# 5
# Security

In this chapter we will authenticate users using a GitHub account and **OAuth 2.0** tokens. This will allow us to secure the site and support multiple users; currently we have a single hardcoded token and user. We will also add HTTPS to our site and explore some other modules that we can use to secure other common security vulnerabilities.

## Setting up Passport

Passport is an authentication middleware for node that supports; via plugin; multiple authentication strategies, including Basic Auth, OAuth, and OAuth 2. Passport works by defining a route middleware to be used to authenticate the request.

Let's install Passport:

```
npm install passport --save
```

Passport does not include a GitHub strategy; for this we need to install `passport-github`; a strategy for authenticating with GitHub using the OAuth 2.0 API:

```
npm install passport-github --save
```

## Acceptance testing with Cucumber and Zombie.js

OAuth authentication uses a callback mechanism; this is messy to test with an integration-testing tool such as SuperTest; we require something a little more end-to-end.

Cucumber allows teams to describe software behavior in a simple plain text language called **Gherkin**. The process of describing this behavior aids development; the output serves as documentation that can be automated to run as a set of tests. Let's install cucumber:

```
npm install -g cucumber
```

Zombie.js is simple, lightweight framework for doing headless full-stack testing. Let's install `Zombie.js`:

```
npm install zombie --save-dev
```

Let's automate running Cucumber with a grunt task:

```
npm install grunt-cucumber --save-dev
```

Add the following to our gruntfile `./gruntfile.js`. The section `files` defines the location of our feature files, and `options:steps` defines the location of our step definitions:

```
cucumberjs: {
  files: 'features',
  options: {
    steps: "features/step_definitions",
      format: "pretty"
  }
},
```

# Feature: Authentication

```
As a vision user
I want to be able to authenticate via Github
So that I can view project activity
```

Let's create our first feature file `./features/authentication.feature`. The following feature file contains a `Feature` section, which for the agile among you will know that it defines the story and its value to the business, and a list of scenarios. Our acceptance criteria; written in the Gherkin language.

The following `Authenticate` feature contains two scenarios, including one to log in, titled `User logs in successfully`, and one to log out, titled `User logs out successfully`:

```
Feature: Authentication
As a vision user
I want to be able to authenticate via Github
So that I can view project activity
```

```
  Scenario: User logs in successfully
    Given I have a GitHub Account
    When I click the GitHub authentication button
    Then I should be logged in
    And I should see my name and a logout link

  Scenario: User logs out successfully
    Given I am logged in to Vision
    When I click the logout button
    Then I should see the GitHub login button
```

Let's run Cucumber using our grunt task:

```
grunt cucumberjs
```

This will generate the following output:

```
2 scenarios (2 undefined)
7 steps (7 undefined)
You can implement step definitions for undefined steps with these
snippets:
this.Given(/^I have a GitHub Account$/, function(callback) {
  callback.pending();
});

this.When(/^I click the GitHub authentication button$/,
  function(callback) {
  callback.pending();
});

this.Then(/^I should be logged in$/, function(callback) {
  callback.pending();
});

this.Then(/^I should see my name and a logout link$/,
  function(callback) {
  callback.pending();
});

this.Given(/^I am logged in to Vision$/, function(callback) {
  callback.pending();
});
```

```
this.When(/^I click the logout button$/, function(callback) {
  callback.pending();
});

this.Then(/^I should see the GitHub login button$/,
  function(callback) {
  callback.pending();
});
```

From the preceding output, you can see that Cucumber has generated a series of stubbed steps that are set to `pending`. These steps represent the `Given`, `When`, and `Then` scenarios we defined in our feature file `./features/authentication/authentication.feature`.

We can use these steps to implement our Cucumber tests. Let's create a step definition file `./features/step_definitions/authentication/authenticate.js`:

```
var steps = function() {
  var Given = When = Then = this.defineStep;
  ..add generated steps here
};

module.exports = steps;
```

Let's run Cucumber using our grunt task:

```
grunt cucumberjs
```

We get the following output:

```
2 scenarios (2 pending)
7 steps (2 pending, 5 skipped)
```

We are now ready to begin implementing our first scenario.

