ANGULARJS AND GO

V. GLENN TARCEA

GTARCEA@UMICH.EDU

NOVEMBER 15TH, 2014

TABLE OF CONTENTS

1 INTRODUCTION

1.1 ABOUT ME

- Glenn Tarcea
- Senior Developer at University of Michigan
- Current Project: Materials Commons

1.2 MATERIALS COMMONS

- Materials Commons is an online collaborative space for Metals Researchers
- We have open sourced all the code for Materials Commons:
  - Go, Javascript, Java, Python, Erlang, C
- You can find our code at:
  - https://github.com/materials-commons
  - https://github.com/prisms-center/materialscommons.org
- There are alot of nice (if sometimes a bit rough) packages:
  - Erlang: gen stomp, resource discovery, process monitoring, OS interfaces
  - Go: Utilities, config, file transfer, FlowJS server
  - Javascript: AngularStomp
  - Java: DM3 Parser for Tika (not touched in a while)

1.3 WHAT THIS TALK IS ABOUT

- This talk will cover creating a website using
  - Go and AngularJS
  - Websockets
  - REST
  - JWT
- The site will allow for simple "collaboration"
  - By using broadcasts to keep each site in sync

1.4 WHAT THIS TALK DOESN’T COVER

- This talk is not a Go or AngularJS tutorial
    - We will go over some aspects of both but will not spend a lot of time on the basics
- It won't cover all aspects of the application
    - We will elide some details but you can refer to the sample app to get all the details

1.5 WHERE TO GET THE APP

1.6 DEMO

- Demonstrate
  - Login/Logout
  - Reconnect/Disconnect
  - Multiple Browsers staying in sync

2 ANGULARJS SETUP

2.1 OVERVIEW

- We'll cover the basics of setting up an angular app and configuring the needed packages
- We use a few client libraries to make our lives easier
  - ui-router to give us multiple state based routes
  - ng-websocket for websocket communication
  - angular-jwt for easy JWT integration
  - Restangular for REST communication
- We will cover configuring and integrating these packages

2.2 MODULE REFERENCES

- Set references to our app modules.
  - We break our app into different modules for the application pieces in AngularJS.

```
var App = App || {};
App.Services = angular.module('app.services', []);
App.Controllers = angular.module('app.cntrlrs', []);
App.Filters = angular.module('app.filters', []);
App.Directives = angular.module('app.directives', []);
var app = angular.module('myapp', [
    "ui.router", "restangular",
    "app.services", "app.cntrlrs", "app.filters",
    "app.directives"
]);
```

2.3 CONFIGURE OUR ROUTES

- We set up routes to pages and views in our system

```
app.config(["$stateProvider", "$urlRouterProvider", "$httpProvider",
            "jwtInterceptorProvider",
            appConfig]);
function appConfig($stateProvider, $urlRouterProvider, $httpProvider,
                  jwtInterceptorProvider) {
    $stateProvider
        .state("login", {
            url: "/login",
            templateUrl: "app/login.html",
            controller: "loginController"
        })
        .state("users", {
            url: "/users",
            templateUrl: "app/users.html",
            controller: "usersController"
        })
        .state("users.add", {
            url: "/add",
            templateUrl: "app/add.html",
            controller: "addUserController"
        });

    // If the route isn't recognized goto /users
    $urlRouterProvider.otherwise("/users");
```

2.4 CONFIGURE AUTHENTICATION

- To configure authentication we need to
  - Control access to protected areas of our app
  - Track user authentication
  - Setup JWT Headers for all REST calls

2.5 CONTROLLING ACCESS

```
// appRun allows us to intercept different events while our
// application is running. Here it is used to control access
// to the application by requiring the user to login.
app.run(["$rootScope", "User", "$state", appRun]);
function appRun($rootScope, User, $state) {
    // $stateChangeStart is fired when a route change is starting.
    // Here we check if the user is already authenticatd. If they
    // aren't then we redirect them to the login page.
    $rootScope.$on('$stateChangeStart', function(event, toState, toParams) {
        if (!User.isAuthenticated()) {
            if (toState.url !== "/login") {
                // Cancel whatever route we were going to
                // and instead go to the login page.
                event.preventDefault();
                $state.go("login");
            }
        }
    });
}
```

