LAB 20e

NODE AUTHENTICATION

What You Will Learn

- How to use a cloud-based MongoDB platform
- How to perform authentication and authorization using Passport

Approximate Time

The exercises in this lab should take approximately 45 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: March 23, 2019

USER AUTHENTICATION

In this lab, you will be making use of MongoDB Atlas, a hosted alternative to having your own Mongo installation.

PREPARING DIRECTORIES

1 This lab has additional content that you will need to copy/upload this folder into your eventual working folder/workspace.

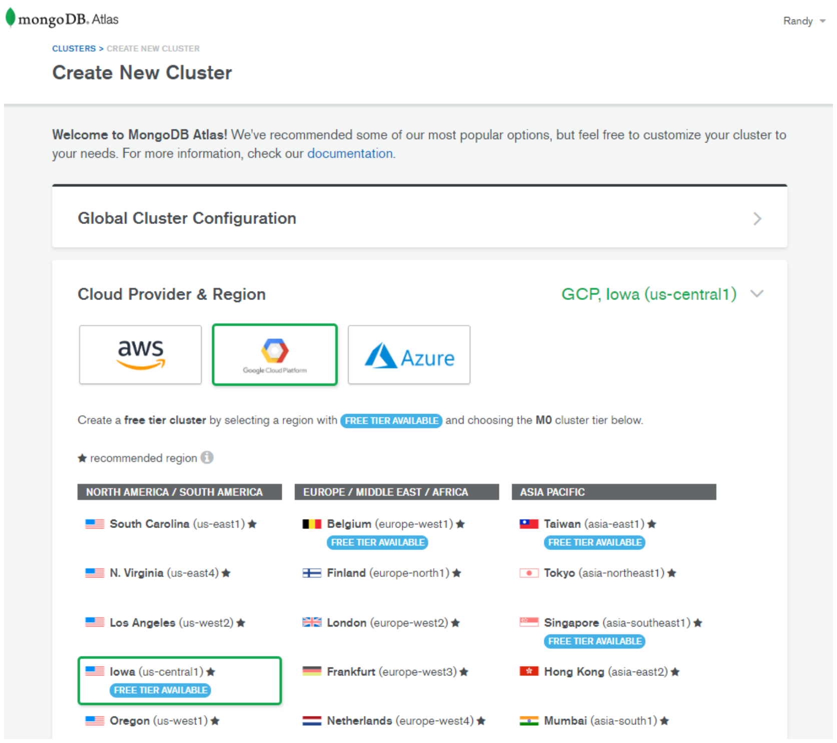
Exercise 20e.1 — CONFIGURING MONGODB ATLAS

1 Navigate to `https://www.mongodb.com/cloud/atlas` and choose the Start free option.

This will require creating a MongoDB account.

2 After registering, you will be asked to set up your cluster. We want to stick within the free tier, so after selecting your cloud provider (I used Google Cloud Platform), select a region with the free tier and be sure to choose an M0 cluster, which is a small shared sandboxed environment that will remain free. Give your cluster a name (I named mine `funwebdev-cluster`). Click the Create Cluster button.

It takes a surprisingly long time to create the cluster, maybe as long as 10 minutes.



Create New Cluster

Welcome to MongoDB Atlas! We've recommended some of our most popular options, but feel free to customize your cluster to your needs. For more information, check our documentation.

Global Cluster Configuration

Cloud Provider & Region — GCP, Iowa (us-central1)

Create a **free tier cluster** by selecting a region with FREE TIER AVAILABLE and choosing the **M0** cluster tier below.

★ recommended region

NORTH AMERICA / SOUTH AMERICA	EUROPE / MIDDLE EAST / AFRICA	ASIA PACIFIC
South Carolina (us-east1) ★	Belgium (europe-west1) ★ FREE TIER AVAILABLE	Taiwan (asia-east1) ★ FREE TIER AVAILABLE
N. Virginia (us-east4) ★	Finland (europe-north1) ★	Tokyo (asia-northeast1) ★
Los Angeles (us-west2) ★	London (europe-west2) ★	Singapore (asia-southeast1) ★ FREE TIER AVAILABLE
Iowa (us-central1) ★ FREE TIER AVAILABLE	Frankfurt (europe-west3) ★	Hong Kong (asia-east2) ★
Oregon (us-west1) ★	Netherlands (europe-west4) ★	Mumbai (asia-south1) ★

Figure 20e.1 – Configuring your MongoDB Atlas cluster

3 After the cluster is created (or while you are waiting), you can create a database user. This is **not** the same as your MongoDB website account name. This is a user name associated just with the database in this cluster.

