Isomorphic Go

Full-stack Go development with GopherJS

GOPHERCON INDIA 2017

wirecog®

Kamesh Balasubramanian

Introduction

- Gopher for 4 years
- Making Web Apps since 1996 (the Perl/CGI days)
- Invented Wireframe Cognition
- Wirecog Founder
- Volunteer Engineering Mentor at CSUN

Reasons to use Go on the Front-end

- Avoid mental context shifts between back-end and front-end coding
- Re-use code between the back-end and the front-end
- Type checking (find mistakes at compile time)
- Avoid data exchange overhead/complexities between client and server
- Avoid callback hell

GopherJS

- A transpiler that converts Go code into JavaScript code
- Bindings are available to JavaScript APIs: DOM, XHR, WebSocket, WebGL
- Bindings are available to various JavaScript libraries: Angular JS, D3, jQuery, VueJS

What's Supported?

Refer to the GopherJS compatibility table to see what packages are fully supported:

https://github.com/gopherjs/gopherjs/blob/master/doc/packages.md

Getting Started

http://www.gopherjs.org

go get -u github.com/gopherjs/gopherjs

The gopherjs command

```
# Builds the JS source file to be included in your web page
$ gopherjs build

# Build and Minify the JS source file
$ gopherjs build -m

# Compiles Go packages and starts a HTTP server
$ gopherjs serve
```

Performance Tips

- Use the -m command line flag to generate minified code.
- Enable gzip compression on your web server
- Use int instead of (u)int8/16/32/64.
- Use float64 instead of float32.

Basic DOM Operations - Alert Message

JavaScript:

```
alert("Hello GopherJS!");
```

GopherJS:

```
js.Global.Call("alert", "Hello GopherJS!")
```

DOM Binding:

```
dom.GetWindow().Alert("Hello GopherJS!")
```

Basic DOM Operations - getElementById()

JavaScript:

```
element = document.getElementById("navContainer");
```

GopherJS:

```
element := js.Global.Get("document").Call("getElementById",
"navContainer")
```

DOM Binding:

```
element :=
dom.GetWindow().Document().GetElementByID("navContainer")
```

Basic DOM Operations - Query Selector

JavaScript:

```
element = document.querySelector("#thumbnails .active");
```

GopherJS:

```
element = js.Global.Get("document").Call("querySelector",
"#thumbnails .active")
```

DOM Binding:

```
dom.GetWindow().Document().QuerySelector("#thumbnails .active")
```

Tip: Alias Calls to js.Global and dom

```
var JS = js.Global


var D = dom.GetWindow().Document()
```

Basic DOM Operations - Changing CSS Style

JavaScript:

```
element = document.GetElementById("modalLayer");
element.style.display = "none";
```

GopherJS:

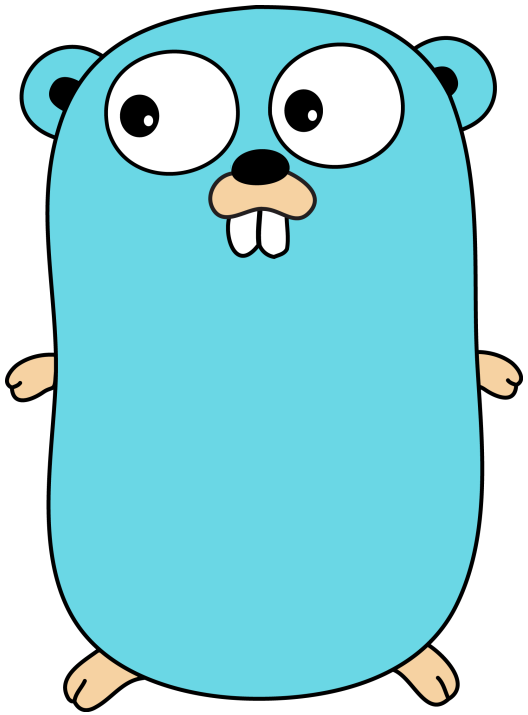
```
var JS = js.Global
element := JS.Get("document").Call("getElementById",
"modalLayer")
element.Get("style").Set("display", "none")
```

DOM Binding:

```
var D = dom.GetWindow().Document()
element := D.GetElementByID("modalLayer")
element.Style().SetProperty("display", "none", "")
```

Let’s make a “Hello World” program using GopherJS:

A 3D version of the Golang Gopher in the web browser



Golang Gopher created by Renée French

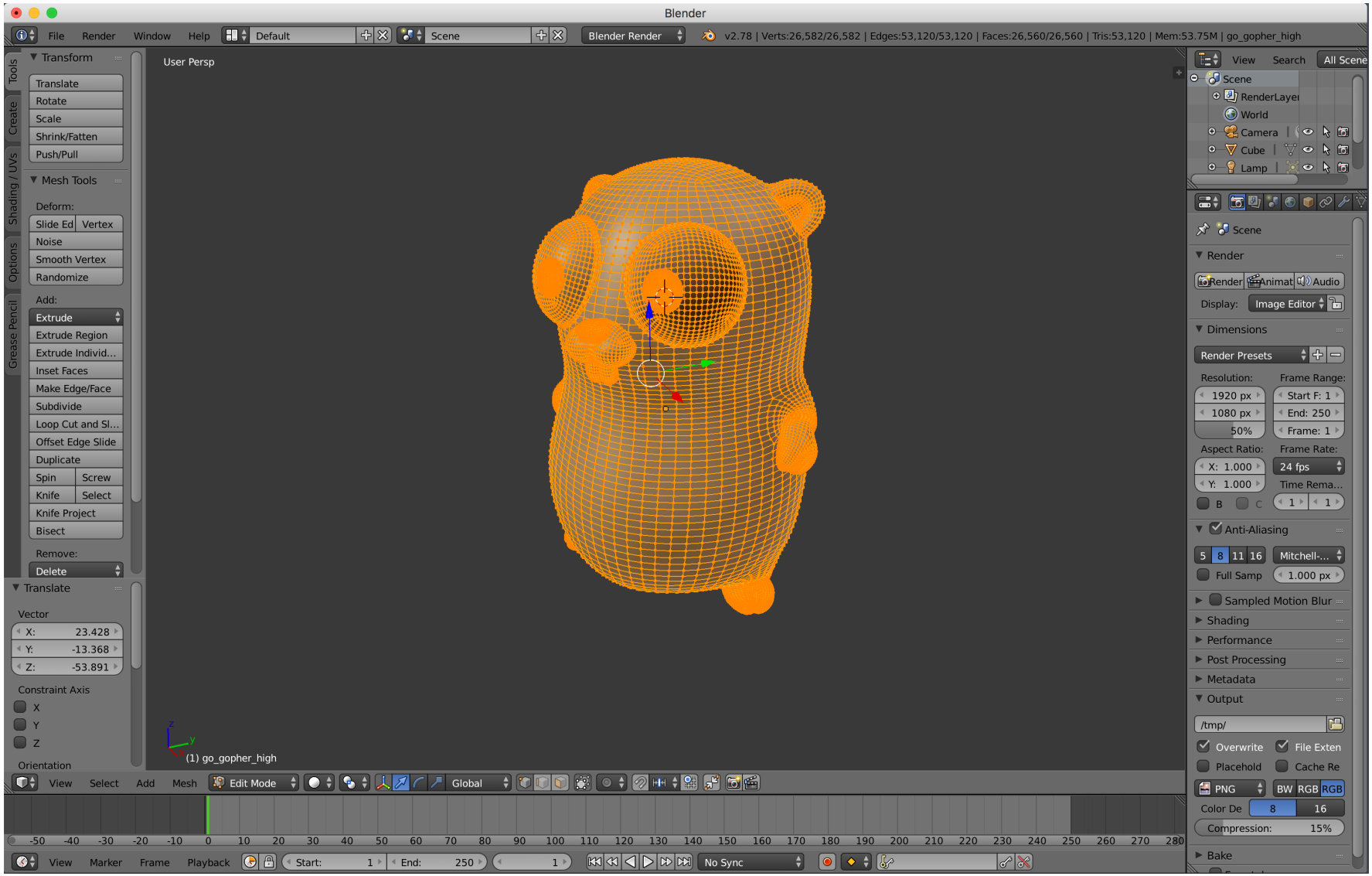
Ingredients for a 3D Gopher in the Web Browser

1. **Three.js** - JavaScript library for displaying 3D graphics in the web browser