# Scenario: User logs in successfully

Let's begin implementing this scenario. First, we need a GitHub `clientId` and `clientSecret`. Visit your GitHub account, click on **Settings** and then **Applications** and again on **Register New Application.** Complete the form by adding the `homepage` URL and the `callback` URL (same as our homepage), and a `clientId` and a `clientSecret` will be generated.

Let's add these details to our config files `./config/*.json`:

```
"auth": {
    "homepage": "http://127.0.0.1:3000"
  , "callback": "http://127.0.0.1:3000/auth/github/callback"
  , "clientId": "5bb691b4ebb5417f4ab9"
  , "clientSecret": "15310740929666983d52808dda32417d733791d0"
}
```

Let's remove the temporary login we set up in *Chapter 2*, *Building a Web API*, and remove the following line and all code related to it `./lib/routes/project.js`:

```
, login = require('../../test/login');
```

We are now ready to implement our GitHub strategy `./lib/github/authentication.js`. We start by defining a function, `GitHubAuth`; we import the `passport` and `passport-github` modules. We instantiate a `GitHubStrategy`, add it to `passport`, and pass a `clientID`, `clientSecret`, a `callbackUrl`, and a `verify` function (all passport strategies require a verify function), that is called when GitHub authenticates passing back an `accessToken`, `refreshToken`, and a `profile`.

Inside this verify function, we have the option of rejecting the user by passing a false out of the `callback` function. We will accept anyone with a GitHub access token; so simply pass back a user profile; which we create using the profile GitHub passed to us. Within the verify function, we instantiate a `GitHubRepo` and call `updateTokens`, which updates their access tokens for use by our Redis cache population.

Our application will support user sessions, so we add two functions to the `passport` module, that include `serializeUser` and `deserializeUser`, which serialize and deserializes the GitHub user profile into and out of a user session:

```
var async = require('async')
, GitHubRepo = require('../github')
, config = require('../configuration');

function GitHubAuth() {
  this.passport = require('passport')
  var GitHubStrategy = require('passport-github').Strategy;

  this.passport.use(new GitHubStrategy({
      clientID     : config.get('auth:clientId'),
      clientSecret : config.get('auth:clientSecret'),
      callbackURL  : config.get('auth:callback')
  },
```

```
    function(accessToken, refreshToken, profile, done) {

      var user = {
        id : profile.username,
        displayName : profile.displayName,
        token : accessToken
      };

      var git = new GitHubRepo(user.token, user.id);

      git.updateTokens(function(){
        process.nextTick(function () {
          return done(null, user);
        });
      });
    };
    ));

    this.passport.serializeUser(function(user, done) {
      done(null, user);
    });

    this.passport.deserializeUser(function(user, done) {
      done(null, user);
    });
  };

  module.exports = new GitHubAuth();
```

Let's add an `updateTokens` function to `GitHubRepo`, which gets all of a users' projects and `async.each` through `each` one updating its token:

```
  GitHubRepo.prototype.updateTokens = function(done) {
    var query = { "user" :  this.user };

    Project.find(query, function(error, projects) {
      if (error) return done();
      if (projects == null) done();

      async.each(projects, function(project, callback) {
        project.token = this.token;

        project.save(function(error, p) {
          callback();
        });
      }
      , function(error) {
```

```
        done();
      });
    });
};
```

Let's add configuration to our config files `./config/*.json`, in order to support Express sessions:

```
  "session": {
    "secret": "th1$1$a$ecret"
    , "maxAge": null
    , "secure": true
    , "httpOnly": true
  }
```

Let's wire up our GitHub strategy to our Express server: `./lib/express/index.js`. The first change we make it to include our new GitHub `authentication` strategy:

```
var gitHubAuth = require('../github/authentication')
```

