Revisiting WebAssembly with Go

It's been a while since I've looked at WebAssembly and Go. Some things have changed since then and I want to document as I revisit them.

Some of these changes deal with the evolution of Go as a programming language, some of them deal with Go's support for Webassmebly, and some have dealt with the evolution of WebAssembly.

First project: Building a template

Before we start looking at specific projects, let's look at what we need to do to compile Go to WebAssembly.

The first step is to initialize the project from GO. We do this by running the following command:

```
go mod init go-project
```

The resulting go.mod will look like this:

```
module go-project

go 1.17
```

As we install more packages, they will be added to `go.mod` so that when people build and run our code they will be in sync with the packages the author used to write it.

`hello-world` is the simplest example we can write and run

```
package main
```

```
import (
  "fmt"
)

func main() {
  fmt.Println("Hello World from Go")
}
```

The compilation itself is simple:

```
GOOS=js GOARCH=wasm go build -o main.wasm
```

This is similar to how we crosscompile Go to platforms other than the one we're compiling on.

`GOOS` defines the target operating system. in this case we choose `js` which is the target OS for WebAssembly.

`GOARCH` defines the target architecture. The target for WebAssembly is `wasm`.

We the two flags set, we rung go build to compile the code, we use the `-o` to specify the name of the output file, and we use the `.wasm` extension to indicate this is a WebAssembly file.

The next part is to load the glue Javascript code into the browser.

`wasm_exec.js` is available with the Go installation. You need to copy it into the js directory of your project with the following command:

```
cp "$(go env GOROOT)/misc/wasm/wasm_exec.js" ./js
```

The Javascript is straight forward. I'm experimenting with fastify to build a simple HTTP server. You could do it with Express or any other framework of your choosing

```
const fastify = require('fastify')({ logger: true })
const path = require('path')
```

```
'path'// Serve the static assets
fastify.register(require('fastify-static'), {
  root: path.join(__dirname, ''),
  prefix: '/'
})

const start = async () => {
  try {
    await fastify.listen(8080, "0.0.0.0")

    console.log("server listening on:", fastify.server.address().port
    )
} catch (error) {
  fastify.log.error(error)
  }
}

start()
```

The HTML portion of the project is where the magic happens. We will discuss the script inside the HTML document that will actually run the code.

The code follows these steps:

1. The `Go()` function provides the Go WASM runtime brought by `wasm_exec.js` (it handles all the binding between the wasm file and the JavaScript API)
2. `instantiateStreaming` allows to load the WebAssembly module from a streamed source.
    1. This function takes two parameters: the “streamed” wasm file and an `importObject` (an object containing the values to be imported into the newly-created WebAssembly Instance)
        1. The first parameter: `fetch(main.wasm)` directly streams the `main.wasm` module
        2. The Go Wasm runtime provides a ready to use importObject that we call with `go.importObject;`
3. If the WebAssembly module loads successfully, we run the instance using `go.run(result.instance)`

4. If the module fails to load we log the error to console

```
const go = new Go() // 1

WebAssembly.instantiateStreaming (fetch("main.wasm"), go.importO"main.wasm
  console.log("main.wasm is loaded")
  go.run(result.instance) // 3"main.wasm is loaded"// 3onsole.log("ouch",
})
"ouch"// 4
})
```

Interacting with the browser DOM

One of the newer things I've learned about working with Go and WebAssembly is the ability to interact with the browser DOM.

We do this by using the syscall/js package

```
package main
  import (
    "syscall/js"
  )

  func main() {
    message := "Hello World from Go"
    document := js.Global().Get("document")
    h2 := document.Call("createElement", "h2")
    h2.Set("innerHTML", message)
    document.Get("body").Call("appendChild", h2)
}
```

All the code from the previous section remains the same.

Having Go and Javascript talk to each other

Using the syscall/js package we can leverage the best language for a given task and then have them communicate with each other.

Calling a Go function from Javascript

Import the `syscall/js` package (this package allows the WebAssembly to access the host environment (the browser)).

The Hello function takes two parameters (this and args) and returns an `interface{}` type.

We retrieve the first argument passed to the Hello function from JavaScript.