Click the **Create Your First User** option from the Get Started button. Alternately, click on the Security Tab for your cluster. Click the Add New User button.

Add a user named `testuser-1` and any password you like. Since this password will show up eventually in our source code, in this example I am setting the password to `testpassword`. Also select the **Read and write to any database** privilege. Click the **Add User** button.

4 The next step is to Whitelist your project's IP address. Normally, this would be the same as the IP address of your development machine and your production server.

Click the **Whitelist Your IP address** option from the Get Started button. Alternately, click on the Security Tab for your cluster, then click the IP Whitelist subtab.. Click the **Add IP Address** button.

Because you may want to develop from a variety of locations (home, university, etc), for now choose the **Allow Access From Anywhere** button. If you were working on a real production application, you would explicitly specify an AP address instead. Click **Confirm**.

This step might take several minutes.

5 Finally, you are ready to connect to your cluster.

Click the **Connect to your cluster** option from the Get Started button. Alternately, click on the Overview Tab for your cluster, then click the **Connect** button. Follow the instructions for connecting via mongo shell or via the MongoDB Compass application.

6 We need to add data now to our cluster. Begin by clicking the **Collections** button from the Overview tab. From the Collection subtab, click the **Create Database** button. Name the database **funwebdev** and the collection **users**.

7 Just as in the previous lab, you will need to use the mongoimport command from your development computer but specify the MongoDB Atlas database.

Click on the **Command Line Tools** tab and scroll down to the example of the command for the `mongoimport` command and use **Copy** button to put command in the clipboard.

8 I would recommend switching to Notepad or some other text editor and then paste the command currently in the clipboard.

9 Edit the command by replacing `<PASSWORD>`, `<DATABASE>`, `<COLLECTION>`, `<FILETYPE>`, and `<FILENAME>` with the following: `testpassword`, `funwebdev`, `users`, `json`, `Logins.json`.

10 While still in the text editor, add the following flag to the end of the command:

```
--jsonArray
```

If you don't add this flag to the mongoimport command, you will get a "JSON decoder out of sync" error message.

11 Copy this revised command in the text editor onto the clipboard.

12 In your Terminal / Command prompt, navigate to the data folder for this lab.

13 Paste your mongoimport command from the clipboard into the terminal and then press enter to execute the command. It should work!

The first time I tried this, I got a strange "cert already in hash table" error. Rerunning the command a few more times got things working for me.

14 If everything worked, examine the new data via the Collection subtab of the MongoDB Atlas website.

We are now ready to use Node to access this data.

Exercise 20e.2 — Setting Up Node Project

1 Navigate to your working folder and enter the following command.

```
npm init
```

2 If you haven't already installed nodemon, do so now via:

```
npm install -g nodemon
```

3 Install several packages at once via the following command.

```
npm install --save express mongoose dotenv
```

4 Install more packages at once via the following command.

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

This installs some standardized express middleware, to support server sessions, the parsing of cookies, and a way to push error messages to view template files.

5 Install even more packages via the following command.

```
npm install --save ejs express-ejs-layout
```

This installs our view/template engine (instead of using pug, this lab will use ejs).

6 Install even more packages via the following command.

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

This installs a mechanism for encrypting/decrypting bcrypt digests

7 Install even more packages via the following command.

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

This installs the relevant passport packages. We will use passport as our authentication library.

8 You have been provided with a `.env` file, modify the MONGO_URL value using the Connection value you can find in the MongoDB Atlas web page (see Step 5 of previous exercise). Be sure to replace `<PASSWORD>` with your password (e.g., `testpassword`) for this user.

9 Examine `dataConnector.js` in the `handlers` folder. Notice that the `mongoose.connect` call needs to provide the name of the database on Atlas via the `dbName` property. Edit this value if your database name on Atlas is not `funwebdev`.

10 Test your MongoDB Atlas connection via:

```
node mongo-tester
```

If successful, you should see message "connected to MongoDB Atlas".

Exercise 20e.3 — Examine the Existing Files

1 Examine `app.js`. It already has a preliminary structure. Notice the new content (compared to previous labs), that enables express sessions and cookie support.

