

Contributions to Lab Security Enhancement

WEB APP WITH PASSPORT-LOCAL STRATEGY

RAGHUL SOMINENI RAGHUPATHY (RSR379)

N11371498

ABHISHEK BALAJI VENKATARAAMAN SANGEETHA (AVS395) N17497529

Authenticating Node.js Applications with Passport

As a part of the security enhancement lab which dealt with Poly Password hasher, we tried to create a Node.js application from scratch which uses a relatively new but very popular middleware – Passport to take care of our authentication requirements.

Passport does a very good job at separating the other functionalities of a web app from the authentication needs. Passport can be easily configured into any Express-based web application.

Passport provides us with over 140+ authentication mechanisms from which we can choose from. For this Lab, we used the Local Authentication Strategy of Passport and authenticate the users against a locally configured Mongo DB instance, storing the user details in the database.

Now we will talk about the steps we took to implement a simple web app with passport authentication.

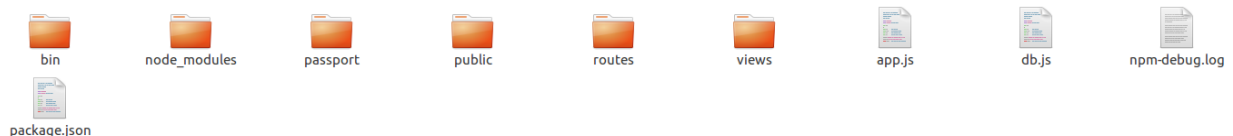
We worked on a Unix based system. Before we started with the application development we had to install Node Package Manager (npm) which is necessary for node.js applications.

Setting Up the Application:

First, we had to install Express & express-generator to generate a boilerplate application by simply executing:

```
express passport-mongo
```

The generated application structure looked something like this:



Now we can go ahead and delete unwanted dependencies and also delete the user.js route and remove its references from the app.js file.

Adding Dependencies for the Project

In order to use the passport authentication strategies, we have to create dependencies in the package.json file. This can be done by entering the following in the file:

```
"passport": "~0.2.0",
```

```
"passport-local": "~1.0.0"
```

Where in the passport-local specifies that we are going to be using the Local Authentication strategy of the passport for the web application. In our application, we decided to use the MongoDB to save the user details. Now to install and save the dependency to package.json, we simply had to run the following command:

```
npm install mongoose --save
```

After this is done, we can run the application by executing the following command:

```
npm install && npm start
```

The app can be viewed at <http://localhost:3000/>. Initially, it is just a basic web page with no functionalities. We are going to be creating pages for login and registration pages later on.

Creating the DB Model – Mongoose

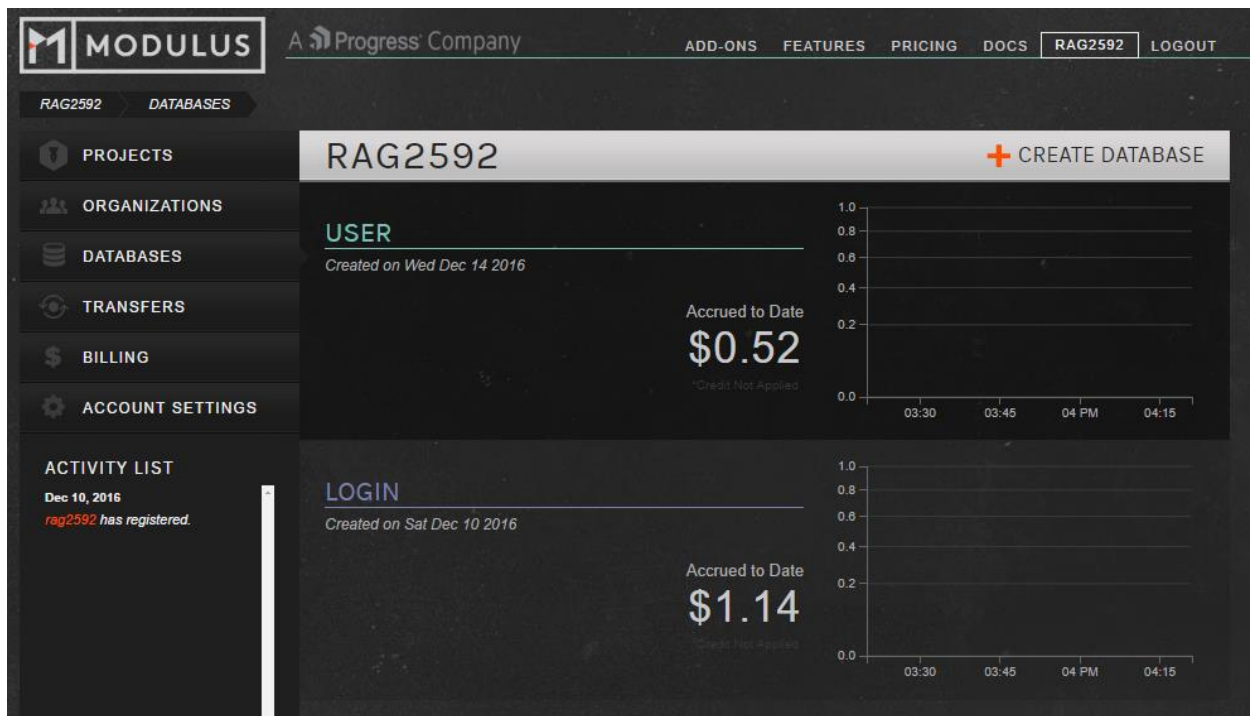
As already said, we would be saving the login details on MongoDB. In order to do that, we have to create a user model and put it in the `models/user.js` file. This should look like:

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

module.exports = mongoose.model('User', {
  id: String,
  username: String,
  password: String,
  email: String,
  firstName: String,
  lastName: String
});
```

DB Configuration – Mongo

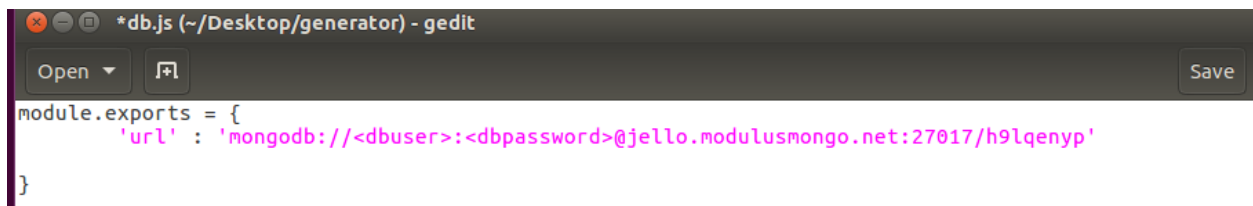
I have used the cloud database service Modulus as my DB service provider. First, I created a DB in modulus, which looks like this:



After the DB has been successfully created, we get a link that can be used to access the DB. The link looks like this:

```
MONGO URI
mongodb://<user>:<pass>@jello.modulismongo.net:27017/h9Iqenyp
```

Now we put this onto a new file db.js, just to have the configuration of the database on a separate file.

A screenshot of a code editor window titled 'db.js (~/Desktop/generator) - gedit'. The editor shows a JavaScript object with a 'url' property. The text is:

```
module.exports = {
  'url' : 'mongodb://<dbuser>:<dbpassword>@jello.modulismongo.net:27017/h9Iqenyp'
}
```

Now we should link this db.js file in the app.js file, this is done by just adding the lines:

```
var dbconfig = require('./db.js');
var mongoose = require('mongoose');
mongoose.connect(dbconfig.url);
```

Passport Configuration

Passport just handles the authentication part of the web app, we had to implementing session handling by ourselves. This was done by using the express-session. In order to use the express-session on our app, we added the following code to the file app.js:

```
var passport = require('passport');
var expressSession = require('express-session');
app.use(expressSession({secret: 'mySecretKey'}));
app.use(passport.initialize());
app.use(passport.session());
```

Now we also add dependencies in the package.json file for express-session. This again was done by executing the command:

```
npm install --save express-session
```

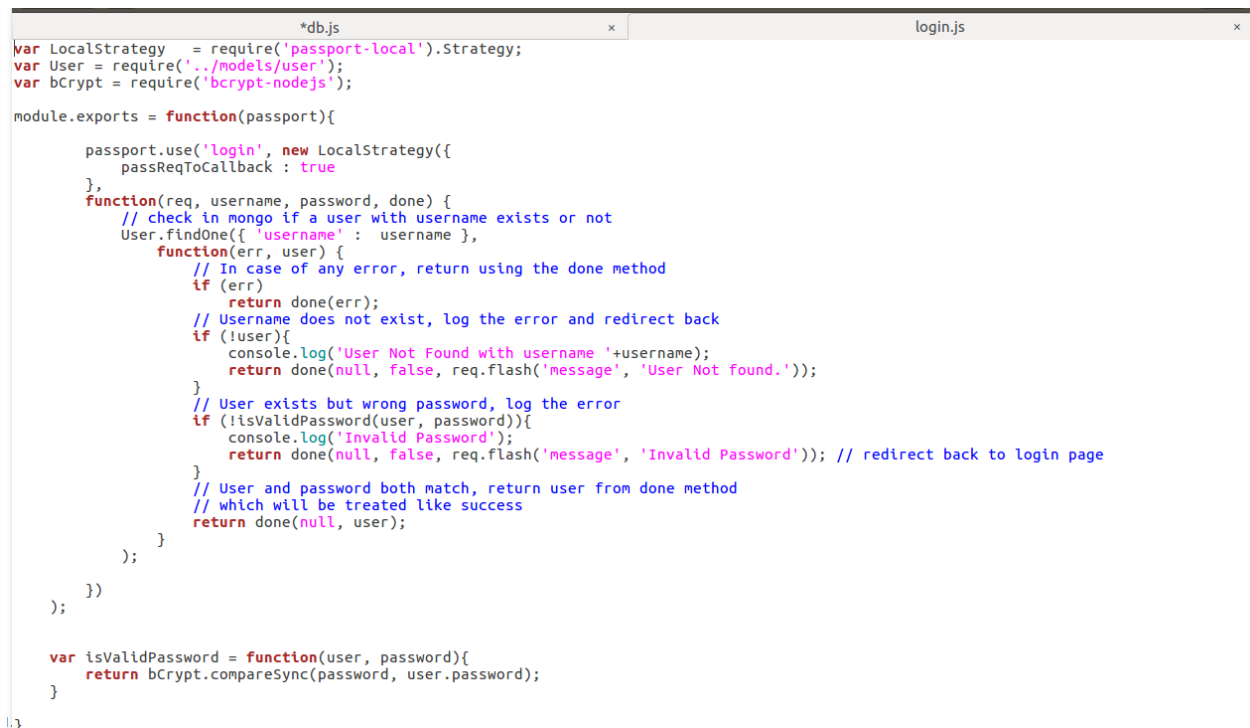
Passport provides us with two methods called `serializeUser` and `deserializeUser` which serializes and deserializes user instance from a session store in order to support login sessions, so that every subsequent request will not contain the user credentials.

Strategies for Passport

The next step is to implement **login** and **signup** strategies for passport. Each of the two would be an instance of the Local Authentication Strategy of the Passport. These would be created using the `passport.use()` function.

Login Strategy

The Login strategy of our Web App looks like this:



```
*db.js x login.js x
var LocalStrategy = require('passport-local').Strategy;
var User = require('../models/user');
var bcrypt = require('bcrypt-nodejs');

module.exports = function(passport){
  passport.use('login', new LocalStrategy({
    passReqToCallback : true
  },
  function(req, username, password, done) {
    // check in mongo if a user with username exists or not
    User.findOne({ 'username' : username },
      function(err, user) {
        // In case of any error, return using the done method
        if (err)
          return done(err);
        // Username does not exist, log the error and redirect back
        if (!user){
          console.log('User Not Found with username '+username);
          return done(null, false, req.flash('message', 'User Not found.'));
        }
        // User exists but wrong password, log the error
        if (!isValidPassword(user, password)){
          console.log('Invalid Password');
          return done(null, false, req.flash('message', 'Invalid Password')); // redirect back to login page
        }
        // User and password both match, return user from done method
        // which will be treated like success
        return done(null, user);
      }
    );
  })
);

var isValidPassword = function(user, password){
  return bcrypt.compareSync(password, user.password);
}
```

If you look at the parameters being passed to the function `passport.use()`, you can see that the first one is the **name** of the strategy which can be used to identify the strategy. The second parameter is the **type** of passport strategy we

would be using, clearly this is going to be **LocalStrategy**. Now, it is understood that the LocalStrategy looks for credentials which are username and password, but it also allows any other named parameters. The **passReqToCallback** config variable allows us to access the **request** object in the callback, thereby enabling us to use any parameter associated with the request.

Next we use the Mongoose API to authenticate the user with the ones already available on the DB. In the parameter list of the function, the last parameter is **done** denotes a useful method which would specify the success or failure of the passport module.

We are also using an encryption technique to make the app more secure, as passwords in general are weak in nature. This is done in the last part of the code wherein we are using the **bcrypt-nodejs** to take care of the encryption and decryption of the passwords.

Registration Strategy

The next strategy is the registration strategy and our implementation of the same looks like this:

```

*db.js
x
login.js

var LocalStrategy = require('passport-local').Strategy;
var User = require('../models/user');
var bcrypt = require('bcrypt-nodejs');

module.exports = function(passport){

  passport.use('signup', new LocalStrategy({
    passReqToCallback : true // allows us to pass back the entire request to the callback
  },
    function(req, username, password, done) {

      findOrCreateUser = function(){
        // find a user in Mongo with provided username
        User.findOne({ 'username' : username }, function(err, user) {
          // In case of any error, return using the done method
          if (err){
            console.log('Error in SignUp: '+err);
            return done(err);
          }
          // already exists
          if (user) {
            console.log('User already exists with username: '+username);
            return done(null, false, req.flash('message', 'User Already Exists'));
          } else {
            // if there is no user with that email
            // create the user
            var newUser = new User();

