Go Easy On Your Browser

Go Meetup @ Kramp Hub

September 12, 2019

by Jonathan Hall

flimzy@flimzy.com

What I will cover

Some code in this talk, but not much.

- JavaScript alternatives in the browser
- Why I chose Go
- Two approaches: GopherJS & WASM
- What is GopherJS?
- Why to use GopherJS
- Why not to use GopherJS
- Examples
- Some “Gotchas”
- A very “short” demo
- The future of Go in the browser

About Me

- Name: **Jonathan Hall**

Position:

- “Between jobs”
- Future **Backend Go Developer at Gain.Pro**
- Current **Founder/Owner/Developer/Benevolent Dictator for Flashback (language app)**
- Current **Freelance Go Developer**

Previous Professional Experience:

- **Digital ICT Manager at Bugaboo**
- **Backend developer at companies such as Teamwork.com, Booking.com & others**

JavaScript alternatives in the browser

More and more languages now support transpiling either to vanilla JavaScript, or to WASM.

- C/C++
- C#
- Java
- Python
- Rust
- Countless special-purpose languages or JS dialects
- Many more

Why I chose Go

- I wanted to write an offline HTML5 App: FlashbckSRS
- I had a negative experience with (pre-ES6) JS
- Found GWT, but wasn’t excited to learn Java
- I wanted to learn Go
- GopherJS works well, and it’s fast (Demo)
- Turns out, it’s easy to contribute

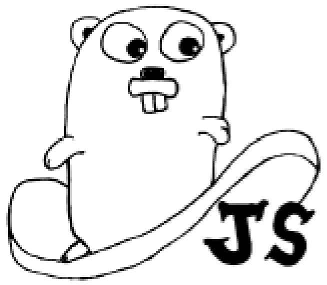
Two approaches: GopherJS & WASM

- GopherJS transpiles Go to JavaScript.
  - Fairly mature
  - Strong community
- Experimental WASM from the official Go compiler since version 1.11 (Aug 2018)
  - Officially supported (though still experimental)
  - ~~Missing some important features~~
  - Still experimental as of Go 1.13. **syscal/js** package subject to rapid change.
  - Has some serious bugs (memory leaks)
  - Will no doubt mature quickly

What is GopherJS?

GopherJS is a ~~compiler~~ transpiler from Go to JavaScript, written by Richard Musiol, who later wrote the WASM port of the Go compiler.

“Build client side apps with the language you already trust on the backend.”



Why GopherJS instead of JavaScript?

- Go libraries in the browser
- JavaScript libraries, too
- Same language on the server and the browser (Where have I heard this before?)
- Good performance
- All the standard reasons Go is so cool:
  - **gofmt**
  - Simple syntax
  - Extensive tooling (though not all supported by GopherJS)
  - Strict types
  - Duck typing
  - field tags in structs
  - interfaces
  - **Go routines**

Why not use GopherJS instead of JavaScript?

- Dynamic types?
- Strong DOM integration
- ~~Callbacks?~~
- Steep learning curve
- Must be familiar with JavaScript internals, too
- Writing wrappers can be time consuming
- Performance can suffer for certain workloads (a hybrid approach can help)
- **Compiled sizes are large (MBs)**

Integrating with existing JS libraries

Two approaches:

- Expose your Go code to JavaScript

  To integrate a specific Go library into a larger JavaScript app.

- Wrap JavaScript libraries in Go

  For projects written primarily in Go which need to communicate with an existing JavaScript library. To manipulate the DOM, access Local Storage, Web Workers, etc.

Calling Go code from JavaScript

A standard Go struct, with getter and setter:

```
type Pet struct {
    name string
}

func (p *Pet) Name() string {
    return p.name
}

func (p *Pet) SetName(name string) {
    p.name = name
}
```

… can be treated as a standard JS object with **MakeWrapper()**:

```
func main() {
    // window.pet = {
    //     New: New
    // };
    js.Global.Set("pet", map[string]interface{}{
        "New": New,
    })
}

func New(name string) *js.Object {
    return js.MakeWrapper(&Pet{name})
}
```

Which can be used from JavaScript:

```
var pet = pet.New('Snoopy');
console.log( pet.Name() ); // Snoopy
pet.SetName('Woodstock');
console.log( pet.Name() ); // Woodstock
```

Calling JavaScript code from Go

js.Object provides the glue between JavaScript objects and Go. It represents a JavaScript object of any type. It is up to you to use it properly.

```
func main() {
    window := js.Global /* `window` in the browser, `GLOBAL` for node.js */
    // var div = window.getElementById("someid");
    div := js.Call("getElementById", "someid")
    // console.log("Inner HTML contains: %s\n", div.innerHTML());
    fmt.Printf("Inner HTML contains: %s\n", div.Get("innerHTML").String() )
}
```

Simple Type Conversions

The **js** package has several easy type conversion methods:

- Bool() bool
- Float() float64
- Int() int
- Int64() int64
- Uint64() uint64
- String() string

Any other needs can use

- Interface() interface{}

to extract the underlying type, using reflection.

