Go Wiki: WebAssembly

Table of Contents

- Introduction
- JavaScript (GOOS=js) port
  - Getting Started
  - Executing WebAssembly with Node.js
  - Running tests in the browser
  - Interacting with the DOM
  - Configuring fetch options while using net/http
  - WebAssembly in Chrome
  - Further examples
- WASI (GOOS=wasip1) port
  - Getting Started (WASI)
- Go WebAssembly talks
- Editor configuration
- Debugging
  - Analysing the structure of a WebAssembly file
- Reducing the size of Wasm files
- Other WebAssembly resources

Introduction

Go 1.11 added an experimental port to WebAssembly. Go 1.12 has improved some parts of it, with further improvements expected in Go 1.13. Go 1.21 added a new port targeting the WASI syscall API.

WebAssembly is described on its [home page](https://webassembly.org/) as:

> WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

If you're new to WebAssembly read the Getting Started section, watch some of the Go WebAssembly talks, then take a look at the Further examples below.

JavaScript (GOOS=js) port

Getting Started

This page assumes a functional Go 1.11 or newer installation. For troubleshooting, see the Install Troubleshooting page.

> If you are on Windows, we suggest to follow this tutorial using a BASH emulation system such as Git Bash.
>
> For Go 1.23 and earlier, the wasm support files needed in this article are located in `misc/wasm`, and the path should be replaced when performing operations with files such as `lib/wasm/wasm_exec.js`.

To compile a basic Go package for the web:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, WebAssembly!")
}
```

Set GOOS=js and GOARCH=wasm environment variables to compile for WebAssembly:

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

That will build the package and produce an executable WebAssembly module file named main.wasm. The .wasm file extension will make it easier to serve it over HTTP with the correct Content-Type header later on.

Note that you can only compile main packages. Otherwise, you will get an object file that cannot be run in WebAssembly. If you have a package that you want to be able to use with WebAssembly, convert it to a main package and build a binary.

To execute main.wasm in a browser, we'll also need a JavaScript support file, and a HTML page to connect everything together.

Copy the JavaScript support file:

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

Create an `index.html` file:

```
<html>
    <head>
        <meta charset="utf-8"/>
        <script src="wasm_exec.js"></script>
        <script>
            const go = new Go();
            WebAssembly.instantiateStreaming(fetch("main.wasm"), go.importObject).then((result) => {
                go.run(result.instance);
            });
        </script>
    </head>
    <body></body>
</html>
```

If your browser doesn't yet support `WebAssembly.instantiateStreaming`, you can use a polyfill.

Then serve the three files (`index.html`, `wasm_exec.js`, and `main.wasm`) from a web server. For example, with `goexec`:

```
# install goexec: go install github.com/shurcooL/goexec@latest
goexec 'http.ListenAndServe(`:8080`, http.FileServer(http.Dir(`.`)))'
```

Or use your own basic HTTP server command.

Note: The same major Go version of the compiler and `wasm_exec.js` support file must be used together. That is, if `main.wasm` file is compiled using Go version 1.N, the corresponding `wasm_exec.js` file must also be copied from Go version 1.N. Other combinations are not supported.

Note: for the `goexec` command to work on Unix-like systems, you must add the path environment variable for Go to your shell's `profile`. Go's getting started guide explains this:

> Add `/usr/local/go/bin` to the PATH environment variable. You can do this by adding this line to your `/etc/profile` (for a system-wide installation) or `$HOME/.profile`:
>
> ```
> export PATH=$PATH:/usr/local/go/bin
> ```
>
> Note: changes made to a profile file may not apply until the next time you log into your computer

Finally, navigate to http://localhost:8080/index.html, open the JavaScript debug console, and you should see the output. You can modify the program, rebuild `main.wasm`, and refresh to see new output.

Executing WebAssembly with Node.js

It's possible to execute compiled WebAssembly modules using Node.js rather than a browser, which can be useful for testing and automation.

First, make sure Node is installed and in your PATH.

Then, add `$(go env GOROOT)/lib/wasm` to your PATH. This will allow `go run` and `go test` find `go_js_wasm_exec` in a PATH search and use it to just work for `js/wasm`:

```
$ export PATH="$PATH:$(go env GOROOT)/lib/wasm"
$ GOOS=js GOARCH=wasm go run .
Hello, WebAssembly!
$ GOOS=js GOARCH=wasm go test
PASS
ok      example.org/my/pkg  0.008s
```

If you're running working on Go itself, this will also allow you to run `run.bash` seamlessly.

`go_js_wasm_exec` is a wrapper that allows running Go Wasm binaries in Node. By default, it may be found in the `lib/wasm` directory of your Go installation.

If you'd rather not add anything to your PATH, you may also set the `-exec` flag to the location of `go_js_wasm_exec` when you execute `go run` or `go test` manually.

