LAB 13b

INTERMEDIATE NODE

What You Will Learn

- How to use the EJS templating system
- 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: November 19, 2021

Intermediate Topics Overview

In this lab, you will be covering a variety of more advanced Node techniques. The first of these is how to integrate a templating system into Node so that it can be used in a way similar to PHP. This lab also illustrates how to use the Passport package to implement authentication. **The authentication system uses MongoDB as the underlying database, so it is strongly recommended that you first work through the exercises in Lab 14b.**

Preparing Directories

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

Exercise 13b.1 — Setting Up Node Project

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

   ```
   npm init
   ```

2. Install several packages at once via the following command.

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

3. Install even more packages via the following command.

   ```
   npm install -save ejs
   ```

   This installs our view/template engine.

4. Examine `api-router.js`. Notice that it defines three routes.

5. Run `book-server.js` and then test in browser with the following requests.

   ```
   http://localhost:8080/api/all
   http://localhost:8080/api/isbn10/0321865820
   http://localhost:8080/api/title/calc
   http://localhost:8080/static/images/0132145375.jpg
   ```

Adding a View/Template Engine

In the first Node lab, you used Node as a file and API server. You may have wondered if Node can be used in a way similar to PHP. The answer is yes. It is possible to make use of a view engine to programmatically render HTML in Node. Perhaps the most popular of these are **EJS** (Embedded JavaScript templating) and **Pug** (formerly called Jade).

The way a view engine works is that you create your views using some specialized format that contains presentation information plus JavaScript coding (this file is the *template*). This template file is somewhat analogous to PHP files in that they are a blend of markup and programming).

With EJS, you specify your presentation in `.ejs` files, which uses regular HTML with JS embedded within `<% %>` tags. An EJS view has a similar feel to PHP in that you can mix markup and programming code (except the programming language with EJS is JavaScript).

Express has built-in support for view engines. You only need to install the appropriate package using npm, and then tell Express which folder contains the view files and which view engine to use to display those files.

Exercise 13b.2 — Introducing EJS

1. Examine `public/layout.html` and `public/single-book.html`. These files provide the basic markup you will be implementing in EJS.

2. Create a new folder in your application named `views`.

   This folder is going to contain your .ejs files.

3. Create a folder within `views` named `partials`.

4. Create a new file named `head.ejs` within the `partials` folder.

5. Copy and paste the contents of the `<head>` section from `layout.html` into this file.

6. Create a new file named `header.ejs` within the `partials` folder. Copy and paste the `<header>` element from `layout.html` into this file.

7. Create a new file named `sidebar.ejs` within the `partials` folder. Copy and paste the `<section>` element with the class of `sidebar` from `layout.html` into this file.

8. Create a new file named `footer.ejs` within the `partials` folder. Copy and paste the `<footer>` element from `layout.html` into this file.

9. Create a new file within `views` named `home.ejs`.

10 Add the following content to `home.ejs`.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <%- include('./partials/head'); %>
</head>
<body>
   <%- include('./partials/sidebar); %>
   <main class="book">
      <%- include('./partials/header'); %>
      <section class="pagecontent">
         Home content here
      </section>
   </main>
   <%- include('./partials/footer'); %>
</body>
</html>
```

11 Modify `book-server.js` as follows.

```
/* --- middle ware section --- */

// view engine setup
app.set('views', './views');
app.set('view engine', 'ejs');
```

12 Modify `book-server.js` as follows.

```
/*--- add in site page requests ----*/

app.get('/', (req, res) => {
    res.render('home.ejs');
});
```

13 Re-run `book-server.js` and test by requesting `http://localhost:8080/`

This should display the home page view. In the next exercise you will pass data to the page.

Exercise 13b.3 — Passing Data to an EJS

1 Modify `book-server.js` as follows.

```
app.get('/', (req, res) => {
    res.render('home.ejs',
        { data1: 'hello', data2: 'world'} );
});
```

2 Modify `home.ejs` as follows.

```
<section class="pagecontent">
   <%= data1 %> <%= data2 %>
</section>
```

3 Re-run `book-server.js` and test by requesting `http://localhost:8080/`

The page should now display the passed in data.

4 Add a new route by modifying `book-server.js` as follows.