2. **Golang Gopher** - 3D Model in Wavefront OBJ Format

3. **GopherJS** - Allows us to make magic in the web browser using Go

3D Gopher Model



Gopher Model courtesy of Stickman Ventures

Inspired by the Golang Gopher created by Renée French

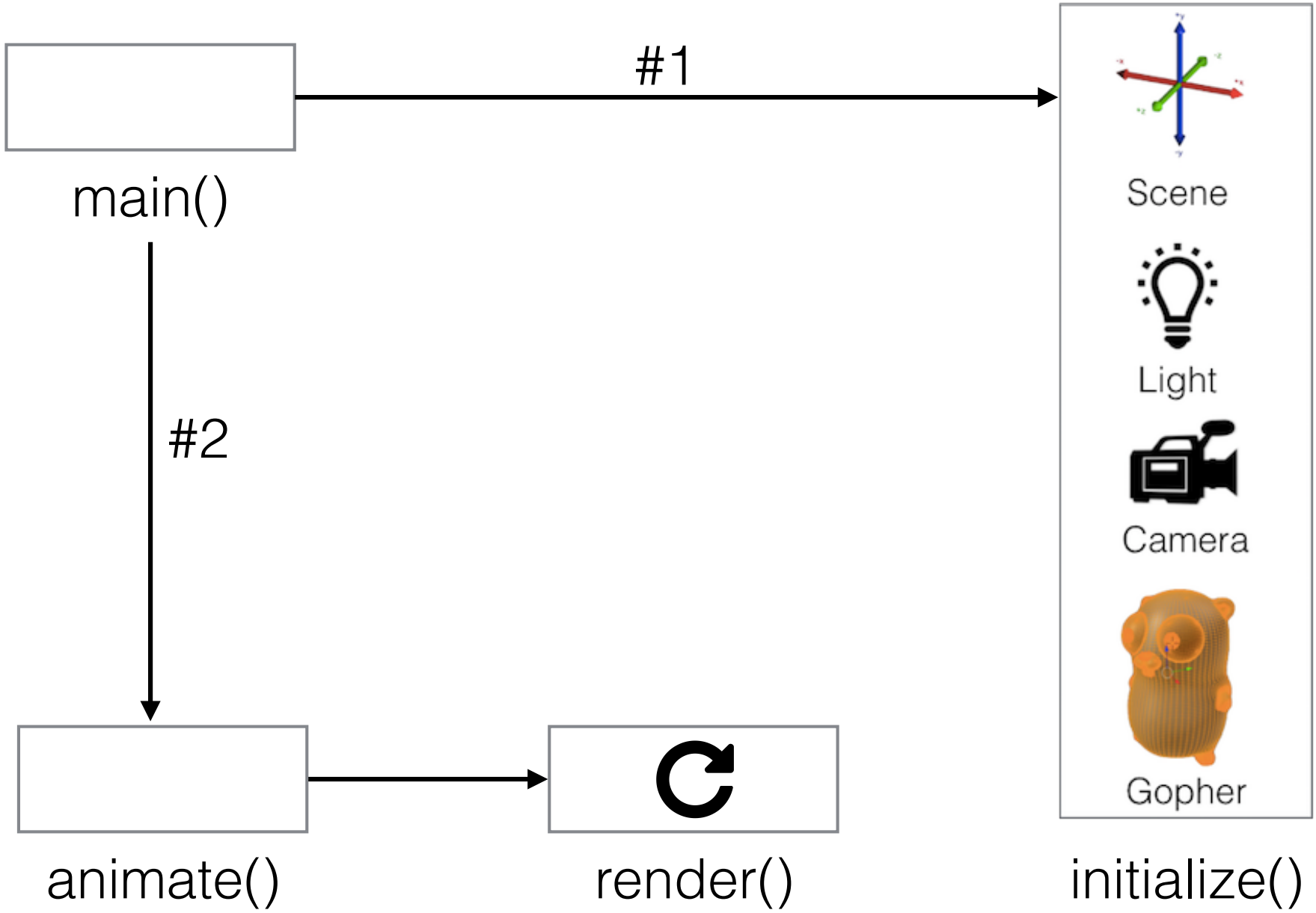
Challenge

- There are no functional GopherJS bindings for the Three.JS Library
- **Solution**: GopherJS is interoperable with JavaScript (use js.Global)

HTML Code Structure

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <title>Hello 3D Gopher!</title>
                                          …

   </head>
  <body>
      <script src="js/three.js"></script>
      <script src="js/OBJLoader.js"></script>

                                          …

      <script src="js/gopher3d.js"></script>
   </body>
</html>
```

3D Gopher Program Flow



main() - Pretty Simple

```
func main() {

   println("Hello 3D Gopher!”)
   initialize()
   animate()

}
```

Declaring JavaScript Objects in Go

```
// Div container that holds the viewport
var container *dom.HTMLDivElement

// Three.js Camera Object
var camera *js.Object

// Three.js Vector3 Object
var cameraTarget *js.Object

// Three.js Scene Object
var scene *js.Object

// Three.js Renderer Object
var renderer *js.Object
```

initialize() - Setting up the Viewport Container

```
container = D.CreateElement("div").(*dom.HTMLDivElement)

D.QuerySelector("body").AppendChild(container)
```

initialize() - Setting up the Camera and the Scene

```
        camera = JS.Get("THREE").Get("PerspectiveCamera").New(35,
JS.Get("window").Get("innerWidth").Int() /
JS.Get("window").Get("innerHeight")
.Int(), 1, 15)
        camera.Get("position").Call("set", 3, 0.15, 3);

        cameraTarget = JS.Get("THREE").Get("Vector3").New(0, 0, 0)

        scene = JS.Get("THREE").Get("Scene").New()

        fog := JS.Get("THREE").Get("Fog").New( 0x708090, 2, 15 )
        scene.Set("fog", fog)
```

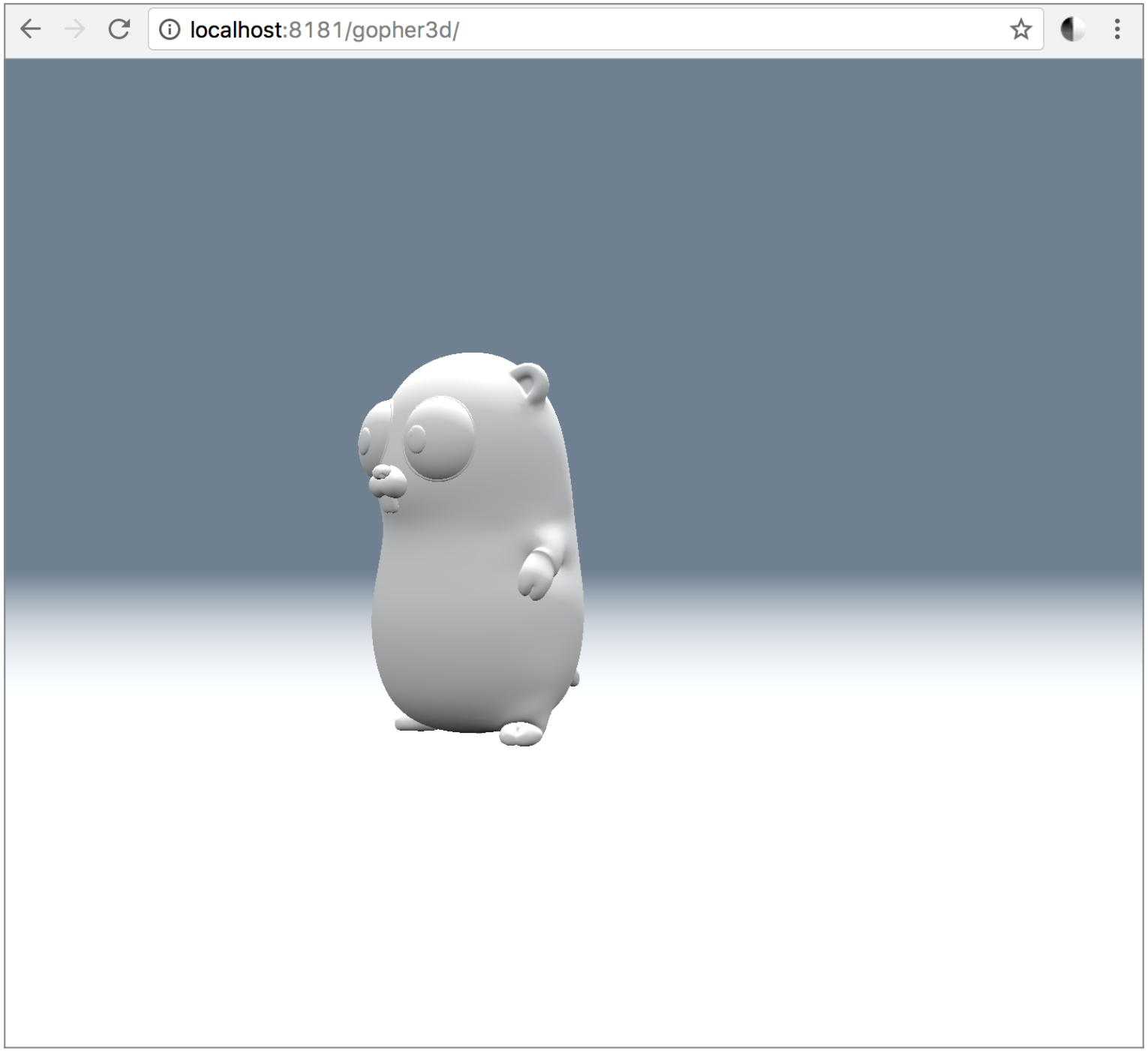
initialize() - Setting up the 3D model

```
        loader := JS.Get("THREE").Get("OBJLoader").New()
        loader.Call("load", "obj/gogopher.obj", func (mesh
*js.Object) {

                mesh.Get("position").Call("set", 0, -0.45, 0.3)
                mesh.Get("scale").Call("set", 0.004, 0.0054,
0.004)
                mesh.Set("castsShadow", true)
                mesh.Set("receiveShadow", true)

                scene.Call("add", mesh)