2 Examine `single-user.json`. It illustrates the structure of a single `User` object in the file `Logins.json`.

3 In the `models` folder, edit the file `User.js` as follows.

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

// define a schema that maps to the structure of the data in MongoDB
const userSchema = new mongoose.Schema({
   id: Number,
   details: {
    firstname: String,
    lastname: String,
    city: String,
    country: String
   },
   picture: {
    large: String,
    thumbnail: String
   },
   membership: {
    date_joined: String,
    last_update: String,
    likes: Number
   },
   email: String,
   password_bcrypt: String,
   apikey: String
 });

 module.exports = mongoose.model('User', userSchema, 'users');
```

4 Examine some of the files in the `views` folder. Unlike the previous lab, which used pug as the view engine, this lab uses ejs, which has a syntax similar to PHP or ASP. The file `layout.ejs` describes the markup for each view.

5 Examine `apiRouter.js`. These will provide JSON data from our database.

Eventually this API will only be available once the user is logged in.

6 You should be able to view the initial home page via:

```
node app
```

Visit `http://localhost:8080/` to see the home page. Click one of the API links inn the header to see one of the APIs.

Exercise 20e.4 — Implementing the Passport Authentication

1 In the `handlers` folder, create a new file named `auth.js`. Add the following content.

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const UserModel = require('../models/User.js');

// maps the passport user+passwd fields to the names of fields in database
const localOptions = {
  usernameField : 'email',
  passwordField : 'password'
};

// define strategy for validating login
const strategy = new LocalStrategy(localOpt, async (email, password, done) => {
  try {
    // Find the user in the DB associated with the email provided by the user
    const userChosen = await UserModel.findOne({ email: email });

    if( !userChosen ){
      //If the user isn't found in the database, set flash message
      return done(null, false, { message : 'Email not found'});
    }
    // Validate password and make sure it matches with the corresponding hash
    //stored in the database. If the passwds match, it returns a value of true.
    const validate = await userChosen.isValidPassword(password);
    if( !validate ){
      return done(null, false, { message : 'Wrong Password'});
    }
    // Send the user information to the next middleware
    return done(null, userChosen, { message : 'Logged in Successfully'});
  }
  catch (error) {
    return done(error);
  }
});

// for localLogin, use our strategy to handle User login
passport.use('localLogin', strategy);

// by default, passport uses sessions to maintain login status  ...
// you have to determine what to save in session via serializeUser
// and deserializeUser. In our case, we will save the email in the session data
passport.serializeUser( (user, done) => done(null, user.email) );

passport.deserializeUser( (email, done) => {
  UserModel.findOne({ email: email }, (err, user) => done(err, user) );
});
```

Exercise 20e.5 — Adding Password and Authentication Checks

1 In the `models` folder, add the following function to the file `User.js` as follows.

```
// We'll use this later on to make sure that the user trying to log in has the
// correct credentials. Can't be arrow syntax because need 'this' within it
userSchema.methods.isValidPassword = async function(formPassword) {
    const user = this;
    const hash = user.password;
    // Hashes the password sent by the user for login and checks if the hashed
    // password stored in the database matches the one sent. Returns true if it
    // does else false.
    const compare = await bcrypt.compare(formPassword, hash);
    return compare;
}
```

2 In the `handlers` folder, create a new file named `helpers.js`. Add the following content.

```
// uses passport authentication infrastructure to check if authentication is
// needed at some point in middleware pipeline.
function ensureAuthenticated (req, resp, next) {
   if (req.isAuthenticated()) {
     return next();
   }
   req.flash('info', 'Please log in to view that resource');
   resp.render('login', {message: req.flash('info')} );
}

module.exports = {
   ensureAuthenticated
};
```

Notice the use of req.flash. The flash mechanism in express is a way to create pseudo global variables. In this case, the string 'Please log in to view that resource' is first being set to a flash variable named 'info'. Then in the resp.render() call, this flash variable content is being passed to the login page,

Exercise 20e.6 — Setting up the Open Routes

1 Add the following requires to the top of `openRouter.js` as follows:

```
const express = require('express');
const passport = require('passport');
const helper = require('./helpers.js');
```

2 Modify the route handler for the root path in `openRouter.js` as follows:

```
router.get('/', helper.ensureAuthenticated, (req, resp) => {
   resp.render('home', {user: req.user});
});
```

Adding our helper function to the route handler will ensure that our site only renders the home page if the user is logged in. If the user is logged in, then we will pass the user object for the logged in user to the ejs page.

3 Modify the route handler for the login path in `openRouter.js` as follows:

```
router.get('/login', (req, resp) => {
   resp.render('login', {message: req.flash('error')} );
});
```

If there is an error flash message (see Exercise 20e.5 for more info on flash messages), then pass it to the login page.

4 Modify the route handler for the logout path in `openRouter.js` as follows:

```
router.get('/logout', (req, resp) => {
   req.logout();
   req.flash('info', 'You were logged out');
   resp.render('login', {message: req.flash('info')} );
 });
```

Passport is handling the sessions used to indicate the logged-in status; the req.logout() lets passport remove the logged-in user from its session state.