```
app.get('/site/list', (req, res) => {
    res.render('list.ejs', { books: controller.getAll() } );
});
```

The getAll() method returns an array of book objects.

5 Make a copy of `home.ejs` named `list.ejs`.

6 Change the pagecontent `<section>` of `list.ejs` as follows:

```
<section class="pagecontent">
   <h1>List of Books</h1>
   <div class="booklist">
      <% books.forEach(book => { %>
         <div>
            <a href="/site/book/<%= book.isbn10 %>">
               <img src="/static/images/<%= book.isbn10 %>.jpg"
                    width="100" />
            </a>
         </div>
         <h3><%= book.title %></h3>
      <% }); %>
   </div>
</section>
```

Notice that this page loops through the passed book data and outputs the relevant markup.

7 Re-run `book-server.js` and test by requesting `http://localhost:8080/site/list`

The results should look similar to Figure 13b-1.

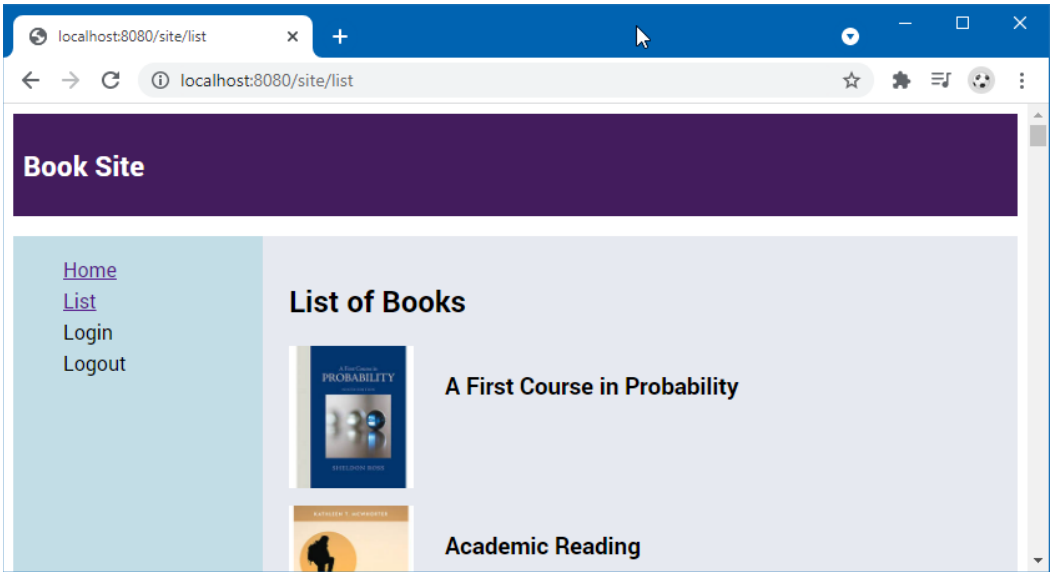

Figure 13.1 – The list page

5 Make a copy of `home.ejs` named `book.ejs`.

6 Change the pagecontent `<section>` of `book.ejs` as follows:

```
<section class="pagecontent">
   <article class="book">
      <div>
         <img src="/static/images/<%= book.isbn10 %>.jpg"  />
      </div>
      <div>
         <h1><%= book.title %></h1>
         <p>Year: <span><%= book.year %></span></p>
         <p>Publisher: <span><%= book.publisher %></span></p>
         <p>Pages: <span><%= book.production.pages %></span></p>
         <p>Categories: <span><%= book.category.main %>,
             <%= book.category.secondary %></span></p>
      </div>
   </article>
</section>
```

7 Add a new route by modifying `book-server.js` as follows.