        })
```

animate() and render()

```
func animate() {

        JS.Get("window").Call("requestAnimationFrame", animate)
        render()
}


func render() {

        camera.Call("lookAt", cameraTarget)
        renderer.Call("render", scene, camera)
}
```

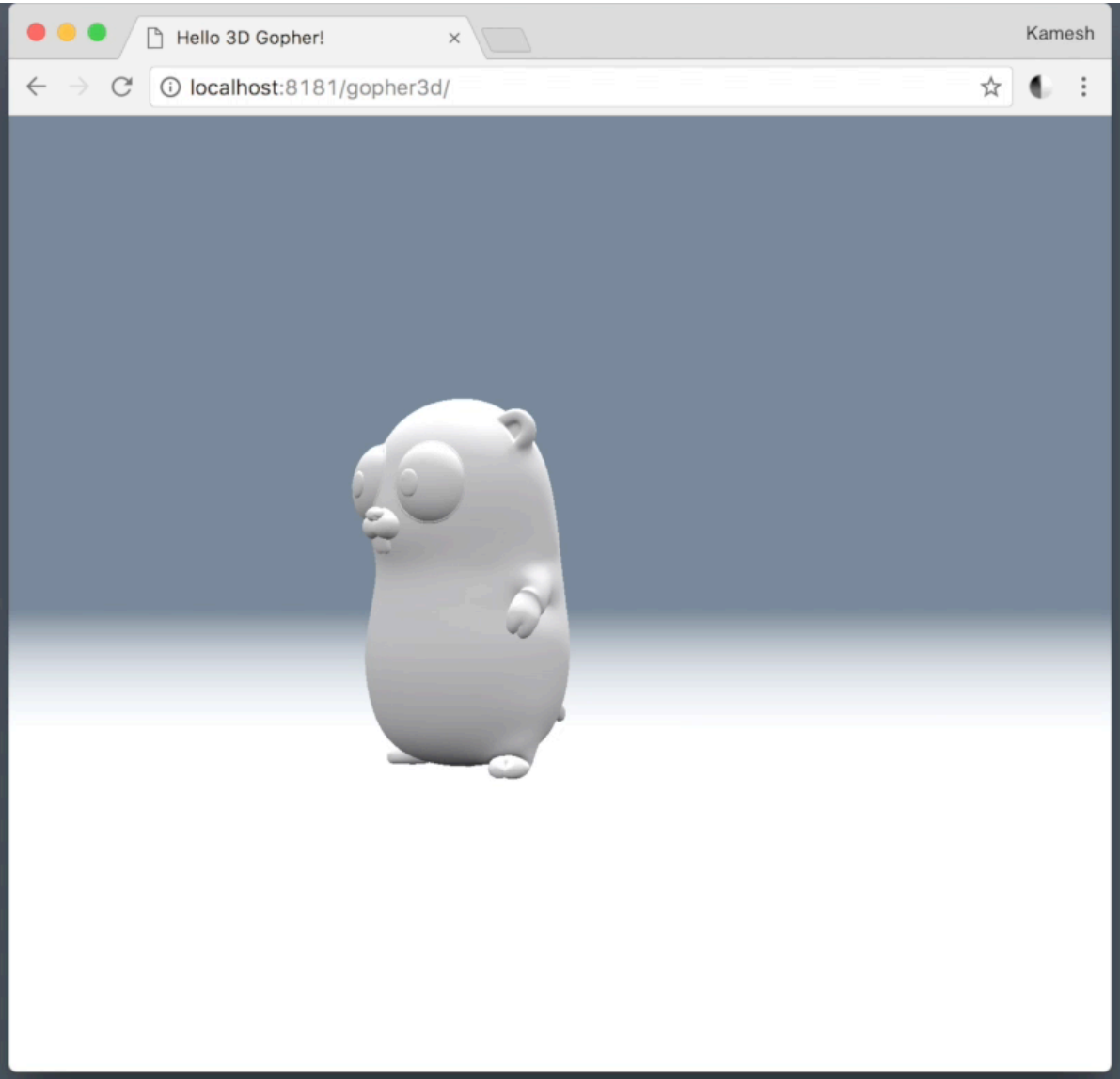
Rendered Gopher



That's a start, but we need to go further

Areas for Improvement

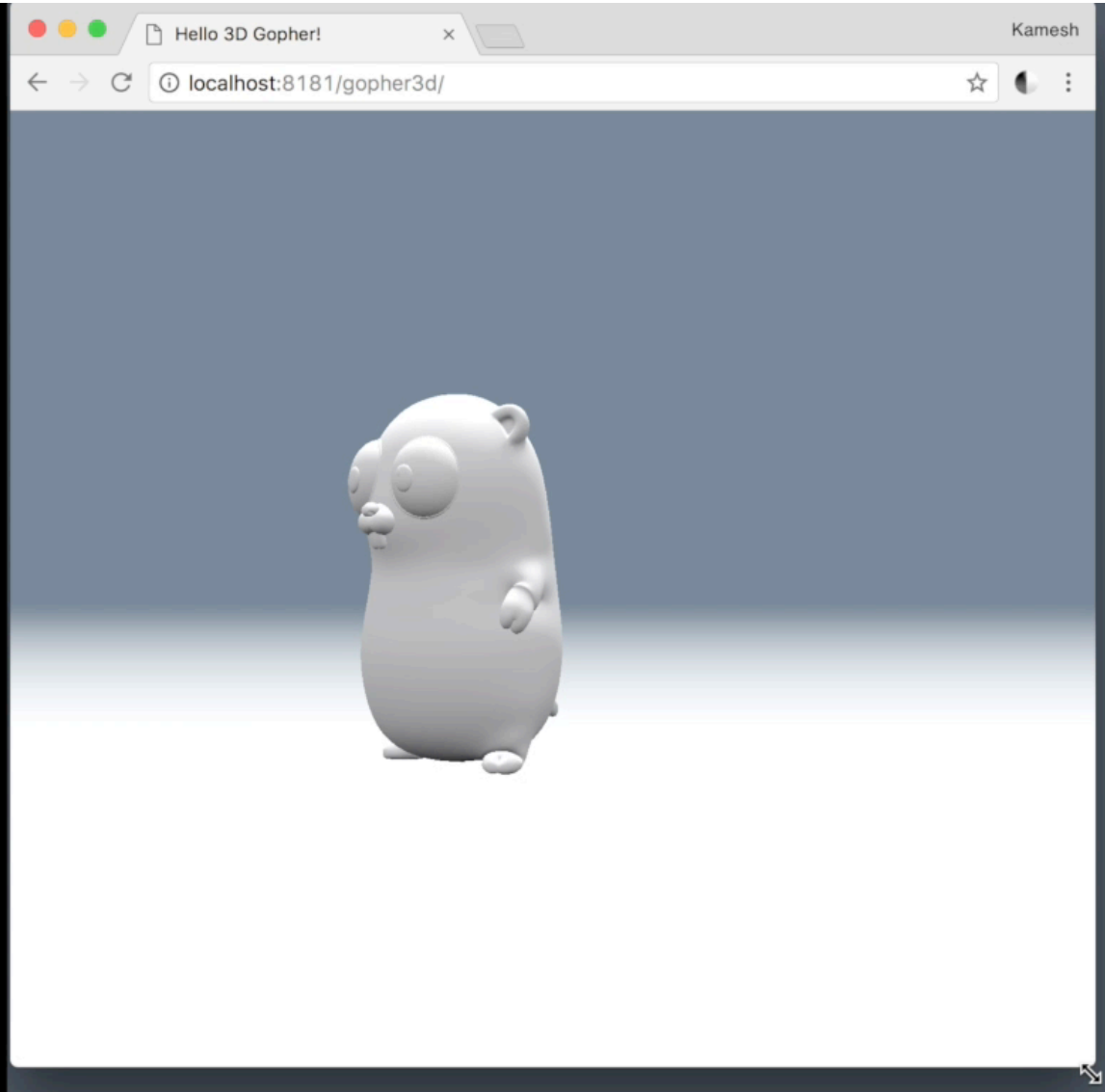
- If we resize the browser window, the viewport doesn't get redrawn
- Let's add color to the 3D Model (preferably "Gopher Cyan")
- Let's make the scene more lively by having the camera pan around the Gopher



Without Resize Handling

Event Handler for Window Resize

```
// We'll add this code to main()
dom.GetWindow().AddEventListener("resize", false, func(event
dom.Event) {
    handleWindowResize()
});



// Event handler to handle window resize events
func handleWindowResize() {

    camera.Set("aspect",
    JS.Get("window").Get("innerWidth").Int() /
    JS.Get("window").Get("innerHeight").Int())

    camera.Call("updateProjectionMatrix")

    renderer.Call("setSize",JS.Get("window").Get("innerWidth").I
    nt(), JS.Get("window").Get("innerHeight").Int())

}
```



With Resize Handling

Adding Color - JS Literal Object Representation

```
// The Mesh Phong Material Data as a JavaScript Literal Object
{color: 0x007794, specular: 0x111111, shininess: 200}