We create a `cookieParser` middleware and include it just before the `bodyParser` middleware, which will parse the cookie header field and populate `req.cookies`. We pass a `secret`; which is a string used to create a signed cookie enabling the detection of a modified cookie:

```
var cookieParser = express.
  cookieParser(config.get('session:secret'));
app.use(cookieParser);
```

The application will require persistent login sessions, so we will include the connect `session` middleware in our Express server in order to provide session support. We will use the `sessionStore`, which is an in-memory session store. We pass in a `secret` and a value for a cookie `maxAge` (a null value will expire the session on closing the browser), `httpOnly` (disallow client-side JavaScript access to cookies; XSS attacks), and `secure` (send cookies over HTTPS only):

```
app.use(express.bodyParser());
var sessionStore = new express.session.MemoryStore();
app.use(express.session({ store: sessionStore,
  secret: config.get('session:secret'),
  cookie: { secure: config.get('session:secure'),
  httpOnly: config.get('session:httpOnly'),
  maxAge: config.get('session:maxAge') }}));
```

The Passport module requires we call `passport.initialize()` in order to initialize `passport`, and in order to provide session support, we must also call the `passport.session()` middleware; we add both to our Express server:

```
app.use(gitHubAuth.passport.initialize());
app.use(gitHubAuth.passport.session());
```

We now define the first of two routes on our Express server; both use the passport strategy for GitHub. The first route is a login route `/auth/github`; hitting this route will redirect you to GitHub and try to authenticate. If you are not logged in to GitHub, you will be asked to log in. If you are doing this for the first time, you will be prompted. You will be asked if you would like to grant Vision access. The second route; is the route GitHub will callback when authentication is complete:

```
app.get('/auth/github',
  gitHubAuth.passport.authenticate('github'),routes.auth.login);

app.get('/auth/github/callback',
  gitHubAuth.passport.authenticate('github',
    { failureRedirect: '/' }), routes.auth.callback);
```

We have configured our Express server with a GitHub passport strategy. Let's add the two missing routes to our routes, `./lib/routes/auth.js`; one for login and one for the callback as described previously:

```
exports.callback = function(req, res) {
  logger.info('Request.' + req.url);
  res.redirect('/');
};

exports.login = function(req, res){
  logger.info('Request.' + req.url);
};
```

In order to simulate the body of our project form containing a `user` and `token`, we will add a middleware that simply adds this data to the form for an authenticated user. We can add the `projectForm.addToken` middleware to all of our routes easily by using `app.all`, which will apply this middleware to all routes that follow it.

Let's make a further change to our Express server: `./lib/express/index.js`, and clean up our middleware by removing all require statements involving it and using `require-directory` with an `./lib/middleware/index.js` file, as we did with our routes. We can now add this `projectForm` above all the routes that require authentication:

```
    , middleware = require('../middleware')

app.all('*', middleware.projectForm.addToken);
.. all routes below
```

Let's create the `projectForm.addToken` middleware in `./lib/middleware/projectForm.js`. The `AddToken` middleware checks if the request is authenticated via `req.isAuthenticated`; we add `user` and `token` to the request:

```
exports.addToken = function(req, res, next){
  if (req.isAuthenticated()) {
    req.body.user = req.session.passport.user.id;
    req.body.token = req.session.passport.user.token;
    req.user = req.session.passport.user;
  };

  next();
}
```

Now that we have authentication in place, let's remove the hardcoded user in `./lib/routes/home.js`:

```
exports.index = function(req, res){
  var model = {
    title: 'vision.',
    description: 'a project based dashboard for github',
    author: 'airasoul',
    user: req.isAuthenticated() ? req.user.displayName : ''
  };

  res.render('index', model);
};
```

Now when we click on the GitHub logo in our header, we are redirected to GitHub which will ask you to log in. Once you have logged in to GitHub, you must grant access to our Vision application; however, future attempts to log in will not require you to grant access to Vision.

