Web Development

Security Single Page Applications

Description

This assignment adds security to the ongoing class assignment. Previous assignments implemented login and registration, but did not enforce security. Securing single page applications (SPAs) consists of securing access to sensitive views and securing access to APIs and Web services. PassportJS, a Node.js security module, will be used to secure the SPA with username and password, as well as authority providers such as Google and Facebook.

Configure Express Session Support

In **`server.js`**, load and configure the server to use cookie based session support. Use the following code as an example.

```
var cookieParser = require('cookie-parser');
var session      = require('express-session');
...
app.use(cookieParser());
app.use(session({ secret: process.env.SESSION_SECRET }));
```

From the terminal command line install the modules locally using **`npm`**

```
npm install cookie-parser --save
npm install express-session --save
```

Make sure both the **`cookie-parser`** and **`express-session`** are declared as a dependency in the **`package.json`** file.

```
"dependencies": {
    ...
    "cookie-parser": "^1.4.1",
    "express-session": "^1.13.0",
    ...
}
```

Configure and Initialize Passport and Passport Session Support

Load the passport module, initialize it, and configure passport's session support. Configure passport session after configuring express session.

```
var passport = require('passport');
...
```

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

Configure Passport User Serialization and Deserialization

In **user.service.server.js**, implement the **serializeUser** method to store an encrypted representation of the user in a cookie. This will allow Passport to maintain session information for the currently logged in user.

```
passport.serializeUser(serializeUser);

function serializeUser(user, done) {
    done(null, user);
}
```

Implement the **deserializeUser** method to retrieve the currently logged in user from the encrypted cookie created in **serializeUser**. The function can use the user's primary key to verify the user still exists in the database. Below is a sample implementation.

```
passport.deserializeUser(deserializeUser);

function deserializeUser(user, done) {
    developerModel
        .findDeveloperById(user._id)
        .then(
            function(user){
                done(null, user);
            },
            function(err){
                done(err, null);
            }
        );
}
```

Implement Passport Local Strategy

The passport local strategy allows implementing simple username and password based authentication. At the top of your **user.service.server.js** Web service, load the **LocalStrategy**

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;
```

Configure the local strategy to parse the **username** and **password** from the request and then use the **userModel** to retrieve the user by **username** and **password**. Use code similar to the code below

```
passport.use(new LocalStrategy(localStrategy));

function localStrategy(username, password, done) {
    userModel
```

```
        .findUserByCredentials(username, password)
        .then(
            function(user) {
                if(user.username === username && user.password === password) {
                    return done(null, user);
                } else {
                    return done(null, false);
                }
            },
            function(err) {
                if (err) { return done(err); }
            }
        );
}
```

Refactor Login to Use Passport

Refactor Login Controller to Use Passport

In **`user.controller.client.js`**, refactor the **`login`** event handler to use the **UserService.login()** service

```
  ...
  vm.login = login;

  function login(user) {
      UserService
          .login(user)
          .then(
              function(response) {
                  var user = response.data;
                  $rootScope.currentUser = user;
                  $location.url("/user/"+user._id);
              }
  ...
```

Refactor Login User Client Service to Use Passport

Refactor the **`login`** service in **`user.service.client.js`** to post the login request to the user Web service. The user argument should have properties username and password, each set to the corresponding username and password of the user login in. The service should return a promise for the controller to register a callback.

```
function login(user) {
    return $http.post("/api/login", user);
}
```

Refactor Login Web Service to Use Passport

In **`user.service.server.js`**, add an endpoint to handle a login request from the browser. Use the **`passport.authenticate`** middleware to have passport handle the request before the **`login`** handler. The

authenticate middleware will parse the username and password from the request and search the user by credentials. If the user is found, the **login** function is invoked and the current user is available in **req.user**.

```
app.post  ('/api/login', passport.authenticate('wam'), login);
...
function login(req, res) {
    var user = req.user;
    res.json(user);
}
```

Implement Logout Using Passport

Implement Logout Controller to Use Passport

In **user.controller.view.js**, implement an event handler bound to the logout button in the profile view. Use the **logout** client service in **UserService** to issue a logout request to the server. On successful logout, set the **currentUser** to **null**. Use the code below as an example.

```
...
var vm = this;
...
vm.logout = logout;
...
function logout() {
    UserService
        .logout()
        .then(
            function(response) {
                $rootScope.currentUser = null;
                $location.url("/");
            }
...
```

Implement Logout User Client Service to Use Passport

In **user.service.client.js**, add a logout API to post a logout request to the server. The API should return a promise for the controller to register a callback and receive a server response.