// Struct to represent Mesh Phong Material Data
type MeshPhongMaterialData struct {
        *js.Object
        Color int `js:"color"`
        Specular int `js:”specular"`
        Shininess int `js:"shininess"`
}
```

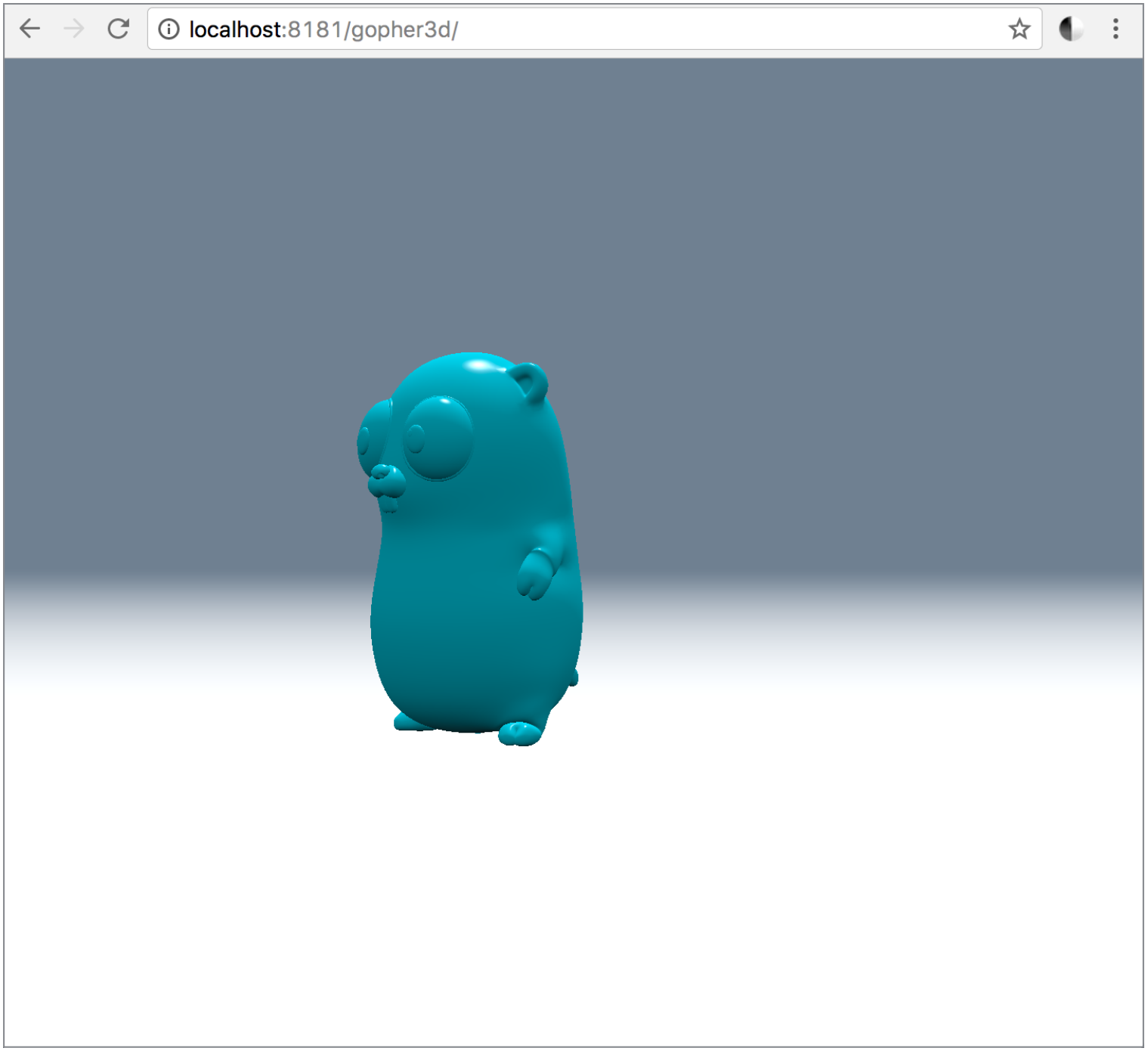
Instantiating the Material Object

```
// With GopherJS, we have to create the object like so:
gopherMeshPhongMaterial := &MeshPhongMaterialData{Object:
js.Global.Get("Object").New()}

gopherMeshPhongMaterial.Color = 0x007794
gopherMeshPhongMaterial.Specular = 0x111111
gopherMeshPhongMaterial.Shininess = 200

material :=
JS.Get("THREE").Get("MeshPhongMaterial").New(gopherMeshPhongMater
ial)
```

Using jsbuiltin Binding for instanceof Operator

```
 // Note: We make use of the jsbuiltin binding to perform the
instanceof check


 mesh.Call("traverse", func (child *js.Object) {
    if jsbuiltin.InstanceOf(child, JS.Get("THREE").Get("Mesh")) {

       child.Set("material", material)
    }
 });
```

After Adding Color

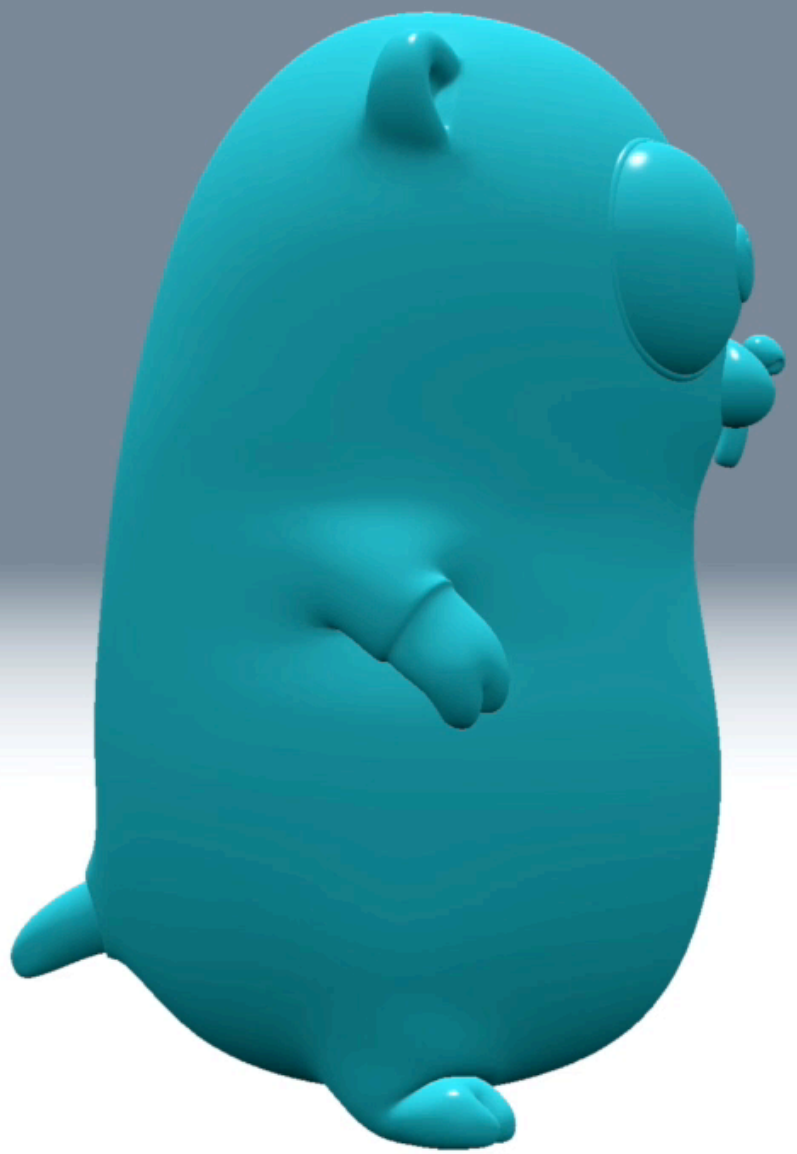


Make the Camera Pan Around the Gopher

```
function panCamera(camera, cameraTarget) {

   var timer = Date.now() * 0.0005;
   camera.position.x = Math.cos(timer) * 2.7 * -1;
   camera.position.z = Math.sin(timer) * 2.7;
   camera.lookAt( cameraTarget );
}
```

Make the Camera Pan Around the Gopher

```
func render() {

   // Interoperability: Calling a JS function with GopherJS
   JS.Get("window").Call("panCamera", camera, cameraTarget)

   renderer.Call("render", scene, camera)