```
app.get('/site/book/:isbn', (req, res) => {
    res.render('book.ejs', { book:
       controller.findByISBN10(req.params.isbn) } );
});
```

8 Re-run `book-server.js` and test by requesting `http://localhost:8080/site/list`. Click on any of the book cover images. The results should look similar to Figure 13b-2.

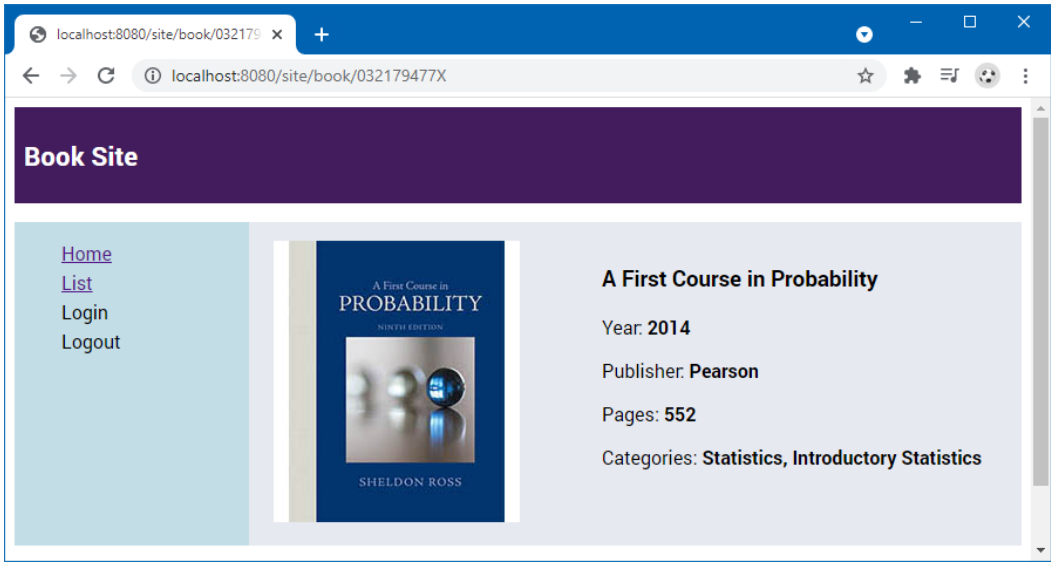

Figure 13.2 – The book page

User Authentication With Passport

In the remainder of this lab, you will implement a simple authentication system using the Passport middleware package. You will also use MongoDB as the data source for the user information (the rest of this lab assumes you have already worked through the exercises in Lab 14b).

Exercise 13b.4 — Installing Additional Packages

1 Install several 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.

2 Install even more packages via the following command.

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

This installs a mechanism for encrypting/decrypting bcrypt digests as well as the Mongoose ORM for working with MongoDB.

3 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.

4 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 2 of Exercise 14b.9). Be sure to replace `<PASSWORD>` with your password for the user.

5 Examine `mongoDataConnector.js` in the `scripts` folder. Notice that the mongoose.connect call uses the connection string from the previous step.

6 Examine the `users.json` file in the `data` folder. You should have imported this file into your MongoDB collection in Exercise 14b.5. If you haven't, do so now.

7 Test your MongoDB connection via:

```
node mongo-tester
```

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

8 Create a new file in `views` folder named `login.ejs`.

9 Copy and paste the content in `public/loginform.html` into `login.ejs`.

10 Add the following to `login.ejs`.

```
<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>
<% } %>
```

Exercise 13b.5 — Defining and Testing The Mongoose Model

1 Examine the `users.json` file in the `data` folder. You will be creating a Mongoose model for this schema.

2 In the `scripts` folder, create a new file named `User.js` and then add the following content.

```
const mongoose = require('mongoose');
const bcrypt = require('bcrypt');
// 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: String,
   apikey: String
 });
 module.exports = mongoose.model('User', userSchema, 'users');
```

3 Create a new file named `user-tester.js` and add the following.

```
require('dotenv').config();
const express = require('express');
const connector =
   require('./scripts/mongoDataConnector.js').connect();

const UserModel = require('./scripts/User.js');
UserModel.findOne({ email: "zpochet2@apple.com" }, (err, data) =>
{
   if (err) {
      console.log('user not found');
   } else {
      console.log('-- User found ---');
      console.log(data);
   }
});
```

This example uses the findOne() method provided by Mongoose.

4 Test via: `node user-tester.js`

This should display the user data for the specified email.

Exercise 13b.6 — Implementing the Passport Authentication

1 In the `scripts` 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('../User.js');

// maps the passport fields to the names of fields in database
const localOpt = {
  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 this email
    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 the bcrypt digest
    //stored in the database. If they match, return 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 13b.7 — Adding Password and Authentication Checks

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

```
// We'll use this later on to check if user 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
    // digest 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 `script` folder, create a new file named `helpers.js`. Add the following content.

```
// uses passport authentication 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.8 — Configuring the Routes

1 Add the following near the top of `book-server.js` as follows:

```
const express = require('express');
const session = require('express-session');
const cookieParser = require('cookie-parser');
const flash = require('express-flash');
const passport = require('passport');
const helper = require('./scripts/helpers.js');
require('./scripts/mongoDataConnector.js').connect();
```

2 Add the following to the middleware section:

```
// Express session
app.use(cookieParser('oreos'));
app.use(
    session({
      secret: process.env.SECRET,
      resave: true,
      saveUninitialized: true
    })
);
// Passport middleware
app.use(passport.initialize());
app.use(passport.session());