            // set the user's local credentials
            newUser.username = username;
            newUser.password = createHash(password);
            newUser.email = req.param('email');
            newUser.firstName = req.param('firstName');
            newUser.lastName = req.param('lastName');

            // save the user
            newUser.save(function(err) {
              if (err){
                console.log('Error in Saving user: '+err);
                throw err;
              }
              console.log('User Registration succesful');
              return done(null, newUser);
            });
          }
        });
      };

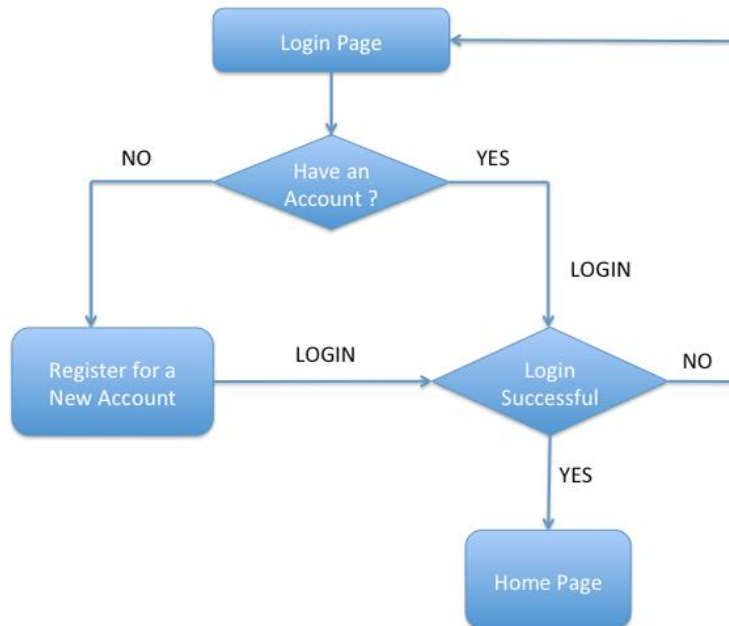
      // Delay the execution of findOrCreateUser and execute the method
      // in the next tick of the event loop
      process.nextTick(findOrCreateUser);

    })
  );

  // Generates hash using bcrypt
  var createHash = function(password){
    return bcrypt.hashSync(password, bcrypt.genSaltSync(10), null);
  }
}
```


As we did in the Login Strategy, we use the Mongoose API to make sure that the user trying to register is not already present in the DB. If not, then a new entry is created for the user saving his information on the DB.

Application's Flow or Routes



Now we had to create an index.js page which would take care of the routes being taken by the user of the application, which would look like this:

```

var express = require('express');
var router = express.Router();

var isAuthenticated = function (req, res, next) {
  // if user is authenticated in the session, call the next() to call the next request handler
  // Passport adds this method to request object. A middleware is allowed to add properties to
  // request and response objects
  if (req.isAuthenticated())
    return next();
  // if the user is not authenticated then redirect him to the login page
  res.redirect('/');
}

module.exports = function(passport){
  /* GET login page. */
  router.get('/', function(req, res) {
    // Display the Login page with any flash message, if any
    res.render('index', { message: req.flash('message') });
  });

  /* Handle Login POST */
  router.post('/login', passport.authenticate('login', {
    successRedirect: '/home',
    failureRedirect: '/',
    failureFlash : true
  }));

  /* GET Registration Page */
  router.get('/signup', function(req, res){
    res.render('register',{message: req.flash('message')});
  });

  /* Handle Registration POST */
  router.post('/signup', passport.authenticate('signup', {
    successRedirect: '/home',
    failureRedirect: '/signup',
    failureFlash : true
  }));

  /* GET Home Page */
  router.get('/home', isAuthenticated, function(req, res){
    res.render('home', { user: req.user });
  });

  /* Handle Logout */
  router.get('/signout', function(req, res) {
    req.logout();
    res.redirect('/');
  });


  return router;
}

```

In the above code, the most important part is the use of the function **passport.authenticate()**, which makes sure that the correct strategy is called based on the route which is being taken.

Now that we have completed the defining the strategies and also defined the routes properly, the next step would be give a proper look for our page. For this purpose, two views were created. **Layout.jade** contains the basic layout & styling information. **Index.jade** contains the login page with routes for signup.

Final View Of the Application



Email

Password

Sign in

[Create an account](#)