}
```



Websocket Client / Server Example

Gorilla Web Toolkit

http://www.gorillatoolkit.org

Gorilla icon created by Martin Bérubé

Gorilla Websocket Client/Server Example

https://github.com/gorilla/websocket

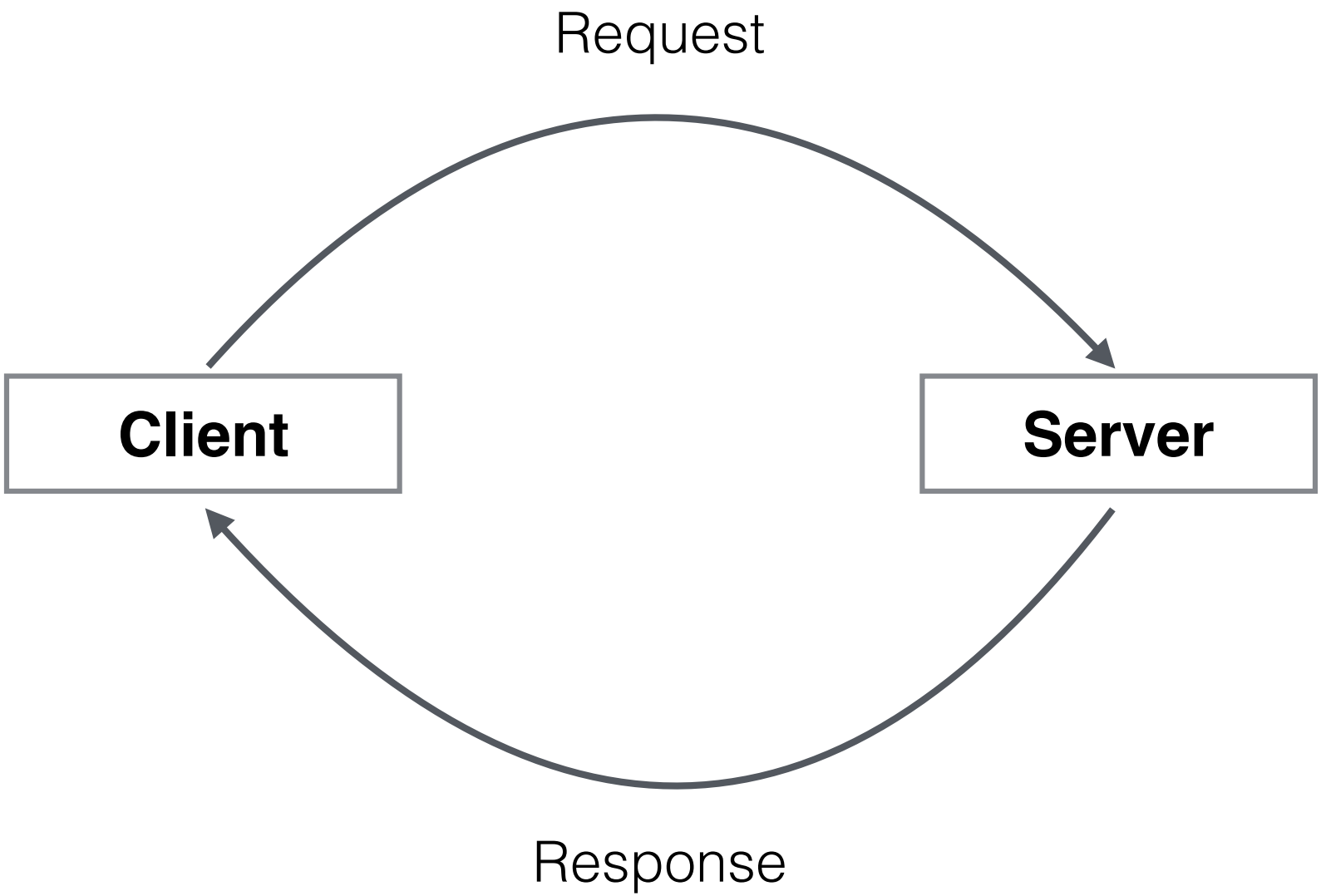
Gorilla WebSocket implements the WebSocket protocol in Go

Let’s examine their websocket client/server example:

https://github.com/gorilla/websocket/tree/master/examples/echo

The server echoes messages sent to it by the client. The client sends a message (in this example it is a timestamp) every second and prints all messages received from the server.

Echo Client/Server



Gorilla Websocket Example Server's echo Function

```
var upgrader = websocket.Upgrader{} // use default options

func echo(w http.ResponseWriter, r *http.Request) {

  c, err := upgrader.Upgrade(w, r, nil)
  if err != nil {
      log.Print("upgrade:", err)
      return
  }

  defer c.Close()

  for {
      mt, message, err := c.ReadMessage()
      if err != nil {
          log.Println("read:", err)
          break
      }

      log.Printf("recv: %s", message)
      err = c.WriteMessage(mt, message)
      if err != nil {
          log.Println("write:", err)
          break
      }
  }
}
```

Gorilla Websocket Example Client (part 1 of 3)

```
func main() {

  flag.Parse()
  log.SetFlags(0)
  interrupt := make(chan os.Signal, 1)
  signal.Notify(interrupt, os.Interrupt)

  u := url.URL{Scheme: "ws", Host: *addr, Path: "/echo"}
  log.Printf("connecting to %s", u.String())

  c, _, err := websocket.DefaultDialer.Dial(u.String(), nil)

  if err != nil {
     log.Fatal("dial:", err)
  }

  defer c.Close()
```

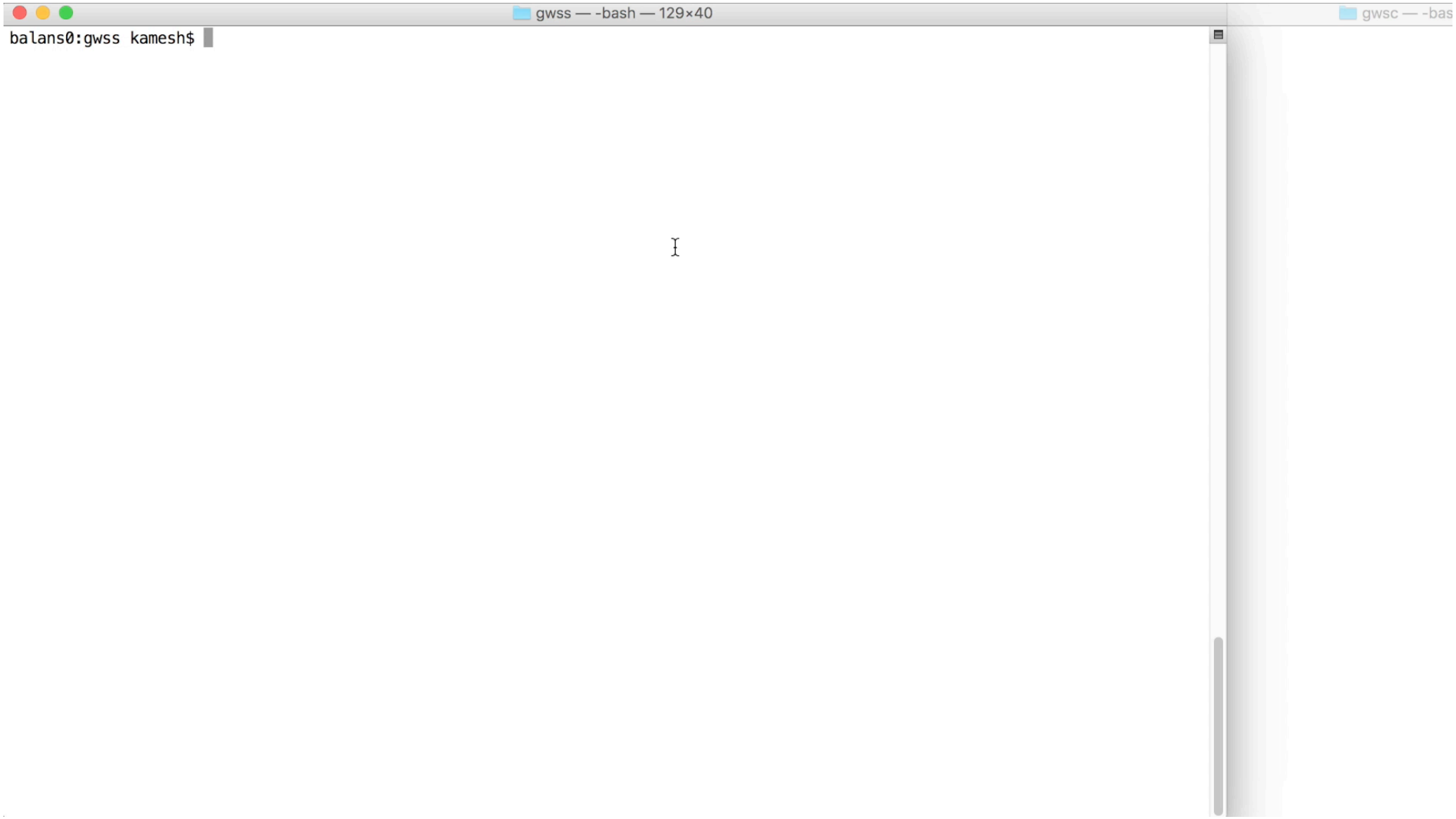
Gorilla Websocket Example Client (part 2 of 3)

```
go func() {

    defer c.Close()
    defer close(done)

    for {
        _, message, err := c.ReadMessage()

        if err != nil {
            log.Println("read:", err)
            return
        }

        log.Printf("recv: %s", message)
    }
}()

ticker := time.NewTicker(time.Second)
defer ticker.Stop()
```

Gorilla Websocket Example Client (part 3 of 3)

```
    for {
         select {

         case t := <-ticker.C:
              err := c.WriteMessage(websocket.TextMessage, []byte(t.String()))
              if err != nil {
                   log.Println("write:", err)
                   return
              }

         case <-interrupt:
              log.Println("interrupt")
              // To cleanly close a connection, a client should send a close
              // frame and wait for the server to close the connection.