2.6 CONFIGURING JWT

- The following code is also in appConfig (where we also configured the routes)
- It configures $http (and Restangular) to include the JWT token in all REST calls

```
// The JWT token is stored in sessionStorage. When our
// app starts up we explicitly clear the previous token.
sessionStorage.setItem("token", null);

// This interceptor will set the Authorization field
// in the header with the JWT token.
jwtInterceptorProvider.tokenGetter = function() {
    var token = sessionStorage.getItem("token");
    return token ? token : "";
};
$httpProvider.interceptors.push("jwtInterceptor");
```

2.7 CONFIGURE WEBSOCKETS

- Websockets uses events
- We only want to connect to the websocket after authentication
  - Unfortunately the Websocket spec doesn't allow us to add headers (JWT)
  - We could pass the token in the initial URL and then in each event to the server
    - We don't do this here but it is an option
- ws is a convience service we wrote

```
// Connect the socket
  $websocket.$new({
      url: ws.url(),
      reconnect: true,
      reconnectInterval: 500
  });

  // Wait on events
  var s = ws.get();
  s.$on("addeduser", function(user) {
      $timeout(function() {
          Users.add($scope.users, user);
      });
```

3 REST USING RESTANGULAR

3.1 OVERVIEW

- Restangular makes REST easy by
  - Providing Promises
  - Restangularizing your objects
    - Methods are attached to the returned object
    - You don't have to remember the
  - Easy to use API

3.2 EXAMPLE

This example demonstrates retrieving and updating a user

```
var user = Restangular.all("users", 123);
// change their name
user.fullname = "New Name";
user.post()
```

3.3 RECALL SENDING JWT

Recall that we configured the underlying $http service to include Authorization in the header with the JWT Token. Just to review:

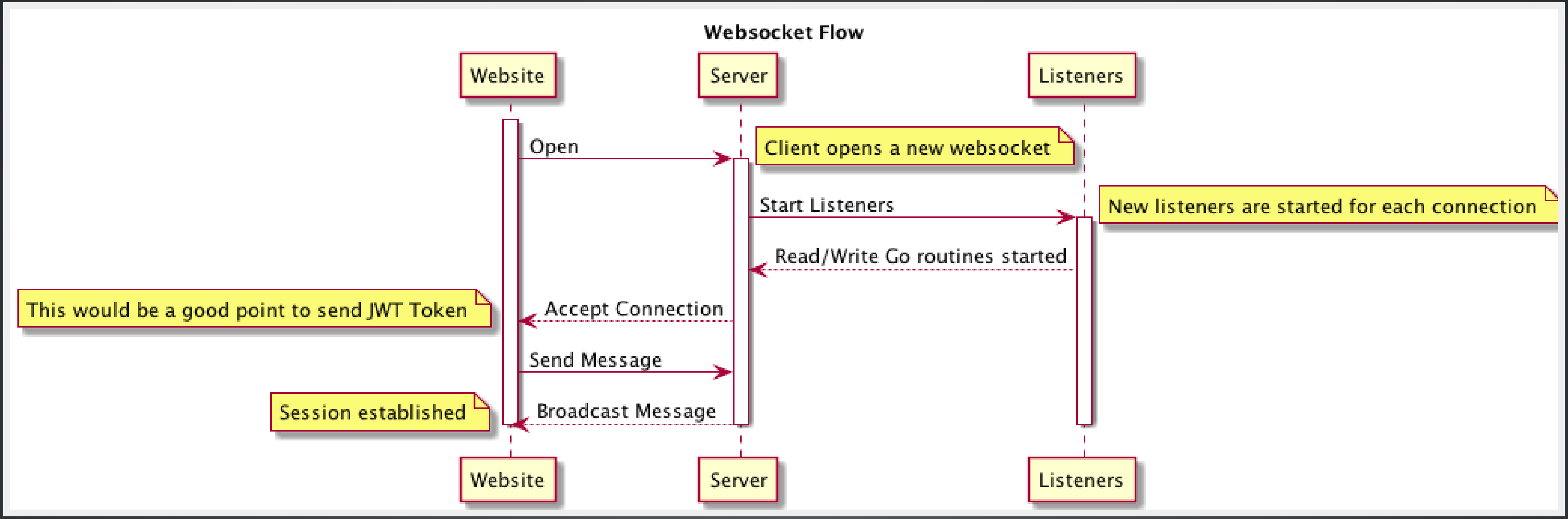
```
// The JWT token is stored in sessionStorage. When our
// app starts up we explicitly clear the previous token.
sessionStorage.setItem("token", null);

// This interceptor will set the Authorization field
// in the header with the JWT token.
jwtInterceptorProvider.tokenGetter = function() {
    var token = sessionStorage.getItem("token");
    return token ? token : "";
};
$httpProvider.interceptors.push("jwtInterceptor");
```