To export the Hello function to the global JavaScript context, we use the `FuncOf` Go method (`FuncOf` is used to create a Func type).

```
package main

import (
  "syscall/js"
)
func Hello(this js.Value, args []js.Value) interface{} {
  message := args[0].String()
  return "Hello " + message
}

func main() {
  js.Global().Set("Hello", js.FuncOf(Hello))
}
```

We also need to update the script inside `index.html` to call the `Hello` function.

```
function loadWasm(path) {
```

```
  const go = new Go()
  return new Promise((resolve, reject) => {
    WebAssembly.instantiateStreaming(
        fetch(path),
        go.importObject
      )
      .then(result => {
        go.run(result.instance)
        resolve(result.instance)
      })
      .catch(error => {
        reject(error)
      })
  })
}

loadWasm("main.wasm").then(wasm => {
  let message = Hello("Bob Morane")
  document.querySelector("h1").innerHTML = message
}).catch(error => {
  console.log(error)
})
```

Once the wasm file is loaded and executed, we can call the Hello function.

Change the value of the <h1></h1> tag with the result of the Hello function.

Calling a Javascript function from Go

The WebAssembly Go runtime converts the Go types to JavaScript types:

Go	JavaScript
js.Value	[its value]
js.Func	function
nil	null
bool	boolean

Go	JavaScript
integers and floats	number
string	string
[]interface{}	new array
map[string]interface{}	new object

The `syscall/js` package provides the functions to access the wasm environment and to the host. Thanks to this package, we can access the functions and variables of the JavaScript host (your browser).

This example shows different things that you can do with the `syscall/js` package.

1. Call the sayHello JavaScript function with a string as parameter (Bill)
2. Get the value of the message JavaScript variable
3. Change the value of the message JavaScript variable
4. Get the bill JavaScript object
5. Add 2 fields with values to the bill JavaScript object
6. `<-make(chan bool)` tells the Go application that we don't want to exit by using a channel
    1. The channel waits for data and will pause the execution until it receives data.

```
package main
import (
  "fmt"
  "syscall/js"
)

func main() {"syscall/js"// 1intln(js.Global().Call("sayHello", "Bill"))

  // 2
  me"sayHello"// 2obal().Get("message").String()

  fmt.Println("message (before):", message)

  // 3
```

```
  js.Global()."message"// 3, "Hello from Go")

  //4
  bill := js.Global().G"Hello from Go"//4 rintln("bill (before):", bill)

  // 5
  bill.Set("firstName", "Bill")
  bil"bill (before):"// 5
  bill.Set("firstName", "Bill")
  bill.Set("lastName", "Ballantine")

  // 6
  <-make(chan bool)
}
```

We also need to change the script in `index.html`.

We create Javascript functions and constants that will be used in conjunction with the Go code.

The `loadWasm` function is used to load the Wasm file without having to write it every time.

We then load the Wasm file, execute the code and call the functioons defined in Go.

```
const message = "this is a message"

function sayHello(name) {
  return `Hello ${name}`
}

const bill = {
  age: 42
}

function loadWasm(path) {
  const go = new Go()
```

```
  return new Promise((resolve, reject) => {
    WebAssembly.instantiateStreaming(
        fetch(path),
        go.importObject
      )
      .then(result => {
        go.run(result.instance)
        resolve(result.instance)
      })
      .catch(error => {
        reject(error)
      })
  })
}

// Load the wasm file
loadWasm("main.wasm").then(wasm => {
  console.log("message (after):", message)
  console.log("bill (after):", bill)
}).catch(error => {
  console.log("ouch", error)
})
"main.wasm"
```

So we can create Go code that provides functionality to Javascript applications and consumes data from Javascript.

> So what's next?
>
> There are several things that we can explore from here. Some of them include

- Reduce the size of the Wasm binaries by using TinyGo and understanding the tradeofs and limitations of the TinyGo subset of the language
- Using different abstraction layers like Wasmtime, Wasmedge and Wasmer, and WASI to create standalone applications that run WebAssembly
    - Take advantage of tools like subo, part of the Suborbital family of tools