Let's complete our Cucumber steps for login using Zombie.js. `./features/step_definitions/authentication/authenticate.js`. First, we include zombie and and define a `steps` function. Then, we set `silent` and `debug` to enable Zombie.js debugging output. We define `Given = When = Then` as Cucumber steps and add a `Before` step, which runs before each test. From here we instantiate a zombie `Browser`:

```
var Browser = require('zombie')
, assert = require('assert')
S = require('string')
config = require('../../../lib/configuration');

var steps = function() {
  var silent = false;
```

```
  var debug = false;
  var Given = When = Then = this.defineStep;
  var browser = null;
  var me = this;

   this.Before(function(callback) {
     browser = new Browser();
     browser.setMaxListeners(20);
     setTimeout(callback(), 5000);
   });
};

module.exports = steps;
```

The step `I have a GitHub Account` uses the zombie browser to visit the GitHub login page, and waits for the page to load and fill in the login details; we then click on the sign in button:

```
this.Given(/^I have a GitHub Account$/, function(callback) {
  browser.visit('https://github.com/login',
    {silent: silent, debug: debug});

    browser.wait(function(){
      browser
        .fill('login', '#LOGIN#')
        .fill('password', '#PASSWORD#')
        .pressButton('Sign in', function() {
          callback();
    });
  });
});
```

The step `I click the GitHub authentication button` uses the zombie browser to visit the GitHub login page and waits for the page to load and fill in the login details; we then click on the sign in button:

```
this.When(/^I click the GitHub authentication button$/,
  function(callback) {
    browser.visit(config.get('auth:homepage'),
    {silent: silent, debug: debug});

    browser.wait(function(){
      browser
        .clickLink('#login', function() {
          callback();
```

```
        });
      });
});
```

The step `I should be logged in` uses the zombie browser to visit the GitHub login page and waits for the page to load and fill in the login details; we then click on the sign in button:

```
this.Then(/^I should be logged in$/, function(callback) {
  assert.ok(browser.success);
  callback();
});
```

The step `I should see my name and a logout link` uses the zombie browser to visit the GitHub login page and waits for the page to load and fill in the login details; we then click on the sign in button:

```
this.Then(/^I should see my name and a logout link$/,
  function(callback) {
  assert.equal(browser.text('#welcome'),
    'welcome Andrew Keig, click here to sign out');
      callback();
});
```

# Scenario: User logs out successfully

```
  Given I am logged in to Vision
  When I click the logout button
  Then I should see the GitHub login button
```

Let's add a logout route to our Express server: `./lib/express/index.js`:

```
app.get('/logout', routes.auth.logout);
```

Now add the route to our routes: `./lib/routes/auth.js`:

```
exports.logout = function(req, res){
  logger.info('Request.' + req.url);
  req.logout();
  res.redirect('/');
};
```

Let's complete our Cucumber steps for logout using Zombie.js in `./features/step_definitions/authentication/authenticate.js`

The step `I am logged in to Vision` uses the zombie browser to visit the Vision home page, waits for the page to load, and clicks on the login link:

```
this.Given(/^I am logged in to Vision$/, function(callback) {
  browser.visit(config.get('auth:homepage'),
  {silent: silent, debug: debug});

  browser.wait(function(){
    browser
    .clickLink('#login', function() {
      callback();
    });
  });
});
```

The step `I click the logout button` uses the zombie browser to visit the Vision home page, waits for the page to load, and clicks on the logout link:

```
this.When(/^I click the logout button$/, function(callback) {
  browser.visit(config.get('auth:homepage'),
  {silent: silent, debug: debug});

  browser.wait(function(){
    browser
    .clickLink('#logout', function(err) {
      callback();
    });
  });
});
```

The step `I should see the GitHub login button` checks to see if the browser response returns a `success`, and then checks to see if the GitHub login link is accessible:

```
this.Then(/^I should see the GitHub login button$/,
  function(callback) {
  assert.ok(browser.success);
  var containsLogin =
    S(browser.html('#login')).contains('vision/github.png')
    assert.equal(true, containsLogin);
    callback();
  });
```

# Securing our site with HTTPS

In order to make our site secure, we will run the entire application under HTTPS. We will need two files: a PEM encoded SSL certificate `./lib/secure/cert.pem`, and a private key `./lib/secure/key.pem`. In order to create an SSL certificate, we first need to generate a private key and a certificate signing request (CSR). For development purposes, we will create a self-signed certificate. Run the following commands:

```
cd ../vision/lib/secure
```

```
openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout key.pem
-out cert.pem
```

Upon running the second command, you will enter an interactive prompt to generate a 2048-bit RSA private key and a certificate signing request (CSR). You will need to enter various pieces of information including address details, common name or domain name, company details, and an email address.