```
$ GOOS=js GOARCH=wasm go run -exec="$(go env GOROOT)/lib/wasm/go_js_wasm_exec" .
Hello, WebAssembly!
$ GOOS=js GOARCH=wasm go test -exec="$(go env GOROOT)/lib/wasm/go_js_wasm_exec"
PASS
ok      example.org/my/pkg  0.008s
```

Finally, the wrapper may also be used to directly execute a Go Wasm binary:

```
$ GOOS=js GOARCH=wasm go build -o mybin .
$ $(go env GOROOT)/lib/wasm/go_js_wasm_exec ./mybin
Hello, WebAssembly!
$ GOOS=js GOARCH=wasm go test -c
$ $(go env GOROOT)/lib/wasm/go_js_wasm_exec ./pkg.test
PASS
ok      example.org/my/pkg  0.008s
```

Running tests in the browser

You can also use wasmbrowsertest to run tests inside your browser. It automates the job of spinning up a webserver and uses headless Chrome to run the tests inside it and relays the logs to your console.

Same as before, just go `get github.com/agnivade/wasmbrowsertest` to get a binary. Rename that to `go_js_wasm_exec` and place it to your PATH!

```
$ mv $GOPATH/bin/wasmbrowsertest $GOPATH/bin/go_js_wasm_exec
$ export PATH="$PATH:$GOPATH/bin"
$ GOOS=js GOARCH=wasm go test
PASS
ok      example.org/my/pkg  0.008s
```

Alternatively, use the `exec` test flag.

```
GOOS=js GOARCH=wasm go test -exec="$GOPATH/bin/wasmbrowsertest"
```

Interacting with the DOM

See https://pkg.go.dev/syscall/js.

Also:

- app: A PWA-compatible, React-based framework with custom tooling.
- dom: A library for streamlining DOM manipulation is in development.
- dom: Go bindings for the JavaScript DOM APIs.
- domjs: A pure Go framework for creating complete GUI application.
- gas: Components based framework for WebAssembly applications.
- GoWebapp: A library to build pages with pure Go and add WebAssembly bindings.
- hogosuru: An advanced webassembly framework that implements most of the features (including indexeddb, serviceworker, websocket and much more) of browsers directly accessible in GO.
- VECTY: Build responsive and dynamic web frontends in Go using WebAssembly, competing with modern web frameworks like React & VueJS.
- vert: WebAssembly interop between Go and JS values.
- vue: The progressive framework for WebAssembly applications.
- Vugu: A wasm web UI library featuring HTML layout with Go for app logic, single-file components, rapid dev and prototyping workflow.
- webapi: A binding generator and generated bindings for DOM, HTML, WebGL, and more.
- wrbgen: Define components in HTML and generate Go types and constructor functions for them using webapi.

Canvas

- A new canvas drawing library - seems pretty efficient.
  - Simple demo

Configuring fetch options while using net/http

You can use the net/http library to make HTTP requests from Go, and they will be converted to fetch calls. However, there isn't a direct mapping between the fetch options and the http client options. To achieve this, we have some special header values that are recognized as fetch options. They are -

- `js.fetch:mode`: An option to the Fetch API mode setting. Valid values are: "cors", "no-cors", "same-origin", navigate". The default is "same-origin".
- `js.fetch:credentials`: An option to the Fetch API credentials setting. Valid values are: "omit", "same-origin", "include". The default is "same-origin".
- `js.fetch:redirect`: An option to the Fetch API redirect setting. Valid values are: "follow", "error", "manual". The default is "follow".

So as an example, if we want to set the mode as "cors" while making a request, it will be something like:

```
req, err := http.NewRequest("GET", "http://localhost:8080", nil)
req.Header.Add("js.fetch:mode", "cors")
if err != nil {
    fmt.Println(err)
    return
}
resp, err := http.DefaultClient.Do(req)
if err != nil {
    fmt.Println(err)
    return
}
defer resp.Body.Close()
// handle the response
```

Please feel free to subscribe to #26769 for more context and possibly newer information.

WebAssembly in Chrome

If you run a newer version of Chrome there is a flag (`chrome://flags/#enable-webassembly-baseline`) to enable Liftoff, their new compiler, which should significantly improve load times. Further info here.

Further examples

General

- Shimmer - Image transformation in wasm using Go. Live DEMO.
- Video filtering - Filters for video from webcam (source code)
- HandyTools - Provide tools like base64 encoding/decoding, convert Unix time, etc (live DEMO)

Canvas (2D)

- GoWasm Experiments - Demonstrates working code for several common call types
  - bouncy
  - rainbow-mouse
  - repulsion
  - bumpy - Uses the 2d canvas, and a 2d physics engine. Click around on the screen to create objects then watch as gravity takes hold!
  - arty
  - hexy (**new**)
- Gomeboycolor-wasm
  - WASM port of an experimental Gameboy Color emulator. The matching blog post contains some interesting technical insights.
- TinyGo canvas
  - This is compiled with TinyGo instead of standard go, resulting in a **19.37kB (compressed)** wasm file.
- Car and Mouse
  - A game where you gain points by leading a small canvas drawn car with your cursor

Database

- TiDB-Wasm - Running TiDB, a golang database in the browser on Wasm.

WebGL canvas (3D)

- Basic triangle (source code) - Creates a basic triangle in WebGL
  - Same thing, ported to TinyGo (source code) - ~14kB compressed (3% of the size of mainline Go version)
- Rotating cube (source code) - Creates a rotating cube in WebGL
  - Same thing, ported to TinyGo (source code) - ~23kB compressed (4% of the size of mainline Go version)
- Splashy (source code) - Click around on the screen to generate paint...

WASI (GOOS=wasip1) port

Getting Started (WASI)

Go 1.21 introduced WASI as a supported platform. To build for WASI, use the `wasip1` port:

```
$ GOOS=wasip1 GOARCH=wasm go build -o main.wasm
```

The official blog has a helpful introduction to using the WASI port: https://go.dev/blog/wasi.

Go WebAssembly talks

- Building a Calculator with Go and WebAssembly (Source code)
- Get Going with WebAssembly
- GoWASM/AssemblyScript - by cha2010 (Chinese)
- Go for frontend

Editor configuration

- Configuring GoLand and IntelliJ Ultimate for WebAssembly - Shows the exact steps needed for getting Wasm working in GoLand and IntelliJ Ultimate

Debugging

WebAssembly doesn't yet have any support for debuggers, so you'll need to use the good 'ol `println()` approach for now to display output on the JavaScript console.

An official WebAssembly Debugging Subgroup has been created to address this, with some initial investigation and proposals under way:

- WebAssembly Debugging Capabilities Living Standard (source code for the doc)
- DWARF for WebAssembly Target (source code for the doc)

Please get involved and help drive this if you're interested in the Debugger side of things. :smile:

Analysing the structure of a WebAssembly file

WebAssembly Code Explorer is useful for visualising the structure of a WebAssembly file.

- Clicking on a hex value to the left will highlight the section it is part of, and the corresponding text representation on the right
- Clicking a line on the right will highlight the hex byte representations for it on the left

Reducing the size of Wasm files

At present, Go generates large Wasm files, with the smallest possible size being around ~2MB. If your Go code imports libraries, this file size can increase dramatically. 10MB+ is common.

There are two main ways (for now) to reduce this file size:

1. Manually compress the .wasm file.
   - Using gz compression reduces the ~2MB (minimum file size) example WASM file down to around 500kB. It may be better to use Zopfli to do the gzip compression, as it gives better results than `gzip --best`, however it does take much longer to run.
   - Using Brotli for compression, the file sizes are markedly better than both Zopfli and `gzip --best`, and compression time is somewhere in between the two, too. This (new) Brotli compressor looks reasonable.

   Examples from @johanbrandhorst

   Example 1

Size	Command	Compression time
16M	(uncompressed size)	N/A
2.4M	`brotli -o test.wasm.br test.wasm`	53.6s
3.3M	`gzip --zopfli test.wasm`	3m 2.6s
3.4M	`gzip --best test.wasm`	2.5s
3.4M	`gzip test.wasm`	0.8s

   Example 2

Size	Command	Compression time
2.3M	(uncompressed size)	N/A
496K	`brotli -o main.wasm.br main.wasm`	55.7s
640K	`gzip --zopfli main.wasm`	16.2s
660K	`gzip --best main.wasm`	0.2s
665K	`gzip main.wasm`	0.2s

   Use something like https://github.com/paatj/gzipped to automatically serve compressed files with correct headers, when available.
2. Use TinyGo to generate the Wasm file instead.

   TinyGo supports a subset of the Go language targeted for embedded devices, and has a WebAssembly output target.

   While it does have limitations (not yet a full Go implementation), it is still fairly capable and the generated Wasm files are... tiny. ~10kB isn't unusual. The "hello world" example is 575 bytes. If you go -s that, it drops down to 408 bytes. :wink:

   This project is also very actively developed, so its capabilities are expanding out quickly. See https://tinygo.org/docs/guides/webassembly/ for more information on using WebAssembly with TinyGo.

Other WebAssembly resources

- Awesome-Wasm - An extensive list of further Wasm resources. Not Go specific.

This content is part of the Go Wiki.