// use express flash, which will be used for passing messages
app.use(flash())

// set up the passport authentication
require('./scripts/auth.js');
```

In this example, we are using a session-based approach to maintaining our authentication status. Like the sessions in PHP, the Passport package uses cookies behind-the-scenes to implement server sessions.

3 Now you are ready to modify the routes so that they will only be available if the user is authenticated. Modify the first route as follows:

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

We are providing two handlers functions. The first (ensureAuthenticated function) is called, and if the user is authenticated, it will then pass execution to the next function. The second function passes the user object to the rendered page.

4 Modify the other routes as follows:

```
app.get('/site/list', helper.ensureAuthenticated, (req, res) => {
    res.render('list.ejs',  { books: controller.getAll() } );
});
app.get('/site/book/:isbn', helper.ensureAuthenticated,
         (req, res) => {
   res.render('book.ejs',
     { book: controller.findByISBN10(req.params.isbn) } );
});
```

5 Add handlers for login and logout routes as well:

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

6 You will need to modify the handlers for the api routes as well. Modify `api-router.js` as follows.

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

const handleAllBooks = (app, controller) => {
  app.get('/api/all', helper.ensureAuthenticated, (req,resp) => {
      ...
  } );
};
const handleISBN10 = (app, controller) => {
  app.get('/api/isbn10/:isbn10', helper.ensureAuthenticated,
            (req,resp) => {
      ...
   });
};
const handleTitle = (app, controller) => {
   app.get('/api/title/:substring', helper.ensureAuthenticated,
            (req,resp) => {
      ...
};
```

7 Finally, modify `home.ejs` as follows:

```
<section class="pagecontent">
  Welcome <%= user.details.firstname %>
          <%= user.details.lastname %>
</section>
```

8 You are ready to test by requesting: `http://localhost:8080/` or `http://localhost:8080/api/title/calc`

Both requests should redirect to the login form. Try logging in using `zpochet2@apple.com` as email and `mypassword` as the password. If successful, it should redirect to the home page. You should then be able to access any of the other links on the home page.

9 Use the logout option. Try logging in with incorrect credentials. The login form should display an error message.

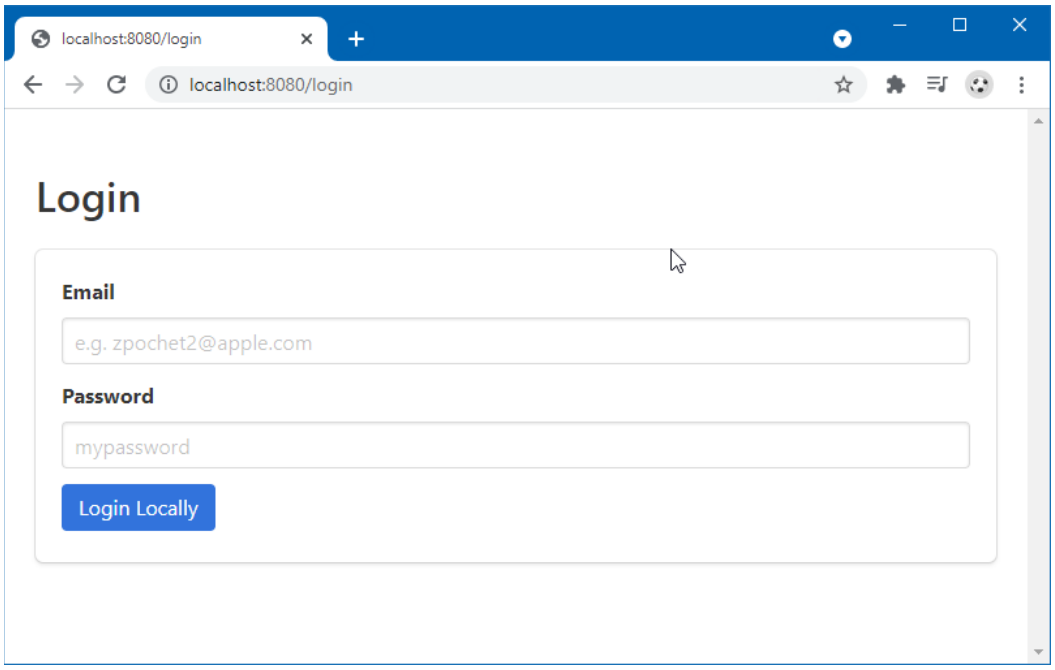

Figure 13.3 – The login page