Let's add a module, `./lib/express/server.js`, that will create a HTTP server based on the `key/cert` we have just created. We import the `https` module, read the `key` and `cert` files from disk, and add them to a options object. Then using the `https` module, we create a server passing in these options:

```
var fs = require('fs')
, https = require('https');

function Server(app){
  var httpsOptions = {
    key: fs.readFileSync('./lib/secure/key.pem'),
      cert: fs.readFileSync('./lib/secure/cert.pem')
  };

  return
    https.createServer(httpsOptions,app).listen(app.get('port'));
}

module.exports = Server;
```

Let's use the `server` from within our Express server `./lib/express/index.js`; remove the line that creates our HTTP server:

```
var httpServer = http.createServer(app).listen(app.get('port'));
```

Replace it with a call to our new HTTPS server:

```
var server = require('./server')(app);
```

Now we need to replace all references to `http://127.0.0.1:3000`; port 3000 with `https://127.0.0.1:8443`; port 8443. Our config file contains two references:

```
"auth": {
    "homepage": "https://127.0.0.1:8443"
  , "callback": "https://127.0.0.1:8443/auth/github/callback"
  , "clientId": "5bb691b4ebb5417f4ab9"
  , "clientSecret": "15310740929666983d52808dda32417d733791d0"
  },
```

We have a further reference in our `backbone.js` script `./public/components/vision.js`. When connecting to our Socket.IO server, we pass a URL `127.0.0.1:3000`. We make another important change here; we pass an options object when connecting to Socket.IO with the setting `secure: true, port: '8443'`:

```
Vision.Application = function(){
  this.start = function(){
    var socketio = io.connect('/', {secure: true, port: '8443'});
    var router = new Vision.Router(socketio);
    Backbone.history.start();
    router.navigate('index', true);
  }
};
```

# Sharing Express sessions with Socket.IO

Now that we have session support in place, we can share the session with Socket.IO allowing us to accept or reject the connection based on this session data. Express and Socket.IO do this using a handshake mechanism. When a client connects to the server, the handshake is initiated, which consists of executing an authorization function on Socket.IO. Here, the cookie associated with the handshake request is examined and rejected if invalid. Let's install `session.socket.io`; a module that has wrapped up this process:

```
npm install session.socket.io --save
```

First off, let's change our Express server, `./lib/express/index.js`, and pass to our `SocketHandler` module the `sessionStore` and the `cookieParser`:

```
var socketHandler = new SocketHandler(httpServer, sessionStore,
  cookieParser);
```

The `SocketHandler` module now accepts the parameters `httpServer`, `sessionStore`, and `cookieParser`. The `SocketHandler` will now instantiate a `SessionSockets` module passing `socketIo`, the `sessionStore` module, and the `cookieParser`. We change the `connection` event to listen on the `SessionSockets` module instead of the `socket.Io` module so that we can access the `session`. Now from within the `subscribe` event, we can check to ensure the `session.passport.user` is valid. We call `session.touch` which updates the `maxAge` and `lastAccess` properties of a session:

```
function SocketHandler(httpServer, sessionStore, cookieParser) {
  var socketIo = new Socket(httpServer)
  var sessionSockets = new SessionSockets(socketIo, sessionStore,
    cookieParser);

  sessionSockets.on('connection', function(err, socket, session) {
    subscriber.subscribe("issues");
    subscriber.subscribe("commits");

    subscriber.client.on("message", function (channel, message) {
      socket.broadcast.to(message.projectId)
      .emit(channel, JSON.parse(message));
    });

    socket.on('subscribe', function (data) {
      var user = session ? session.passport.user : null;
      if (!user) return;
      socket.join(data.channel);
      session.touch();
    });
  });

  sessionSockets.on('error', function() {
    logger.error(arguments);
  });
};

module.exports = SocketHandler;
```