```
var api = {
    ...
    logout: logout,
    ...
};
return api;

function logout(user) {
    return $http.post("/api/logout");
}
```

Implement Logout Web Service to Use Passport

In **user.service.server.js**, add a logout Web service endpoint to handle logout requests from the browser.

```
app.post('/api/logout', logout);
```

Implement the logout service in a function of the same name: logout. Use the request's logout function to invalidate the currently logged in user.

```
function logout(req, res) {
    req.logOut();
    res.send(200);
}
```

Refactor User Registration to Use Passport

Refactor User Registration Controller to Use Passport

In **user.controller.client.js**, refactor the success callback function for **UserService.register()** to store the currently logged in user into the **$rootScope**. Use the code below as an example.

```
UserService
    .register(user)
    .then(
        function(response) {
            var user = response.data;
            $rootScope.currentUser = user;
            $location.url("/user/"+user._id);
        }
...
```

Refactor User Registration User Client Service to Use Passport

In **user.service.client.js**, add a **register** service to the API and implement it by posting a **register** request to the server. The user instance must include properties **username** and **password** with values of the registering user. Use the following code as an example.

```
var api = {
    ...
    register: register
    ...
};
return api;

function register(user) {
    return $http.post("/api/register", user);
}
```

Refactor User Registration Web Service to Use Passport

In **user.service.server.js**, implement a Web service endpoint mapped to a POST **/api/register** to handle user registration. Once the user is created in the database, use the request's **login()** to set the current user. Use the code below as an example.

```
app.post ('/api/register', register);

function register (req, res) {
    var user = req.body;
    userModel
        .createUser(user);
        .then(
            function(user){
                if(user){
                    req.login(user, function(err) {
                        if(err) {
                            res.status(400).send(err);
                        } else {
                            res.json(user);
                        }
                    });
                }
            }
        );
}
```

Verify User Login Using Passport

Protect the Profile Page

In **config.js** configure a **resolve** for the profile route to check if the current user is logged in. Do not allow navigation to the profile unless the user is logged in. Verify user login with a Web service mapped to **/api/loggedin**. Use the code below as an example.

```
.when ("/user", {
    templateUrl: "views/user/profile.view.client.html",
    controller: "ProfileController",
    controllerAs: "model",
    resolve: { loggedin: checkLoggedin }
})

var checkLoggedin = function($q, $timeout, $http, $location, $rootScope) {
    var deferred = $q.defer();
    $http.get('/api/loggedin').success(function(user) {
        $rootScope.errorMessage = null;
        if (user !== '0') {
            $rootScope.currentUser = user;
            deferred.resolve();
        } else {
```

```
            deferred.reject();
            $location.url('/');
        }
    });
    return deferred.promise;
};
```

Implement Logged In Web Service to Use Passport

Implement a Web service to check whether the current user is logged in. Map the Web service to **`/api/loggedin`** and use the request's **`isAuthenticated()`** to check if passport has already authenticated the user in the session

```
app.get ('/api/loggedin', loggedin);

function loggedin(req, res) {
    res.send(req.isAuthenticated() ? req.user : '0');
}
```

Implement PassportJS Facebook Strategy

In **`user.schema.server.js`**, add a subdocument to capture the user's facebook identity.

```
var UserSchema = mongoose.Schema({
    facebook: {
        id:    String,
        token: String
    }
});
```

In **`user.model.server.js`**, add a **`findUserByFacebookId`** API to retrieve a user by their facebook ID. Implement in a function of the same name

```
var api = {
    findUserByFacebookId: findUserByFacebookId,
};

function findUserByFacebookId(facebookId) {
    return User.findOne({'facebook.id': facebookId});
}
```

In **`login.view.client.html`**, add a facebook login button referencing a Web service configured to login the user using facebook

```
<a href="/auth/facebook" class="btn btn-primary btn-block">
    <span class="fa fa-facebook"></span>
    Facebook
</a>
```

In **`user.service.server.js`**, load the passport facebook strategy

```
var passport         = require('passport');
var FacebookStrategy = require('passport-facebook').Strategy;
```

Create a Web service that uses **`passport.authenticate()`** to delegate authentication to facebook