Go bindings for JS code

Bindings are easier to use than direct calls to JS libraries.

```
type Widget struct {
    js.Object
    Value string `js:"value"`
}

func New(value string) *Widget { // Constructor wrapper
    // var o = new Object;
    o := js.Global.Get("Object").New()
    // o.value = value;
    w := &Widget{Object: o, Value: value}
    return w
}

func (w *Widget) Frobnicate(args ...interface{}) {
    // w.frobnicate(args1, arg2, arg3, .. argN);
    w.Call("frobnicate", args...) // Async func with variable number/type of arguments
}
```

… with an idiomatic Go layer on top:

```go
type Widget struct {
    o frobbinding.Widget
}

func (w *Widget) Frobnicate() (output string) {
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        // w.frobnicate(function(o) {
        //     output = o;
        // });
        w.o.Frobnicate(func(o string) {
            output = o
            wg.Done()
        })
    }()
    wg.Wait()
}
```

… and avoid variadic functions when possible. Instead create a separate method for each valid variation:

```go
func (w *Widget) FrobnicateWithTransformation(transform func(string) string) (output string) {
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        // w.frobnicate(transform, function(o) {
        //     output = o;
        // }
        w.o.Frobnicate(transform, func(o string) {
            output = o
            wg.Done()
        })
    }()
    wg.Wait()
}
```

But what if I *want* async operation?

Do it the Go way:

```
for _, widget := range widgets {
    go func() {
        output := widget.Frobnicate()
        fmt.Printf("Input = %s, Frobnicated value = %s\n", widget.Value, output)
    }()
}
```

Some Gotchas

- Must start a new goroutine to use blocking code in JS callback.
- Output files can be very large (multiple megabytes).
- There are some corner-case bugs in GopherJS.
- Development has tapered off significantly since Go/WASM was introduced (~12 months)

“Nearly everything is supported, including Goroutines!”

Standard Go libraries are supported except:

- **build**, **importer**, **gosym**, and a few other things used for compling Go (i.e. who cares?)
- **net/*** (partial support), **tls**
- **os/***, syscall (mostly supported, but in node.js only)
- **runtime** (partial support)
- **unsafe**
- Timezones other than UTC and Local

In practice, **runtime** and **unsafe** are all that often matter.

Standard toolkit is supported except:

- Race detector
- Runtime performance and tracing (pprof, etc)

A “short” demo

I wanted some presentation software that:

- Was simpler than Google Slides
- Could understand Markdown and HTML
- Would allow running a GopherJS demo

I wrote this presentation in GopherJS to do this.

I wrote only a single line of JavaScript when building this software:

```
$global.$ = require('jquery');
```

And that was only to demonstrate that using jQuery with GopherJS is trivial.

It sports the following features:

- Asynchronous loading of slides
- Converts from MD to HTML in the browser
- Web worker to do syntax highlighting of code
- All event handlers written in pure Go

```go
import (
    "bytes"
    "fmt"
    "net/http"                          // For HTTP requests
    "net/url"
    "strconv"
    "strings"

    "github.com/russross/blackfriday" // Markdown parser
    "golang.org/x/net/html"           // DOM parser

    "github.com/flimzy/web/worker"    // Web worker Bindings
    "github.com/gopherjs/gopherjs/js" // Bare JS Bindings
    "github.com/gopherjs/jquery"      // jQuery Bindings
)
```

Full source at http://gitlab.com/flimzy/mdslides

Compiled File Size of Empty program, GopherJS 1.12

Compile options	File size
Compiled, no optimizations	69.4kb
Compiled with -m option	42.9kb
Compiled with -m, + uglifyjs -c -m	25.4kb

Compiled File Size of Empty program, GOARCH=wasm

Compile options	File size
Go 1.12	1.3M
Go 1.13	1.1M

Compiled File Size of Demo, GopherJS 1.12

Compile options	File size
Compiled, no optimizations	8.7M
Compiled with -m option	5.6M

Compile options	File size
Compiled with -m, + uglifyjs -c -m	4.5M

The future of Go in the Browser

1. Go 1.13 was just released:

    - Output size improvements
    - Fixes for typed array conversions
    - Various bug fixes

2. Go 1.14 should improve:

    - Performance
    - Improvements to **net** and **net/http**
    - What else? (28 open Go/WASM bugs

3. Meanwhile, GopherJS continues to track the official Go releases

4. Full GopherJS / Go/WASM isomorphism is possible, changes to **syscall/js** notwithstanding.

Questions?

Thank you

Contact me

- Email: **flimzy@flimzy.com**
- GitHub: http://github.com/flimzy
- GitLab: http://gitlab.com/flimzy
- Blog: http://verbally.flimzy.com/

My language app that started it all (WIP)

- Web site: https://flashbacksrs.com/
- App demo: https://app.flashbacksrs.com/

For further reading

- Go: http://golang.org/
- Official GopherJS web page: http://www.gopherjs.org/
- GitHub repo: http://github.com/gopherjs
- This slide show presentation tool: http://gitlab.com/flimzy/mdslides
- The slides: https://gitlab.com/flimzy/gopherjstalk/tree/12-09-2019-gomeetup