# Cross-site request forgery

**Cross-site request forgery** (**CRSF**) is an attack that tricks the victim into executing malicious actions on a web application in which they are authenticated. Connect/Express comes packaged with a Cross-site request forgery protection middleware. This middleware allows us to ensure that a request to a mutate state is from a valid source. The CRSF middleware creates a token that is stored in the requests session as `_csrf`. A request to our Express server will then need to pass the token in the header field `X-CSRF-Token`.

Let's create a security `./lib/security/index.js` module that adds the `csrf` middleware to our application. We define a function, `Security`, that takes an Express `app` as an argument and removes the middleware when in `TEST` or `COVERAGE` mode.

```
var express = require('express');

function Security(app) {
  if (process.env['NODE_ENV'] === "TEST"  ||
    process.env['NODE_ENV'] === "COVERAGE") return;

  app.use(express.csrf());
};

module.exports = Security;
```

Let's make a change to our Express server `./lib/express/index.js`. The `crsf` middleware requires session support, so we add the following line below the `session` and `passport` middleware:

```
require('../security')(app);
```

As we are using `backbone.js` that uses jQuery under the hood to make AJAX requests, we will need to make a change to our backbone code `./public/components/vision/vision.js`. We will now override the `Backbone.sync` function, so that all requests through it pass the `X-CSRF-Token` in the header. The `X-CSRF-Token` is pulled from a `meta` tag in the master page:

```
Backbone.sync = (function(original) {
  return function(method, model, options) {
    options.beforeSend = function(xhr) {
      var token = $("meta[name='csrf-token']").attr('content');
      xhr.setRequestHeader('X-CSRF-Token', token);
    };
    original(method, model, options);
  };
})(Backbone.sync);
```

We now need to pass the `X-CSRF-Token` to our master page via the master page route. The token is stored in the requests session as `_csrf`, in the following code we add the token to `csrftoken` in our view object:

```
exports.index = function(req, res){
  var model = {
    title: 'vision.',
    description: 'a project based dashboard for github',
```

```
      author: 'airasoul',
      user: req.isAuthenticated() ? req.user.displayName : '',
      csrftoken: req.session._csrf
    };

    res.render('index', model);
};
```

The `csrftoken` is rendered in our master page in a `meta` tag called `csrf-token`; the backbone sync method will put it from this meta tag:

```
<meta name="csrf-token" content="{{csrftoken}}">
```

# Improving security with HTTP headers and helmet

Helmet is a collection of middleware that implements various security headers for Express; for more information on helmet visit `https://npmjs.org/package/helmet`.

Helmet supports the following:

- csp (Content Security Policy)
- HSTS (HTTP Strict Transport Security)
- xframe (X-FRAME-OPTIONS)
- iexss (X-XSS-PROTECTION for IE8+)
- contentTypeOptions (X-Content-Type-Options nosniff)
- cacheControl (Cache-Control no-store, no-cache)

Let's extend our security `./lib/security/index.js` module, and add helmet security for the previous issues:

```
var express = require('express')
, helmet = require('helmet');

function Security(app) {
  if (process.env['NODE_ENV'] === "TEST"  ||
    process.env['NODE_ENV'] === "COVERAGE") return;

  app.use(helmet.xframe());
  app.use(helmet.hsts());
  app.use(helmet.iexss());
  app.use(helmet.contentTypeOptions());
```

```
    app.use(helmet.cacheControl());
    app.use(express.csrf());
};

module.exports = Security;
```

# Summary

By default, Express uses in-memory sessions. In the next chapter we will move our sessions to Redis. We will also configure Socket.IO to use Redis and explore some other interesting ways of scaling Express.