssembly System Interface). The short story: the produced wasm file needs to use the **wasm_exec.js** file provided by the Go toolchain, an then it cannot run outside a JavaScript host.

Fortunately, TinyGo can compile Go program to WebAssembly with a **wasi** target:

```
tinygo build -o hello.wasm -target wasi ./main.go
```

Now, to run it, use this command:

```
wasmtime hello.wasm Bob Morane
```

You should get this output:

```
Hello World from Go
[hello.wasm Bob Morane]
[Bob Morane]
```

1.5. Call a function

It's possible to invoke a Go function from the Wasmtime CLI. Update your source code like this:

main.go

```
package main

import (
    "fmt"
    "os"
)

//export add ①
func add(x, y int) int {
    return x + y
}

func main() {

    fmt.Println("Hello World from Go")
    args := os.Args
    argsWithoutCaller := os.Args[1:]

    fmt.Println(args)
    fmt.Println(argsWithoutCaller)

}
```

① Export the function

The **add** function have to be exported to be callable by the Wasmtime CLI. TinyGo supports a comment directive allowing that: **`//export <function_name>`**.

1.5.1. Build and run

```
tinygo build -o hello.wasm -target wasi ./main.go
```

Now, to run it, use the former command but with the **`--invoke`** flag, the name of the function, the wasm file and the parameters:

```
wasmtime --invoke add hello.wasm 12 30
```

You should get this output:

```
warning: using `--invoke` with a function that takes arguments is
experimental and may break in the future
warning: using `--invoke` with a function that returns values is
experimental and may break in the future
42 ①
```

① **`12+30=42`**

Of course, you cannot miss the two weird messages about the experimental status of the **`invoke`** feature. The WASI specification is in progress, and the WebAssembly specification only supports a very limited number of data types, but it should change. Waiting for that, everything about "types" is more or less "experimental". A **Interface Types Proposal** is in progress (you can follow it here: https://github.com/WebAssembly/interface-types/blob/main/proposals/interface-types/Explainer.md), once ready, we'll be able to use "high-level values" as parameters and returns value.

Many other projects intended to run wasm outside a JavaScript host exist, such as Wamser, WasmEdge, Wasm3, etc.

Even if the notion of types limits us (but there are already some solutions), we will see in the next section that it is already possible to obtain "production-ready" applications.

But first, we will play again, this time with the WasmEdge runtime and see how to create our own CLI and even pass strings to Wasm functions even with Wasi.