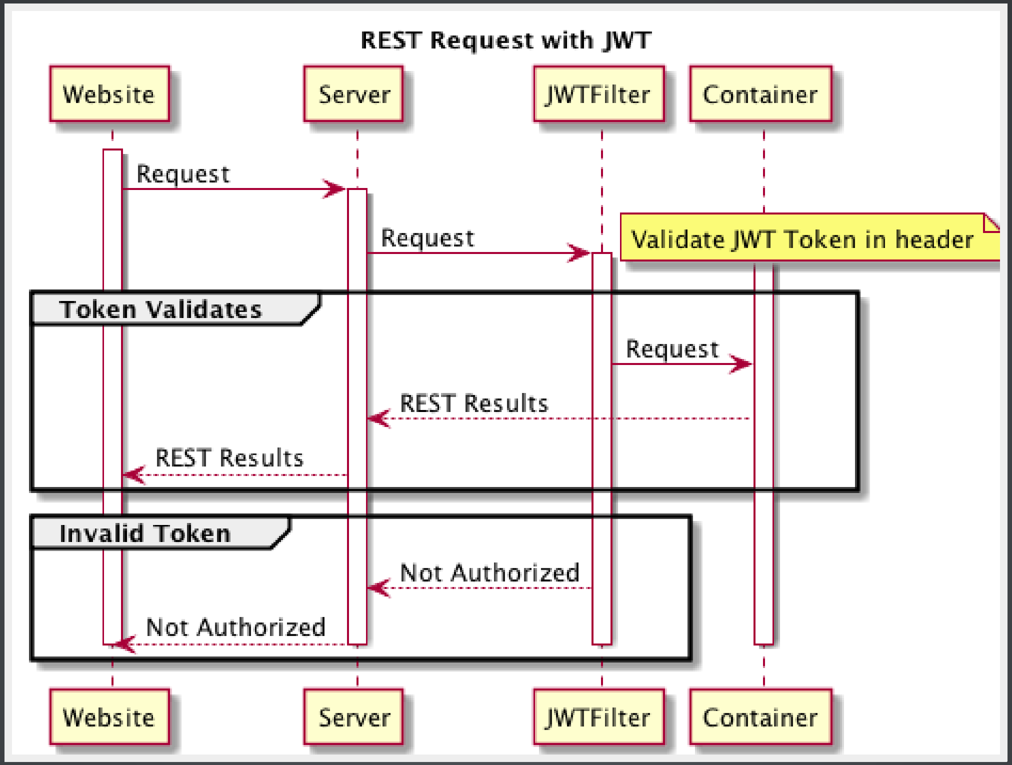
Now whenever we make a Restangular call the header is automatically included.

4 SERVICES OVERVIEW

4.1 WEBSOCKETS


Websocket Flow
Website
Server
Listeners
Open
Client opens a new websocket
Start Listeners
New listeners are started for each connection
Read/Write Go routines started
This would be a good point to send JWT Token
Accept Connection
Send Message
Session established
Broadcast Message
Website
Server
Listeners

4.2 REST AND JWT AUTHENTICATION



5 GO ROUTES SETUP

5.1 OVERVIEW

- Configure the Go HTTP server to handle:
  - Serving our website content
  - REST Calls
  - Websocket connections and broadcast
- Go has an HTTP interface that makes writing web servers and services very easy
  - This is one of the nicest pieces of using Go

5.2 GO WEB SERVER SETUP

- We'll point our web server at our website directory
- This will be our default route
    - The server will automatically pick up the index.html file

```
webdir := ...
dir := http.Dir(webdir)
http.Handle("/", http.FileServer(dir))
addr := "localhost:8081"
fmt.Println(http.ListenAndServe(addr, nil))
```

5.3 REST SETUP

- We'll use a nice REST extension package: go-restful
  - https://github.com/emicklei/go-restful
- Because this package uses HTTP interfaces we can use standard Go http to setup

```
container := ...

// All REST calls come through a /api/... route.
// We strip off /api before sending on to our
// container this way the container doesn't
// care about the prefix.
http.Handle("/api/", http.StripPrefix("/api", container))
```

5.4 WEBSOCKET SETUP

- Continuing the HTTP interface theme the Websocket is also handled through the HTTP handler

```
s := events.NewServer(hub)
http.Handle("/ws", websocket.Handler(s.OnConnection))
```