              err := c.WriteMessage(websocket.CloseMessage,
websocket.FormatCloseMessage(websocket.CloseNormalClosure, ""))

              if err != nil {
                   log.Println("write close:", err)
                   return
              }

              select {
              case <-done:
              case <-time.After(time.Second):
              }

              c.Close()
              return
         }
    }
}
```

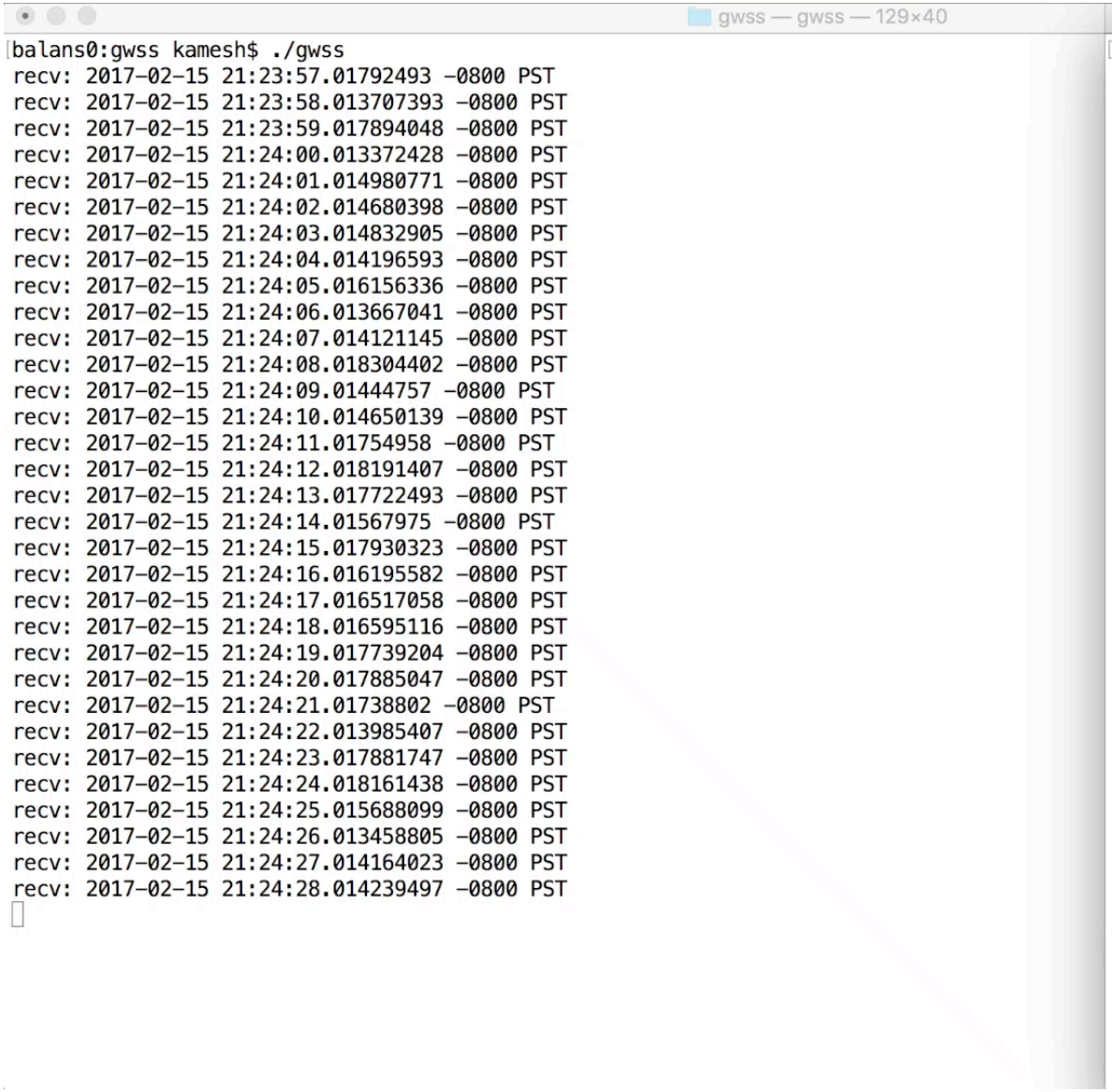
Sample Run

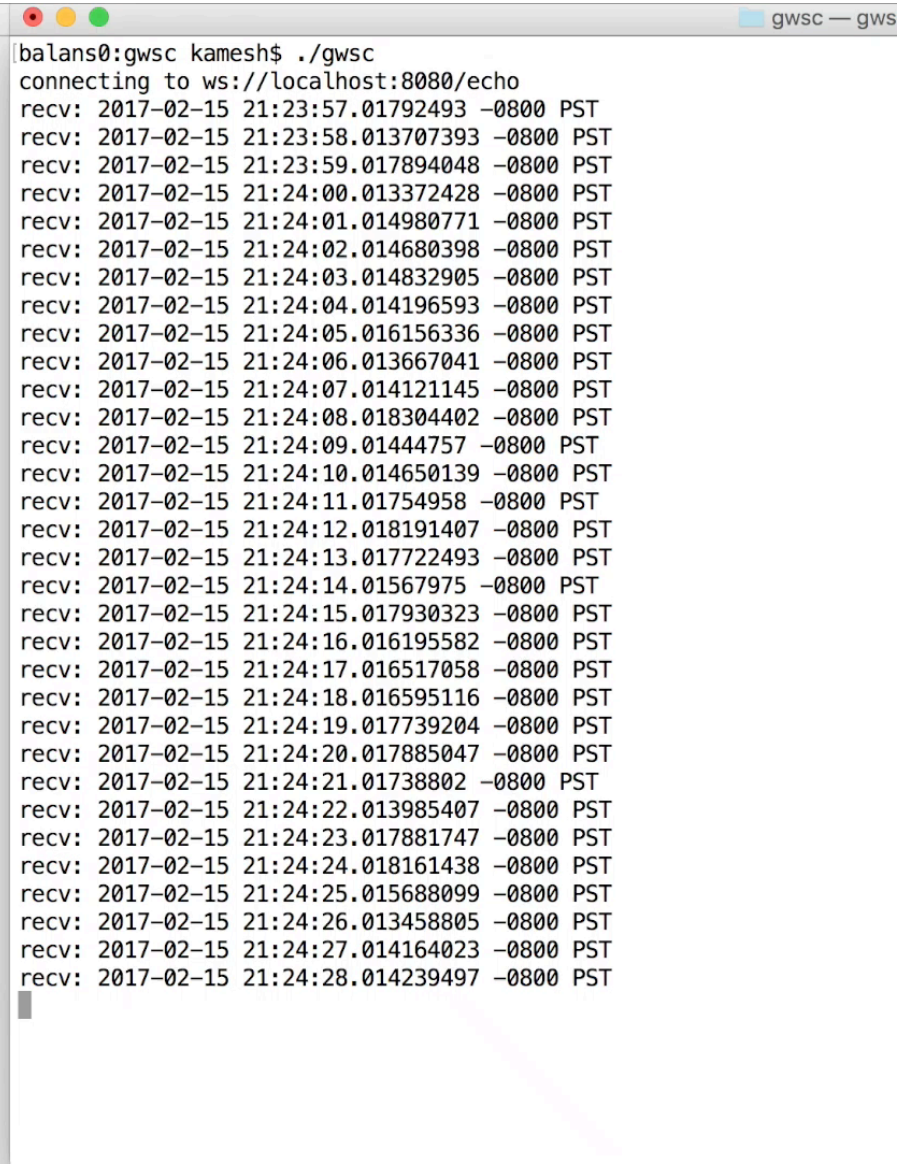


Server

Client

Send Interrupt, Close Connection





Server **Client**

Note: The client responsibly closes the websocket connection.

That's nice, but the example Gorilla websocket client runs on the command line.

With GopherJS we can port this client program to where it naturally belongs…

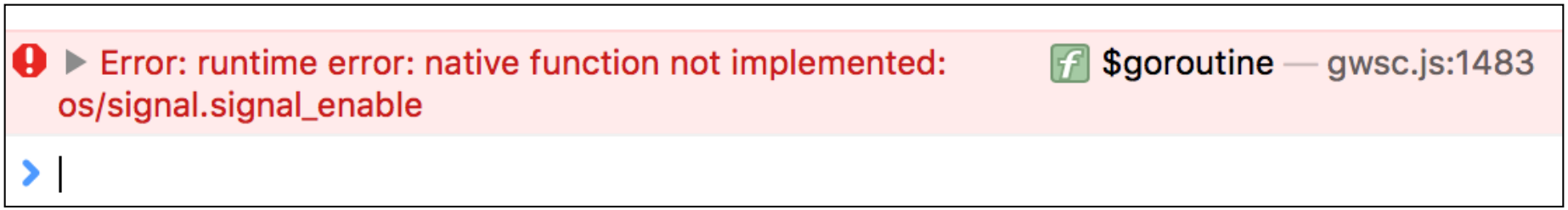
In the web browser.

First Attempt

Let's do a "gopherjs build" of the Gorilla client