5 We now need to implement the actual log in process. This will happen when the user posts from the login form. Add the following route hander:

```
router.post('/login', async (req, resp, next) => {
   // use passport authentication to see if valid login
   passport.authenticate('localLogin',
                          { successRedirect: '/',
                            failureRedirect: '/login',
                            failureFlash: true })(req, resp, next);
});
```

Exercise 20e.7 — Hiding the API Paths

1 Add the following reference to the top of apiRouter.js as follows:

```
const express = require('express');
const ImageModel = require('../models/Image.js');
const UserModel = require('../models/User.js');
const helper = require('./helpers.js');
```

2 Modify the route handler for the first path in `apiRouter.js` as follows:

```
router.get('/images/:id', helper.ensureAuthenticated, (req, resp) => {
```

3 Add the same `ensureAuthenticated` check to the other paths in `apiRouter.js`.

Exercise 20e.8 — Configuring the Views

1 Modify `home.ejs` in the `views` folder as follows:

```
<div class="container">
   <h1 class="title">Home</h1>
   <div class="box">
      <p>Welcome <%= user.details.firstname %> <%= user.details.lastname %></p>
   </div>
</div>
```

Back in Step 2 of Exercise 20e.6, you passed the user object to this view.

2 Modify `login.ejs` in the `views` folder as follows (some code omitted):

```
<div class="control">
   <button type="submit" class="button is-link">Login Locally</button>
</div>
<%
   if (message) { %>
   <p id="msg2" class="help is-danger">
      <%= message %>
   </p>
<% } %>
```

Back in Step 3 and 4 of Exercise 20e.6, you passed the message object to this view.

Exercise 20e.9 — Final Changes to App and Testing

1 Now go back to `app.js` and ad the following requires:

```
const passport = require('passport');
const flash = require('express-flash');
```

2 Further down in that file, add the following code:

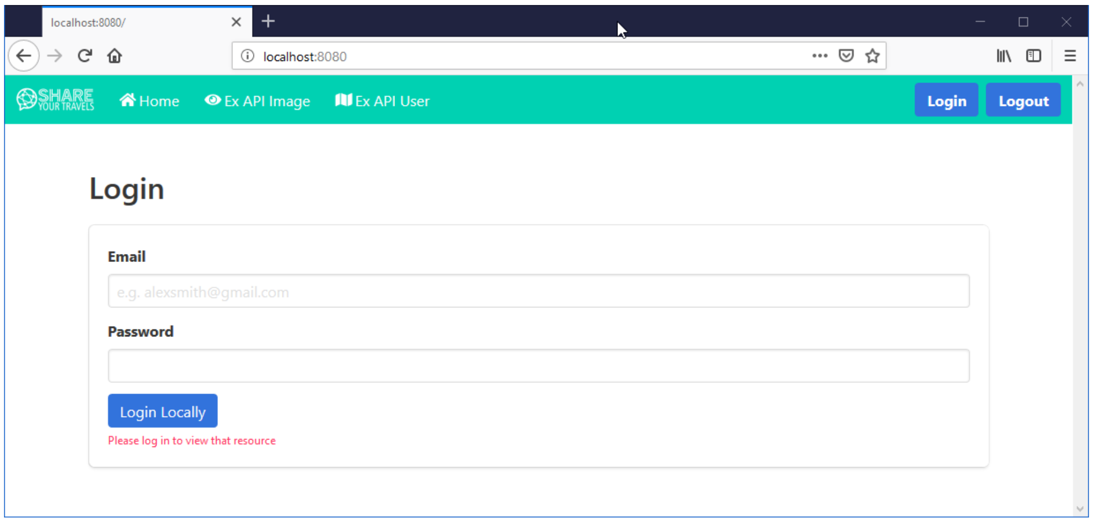
```
/* ----- add new code here ------- */

// Use express flash
app.use(flash());
// set up the passport authentication
require('./handlers/auth.js');
// set up route handlers
const openRoutes = require('./handlers/openRouter.js');
app.use('/', openRoutes);
// these routes only if logged in
const apiRoutes = require('./handlers/apiRouter.js');
app.use('/api', apiRoutes );
```

3 You should be able to view the initial home page via:

```
nodemon app
```

Visit `http://localhost:8080/` to see the home page. You should be taken to the login page. Try logging in using "al@ace.ca" as the email and "mypassword" as the password. The login should work and redirect you to the home page. Try also requesting one of the APIs. It should work when logged in, but if you try logging out, it shouldn't work.



Login

Email

e.g. alexsmith@gmail.com

Password

Login Locally

Please log in to view that resource

Figure 20e.2 – When requesting resource before logged in

Test Your Knowledge #1

Move along ... nothing here yet ...