```
app.get ('/auth/facebook', passport.authenticate('facebook', { scope : 'email' }));
```

Facebook will call back to a URL configured at their developer website, e.g.,

```
app.get('/auth/facebook/callback',
    passport.authenticate('facebook', {
        successRedirect: '/#/user',
        failureRedirect: '/#/login'
}));
```

From facebook's developer's Website, configure a client ID (application ID), client secret, and callback URL. Configure these in the following environment variables locally and on OpenShift:

`FACEBOOK_CLIENT_ID,`
`FACEBOOK_CLIENT_SECRET,`
`FACEBOOK_CALLBACK_URL`

These can be loaded from the service into a local object as follows:

```
var facebookConfig = {
    clientID     : process.env.FACEBOOK_CLIENT_ID,
    clientSecret : process.env.FACEBOOK_CLIENT_SECRET,
    callbackURL  : process.env.FACEBOOK_CALLBACK_URL
};
```

Use the facebook configuration to register a middle tier that will handle facebook related requests

```
passport.use(new FacebookStrategy(facebookConfig, facebookStrategy));
```

When Facebook calls back, it will pass a token and profile information that can be cached in the data model keyed off of facebook's profile ID. Use the ID to look up the user in the database. If the user is not there, store the as a new user, and logged them in. If the user is already there, then don't create them and just logged them in.

```
function facebookStrategy(token, refreshToken, profile, done) {
    developerModel
        .findUserByFacebookId(profile.id)
```

Encrypt Passwords

Install Node.js Encryption Library

From the command line, use **npm** to install the Node.js library **bcrypt-nodejs**. Don't forget to use the **--save** option to configure the library as a dependency of this project.

```
npm install bcrypt-nodejs --save
```

Make sure the library is listed in the **package.json** file

```
"dependencies": {
    "bcrypt-nodejs": "0.0.3",
}
```

In **user.service.server.js**, load the encryption library to encrypt passwords when registering new users and decrypt passwords when logging in.

```
var bcrypt = require("bcrypt-nodejs");
```

Encrypt Passwords on Registration

In **user.service.server.js**, refactor the registration logic to encrypt the user password before saving the new user to the database

```
user.password = bcrypt.hashSync(user.password);
return userModel.createUser(user);
```

Decrypt passwords on Login

In **user.service.server.js**, refactor the login logic to compare encrypted passwords to identify the current user.

```
if(user && bcrypt.compareSync(password, user.password)) {
    return done(null, user);
} else {
    return done(null, false);
}
```

Implement Client Side Validation and Error Handling

Use angular validation and controller logic to enforce the following form element requirements. Highlight invalid fields and display an error message below the invalid field. Display an error alert box at the top of the view styled as a bootstrap alert box, e.g., **class="alert alert-danger"**. Only display error messages after a form has been submitted. Turn off autocorrect and auto capitalization for username field in the login view.

View	Field	Constraint
Login	**username, password**	required
Register	**username, password, verify password**	required
	password, verify password	must match
New Website	**name**	required
Edit Website	**name**	required
New Page	**name**	required
Edit Page	**name**	required
New Widget	**name**	required
Edit Widget	**name**	required

Deliverables

GitHub and OpenShift Deliverables

To allow TAs and instructor to see your changes, please frequently commit and push your work to GitHub and OpenShift repositories. Below is an example of the commands you will use. The example assumes your project is located in **~/summer2016/web-dev**:

```
> cd ~/summer2016/web-dev
> git add .
> git commit -m 'A comment describing your work'
> git push github
> git push openshift
```

Verify that the files have copied to the github repository. Also visit your OpenShift website and verify that your changes are reflected on the remote server.

Tagging a Release

When you consider your work complete and ready for evaluation (ready for release), go to your code repository in GitHub and generate a release by navigating to "releases". Then click on "Create a new release" and type the name of the tag in the input field labeled "Tag version". We will be using the following tags for the various assignments:

assignment5 (previous assignment)
assignment6 (this assignment)

If you need to resubmit the assignment then create a new tag by adding a version number, e.g.,

assignment6.1, assignment6.2, etc...

We will grade the very last release. The date/time you create the tag will be considered the date/time of submission. If you have questions on how to create tags or have any problem at all, please do not hesitate to give me a call at (978) 761-5742 and we can jump on a Google Hangout and I can walk you through the process.

Blackboard Deliverables

Please submit the following in Blackboard

1. GitHub repository URL
2. OpenShift Website URL