```
balans0:gwsc kamesh$ gopherjs build
balans0:gwsc kamesh$ ls -atl
total 21872
-rw-r--r--   1 kamesh  staff  5418196 Feb  7 10:39 gwsc.js
-rw-r--r--   1 kamesh  staff   222936 Feb  7 10:39 gwsc.js.map
drwxr-xr-x   6 kamesh  staff      204 Feb  7 10:39 .
-rw-r--r--   1 kamesh  staff     1645 Feb  7 10:39 main.go
-rwxr-xr-x   1 kamesh  staff  5547132 Feb  7 10:05 gwsc
drwxr-xr-x  10 kamesh  staff      340 Feb  7 09:51 ..
balans0:gwsc kamesh$
```

Initial Result



That didn’t go so well!

Signal is only partially supported

os	☑ partially	node.js only
-- exec	☑ partially	node.js only
-- signal	☑ partially	node.js only
-- user	☑ partially	node.js only

GopherJS Compatibility Table:

https://github.com/gopherjs/gopherjs/blob/master/doc/packages.md

Question: How can we translate the user interrupt paradigm from the command line to the web browser?

Answer: Handle the **onbeforeunload** event which fires when the user closes the browser window or browser tab.

Change the interrupt channel type

```
/*
        interrupt := make(chan os.Signal, 1)
        signal.Notify(interrupt, os.Interrupt)
*/

        interrupt := make(chan bool, 1)
        dom.GetWindow().AddEventListener("beforeunload", false,
func(event dom.Event) {
                interrupt <- true
        });
```

Closing The Connection Responsibly

```
case <-interrupt:

 // To cleanly close a connection, a client should send a close
 // frame and wait for the server to close the connection.

 /* err :=
 c.WriteMessage(websocket.CloseMessage,websocket.FormatCloseMessa
 ge(websocket.CloseNormalClosure, “")) */

 c.Close()
```

There’s a little bit more that we have to do…

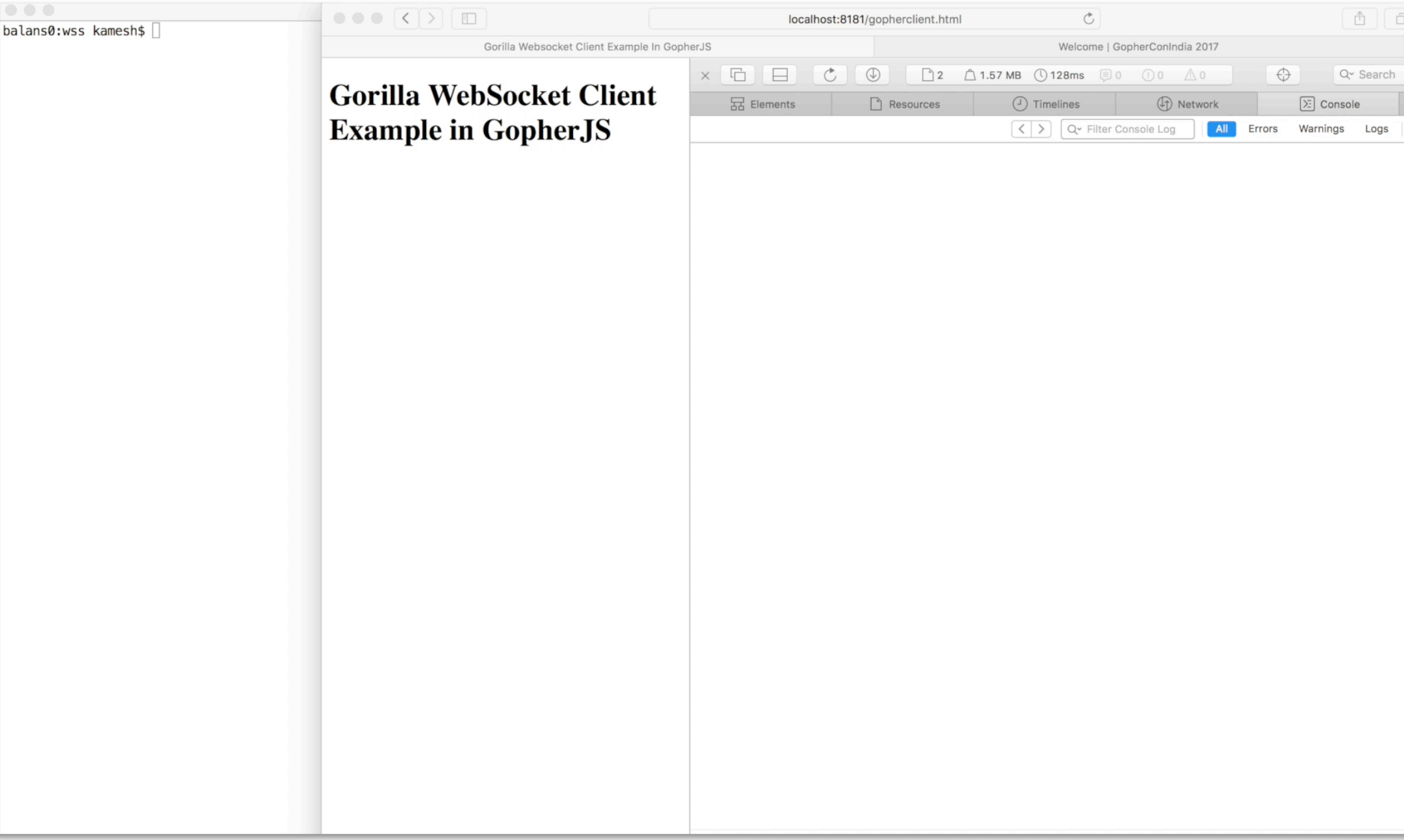
We also need to replace Gorilla’s websocket calls with their equivalent GopherJS websocket binding calls.

GopherJS Websocket Bindings:
https://github.com/gopherjs/websocket

Replacing Gorilla Websocket Calls

```
//c, _, err := websocket.DefaultDialer.Dial(u.String(), nil)
c, err := websocket.Dial(u.String())


//err := c.WriteMessage(websocket.TextMessage,[]byte(t.String()))
_, err = c.Write([]byte(t.String()))



//_, message, err := c.ReadMessage()
message := make([]byte, 1024)
var n int
n, err = c.Read(message)
if err != nil {
  log.Println("read:", err)
  return
}
log.Printf("recv: %s", message[:n])
```

Client Running in the Web Browser



Server

Client

Closing The Websocket connection responsibly

```
recv: 2017-02-07 11:34:37.978 -0800 PST
recv: 2017-02-07 11:34:40.024 -0800 PST
recv: 2017-02-07 11:34:40.029 -0800 PST
read: websocket: close 1006 (abnormal closure): unexpected EOF
```

Without onbeforeunload Handling (Bad)

```
recv: 2017-02-07 11:36:11.62 -0800 PST
recv: 2017-02-07 11:36:12.624 -0800 PST
read: websocket: close 1005 (no status)
```

With onbeforeunload Handling (Good)

Single Page Web App Demo

Client Side Templates

We can render templates directly within the web browser using the html/template package.

Client-side Template - HTML Structure

```
<html>
<head>
<link rel="stylesheet" href="cars.css">
</head>
<body>

<table>
  <thead>
    <tr>
      <th class="">Model Name</th>
      <th class="">Color</th>
      <th class="">Manufacturer</th>
    </tr>
  </thead>
  <tbody id="autoTableBody">