6 GO REST SERVICE

6.1 OVERVIEW

- Here we configure our REST service to handle different types of requests
- This example shows how we handle GET
- The syntax below means we can also use SWAGGER to document and expose our API
  - See:
    - Website: http://swagger.io/
    - Demo: http://petstore.swagger.wordnik.com/

```
ws := new(restful.WebService)
ws.Path("/users").
        Consumes(restful.MIME_JSON).
        Produces(restful.MIME_JSON)

ws.Route(ws.GET("").To(rest.RouteHandler(r.getAllUsers)).
        Doc("Retrieves all users").
        Writes([]schema.User{}))
```

6.2 JWT TOKEN CREATION

- To create the tokens we need a private and public key
- We then have our server read the files

```
# These commands were run to create our public/private files
openssl genrsa -out app.rsa 1024
openssl rsa -in app.rsa -pubout > app.rsa.pub
```

```
// At this point we have read the public and private keys
// Create the JWT Token
token := jwt.New(jwt.GetSigningMethod("RS256"))
token.Claims["ID"] = req.Username
token.Claims["exp"] = time.Now().Add(time.Hour * 72).Unix()
tokenStr, err := token.SignedString(r.privateKey)
if err != nil {
        return err, nil
}

auth := schema.Auth{
        Username: req.Username,
        Token:    tokenStr,
}
```

6.3 JWT TOKEN VERIFICATION

- We write an intercept filter that verifies the token

```
// Setup the filter for the container
 f := filters.NewJWTFilter(publicKey, "/users/login")
 container := restful.NewContainer()
 container.Filter(f.Filter)
```

```
// Verify the token on each rest call
func (f *jwtFilter) Filter(req *restful.Request, resp *restful.Response,
                           chain *restful.FilterChain) {
        // if the user is logging in for the first time then the
        // path will be f.loginPath. If that is the case then we just
        // go to the next filter because there is no token to
        // authenticate against.
        if req.Request.URL.Path != f.loginPath {

                token, err := jwt.ParseFromRequest(req.Request, f.getKey)
                if err != nil || !token.Valid {
                        fmt.Printf("invalid token for url %s: %s\n ", req.Requ
est.URL.Path, err)
                        resp.WriteErrorString(http.StatusUnauthorized, "Not au
thorized")
                        return
                }
        }
        chain.ProcessFilter(req, resp)
}

// Return the key jwt uses to validate a token.
func (f *jwtFilter) getKey(token *jwt.Token) (interface{}, error) {
        return f.publicKey, nil
```

6.4 SERVICE IMPLEMENTATION

```
func (r *usersResource) createUser(request *restful.Request,
        response *restful.Response, user schema.User) (error, interface{}) {

        var req userReq
        if err := request.ReadEntity(&req); err != nil {
                return err, nil
        }
        u, err := r.users.CreateUser(req.Email, req.Fullname)
        return err, u
}
```

7 GO WEBSOCKETS

7.1 OVERVIEW

- Because websockets are long lived there is a bit more we need to do with them.
  - Setup 2 go routines for reading/writing
  - For our purposes we need to register with our broadcaster (EventHub)

```
// OnConnection is called when a new websocket connection is made.
// It creates a persistent client connection and registers that
// connection with the hub. It it meant to be called by the
// websocket.Handler method.
func (s *Server) OnConnection(ws *websocket.Conn) {
	defer func() {
		ws.Close()
	}()

	client := NewClient(ws, s.hub)
	s.hub.Register(client)
	client.Listen()
}
```

7.2 READ HANDLING

- The read handler waits in an event loop
- The write side is similar(ish)

```
// readListener processes messages on the websocket.
func (c *Client) readListener() {
        for {
                select {
                case <-c.done:
                        c.hub.Unregister(c)
                        c.done <- true
                        return
                default:
                        var msg Message
                        err := websocket.JSON.Receive(c.ws, &msg)
                        switch {
                        case err == io.EOF:
                                c.done <- true
                                return
                        case err != nil:
                                c.done <- true
                                return
                        default:
                        }
                }
        }
}
```

8 CONCLUSION

- AngularJS and Go work well together
- The large number of standard libraries for each means you can easily create a reasonably complex application
- There is a lot of angst and questions on the web about using:
  - AngularJS client side authentication
  - JWT with AngularJS (and Go)
  - How to use Websockets
- Hopefully this talk and the example app at https://github.com/gtarcea/1DevDayTalk2014 will help you to get started
- If you have questions please contact me at glenn.tarcea@gmail.com
  - Or send me a pull request with a fix :-)