  </tbody>
</table>

<script src="cars.js"></script>

</body>
</html>
```

Isomorphic Car Struct

```
package model

type Car struct {

        ModelName string
        Color string
        Manufacturer string

}
```

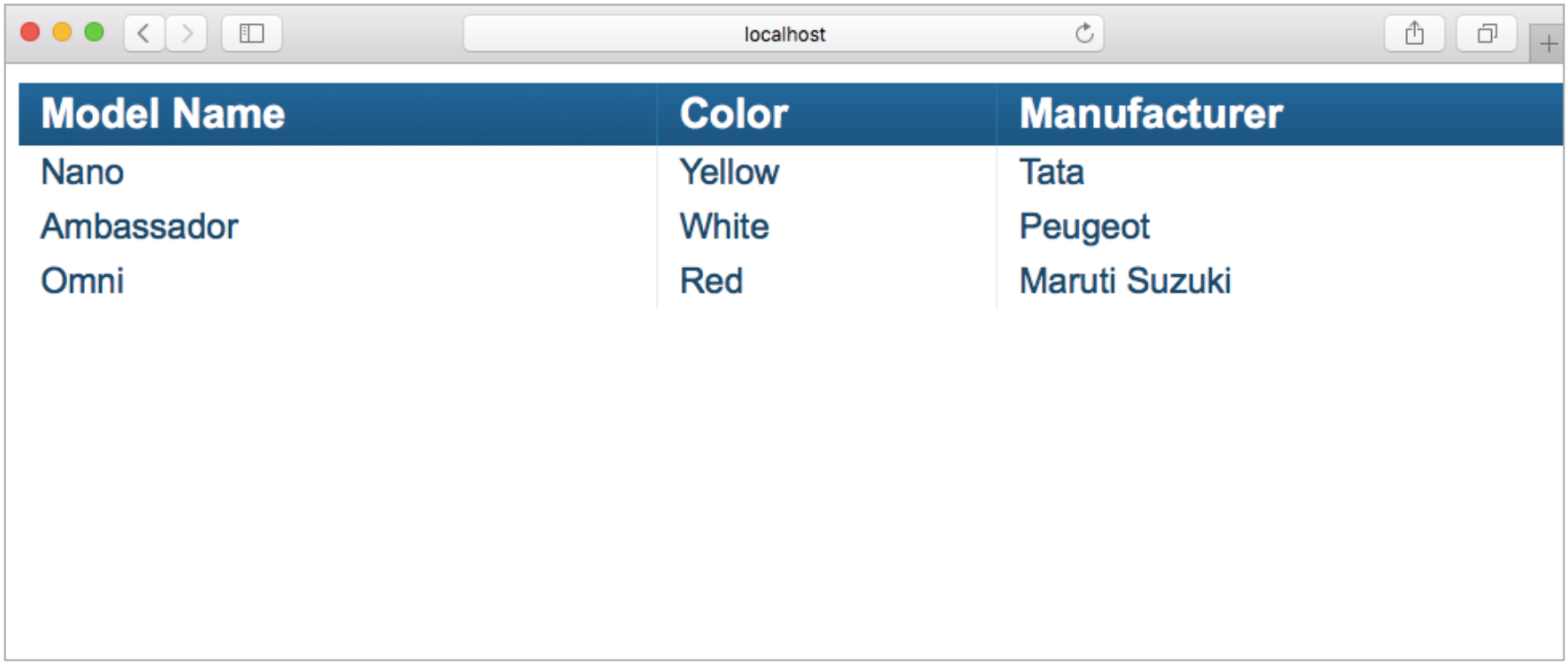
Client-side Template - Go Code

```
package main

import(
  "honnef.co/go/js/dom"
  "html/template"
  "gophercon.in/model"
)

const CarItemTemplate = `
  <td>{{.ModelName}}</td>
  <td>{{.Color}}</td>
  <td>{{.Manufacturer}}</td>
`

var D = dom.GetWindow().Document()
```

Client-side Template - Go Code (continued)

```
func main() {

  nano := model.Car{ModelName:"Nano", Color: "Yellow",
Manufacturer: "Tata"}
  ambassador := model.Car{ModelName:"Ambassador", Color: "White",
Manufacturer: "Peugeot"}
  omni := model.Car{ModelName:"Omni", Color: "Red", Manufacturer:
"Maruti Suzuki"}
  cars := []model.Car{nano, ambassador, omni}

  autoTableBody := D.GetElementByID(“autoTableBody")

  for i := 0; i < len(cars); i++ {
     trElement := D.CreateElement("tr")
     tpl := template.New("template")
     tpl.Parse(CarItemTemplate)
     var buff bytes.Buffer
     tpl.Execute(&buff, cars[i])
     trElement.SetInnerHTML(buff.String())
     autoTableBody.AppendChild(trElement)
  }
```

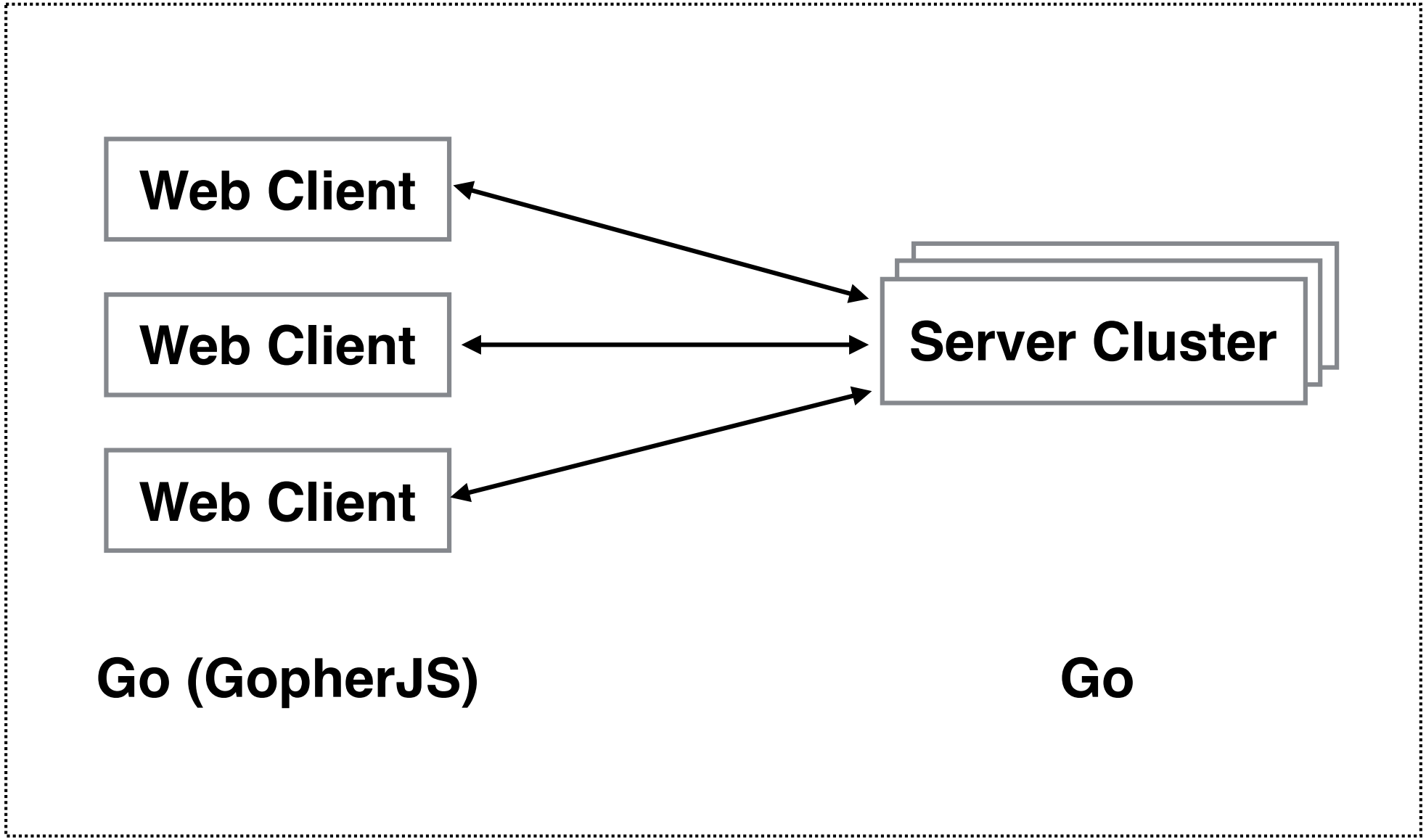
Rendered Client-side Template



Gob Data Over XHR

With Go on the front-end and Go on the back-end, we've created a Go-specific environment between web clients and the server cluster.

A Go-Specific Environment



Gob Data Over XHR

- This is an ideal scenario to use package encoding/gob as our means for data exchange with the server.
- In the following example, we transmit binary data that's encoded in the Gob format to the server.
- The data consists of an isomorphic Car struct slice.
- The struct is considered isomorphic, since it is used both in the front-end and the back-end.

Gob Data Over XHR (client-side)

```
var carsDataBuffer bytes.Buffer
enc := gob.NewEncoder(&carsDataBuffer)
enc.Encode(cars)

xhrResponse, err := xhr.Send("POST", "/cars-data",
carsDataBuffer.Bytes())

if err != nil {
  println(err)
}

println("xhrResponse: ", string(xhrResponse))
```

Gob Data Over XHR - Client Result

Gob Data Over XHR (server-side)

```
func handleCarsData(w http.ResponseWriter, r *http.Request) {

        var cars []model.Car
        var carsDataBuffer bytes.Buffer

        dec := gob.NewDecoder(&carsDataBuffer)
        body, err := ioutil.ReadAll(r.Body)
        carsDataBuffer = *bytes.NewBuffer(body)
        err = dec.Decode(&cars)

        w.Header().Set("Content-Type", "text/plain")

        if err != nil {
                log.Println(err)
                w.Write([]byte("Something went wrong, look into
it"))

        } else {
                fmt.Printf("Cars Data: %#v\n", cars)
                w.Write([]byte("Thanks, I got the slice of cars you
sent me!"))
        }

}
```

Gob Data Over XHR - Server Result

```
Cars Data: []model.Car{model.Car{ModelName:"Nano", Color:"Yellow", Manufacturer:"Tata"}, model.
Car{ModelName:"Ambassador", Color:"White", Manufacturer:"Peugeot"}, model.Car{ModelName:"Omni",
 Color:"Red", Manufacturer:"Maruti Suzuki"}}
```

Key Points

- GopherJS is interoperable with JavaScript
- Certain Go packages may not be [fully] supported by GopherJS, but there may be a suitable bindings package available to substitute their functionality
- We can make use of the html/template package to render client side templates allowing web apps to feel more responsive
- With Go on the front-end and Go on the back-end, we now have a Go-specific environment; an ideal scenario to use encoding/gob for our data exchange needs between client and server.

Special Thanks To

- Richard Musiol - Creator of GopherJS
- Dmitri Shuralyov - GopherJS Contributor
- The GopherJS Team

Questions / Feedback

kamesh@wirecog.com