LAB 16b

SECURITY: DEVELOPMENT

What You Will Learn

- How to implement HTTP authentication in PHP
- How to implement form authentication in PHP
- How to perform authentication and authorization in Node using Passport

Approximate Time

The exercises in this lab should take approximately 90 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 21, 2023

PREPARING DIRECTORIES

The starting `lab16b` folder has been provided for you (within the zip folder downloaded from Gumroad). This lab requires PHP, Node, and database access. If you are only interested in, say, the PHP exercises, then you can skip the exercises involving Node (or vice-versa).

Some (but not all) of the PHP exercises are duplicated in Node. If you wish to do both the PHP and the Node exercises in this lab, you can do all your work within the `htdocs` folder provided by XAMPP.

The PHP exercises will eventually make use of a database. These lab instructions assume you have access to SQLite in PHP. A SQL import script is also provided for those who wish to use MySQL instead. The Node exercises also make use of an SQLite database.

Besides `Lab12` (PHP) and `Lab13` (Node), this lab assumes you have completed the exercises in `Lab14a` (database in PHP) and/or `Lab14b` (database in Node) as well as the PHP and/or Node exercises in `Lab15` (State Management).

1 Create a folder named `lab16b` in your PHP development location on your development machine. Copy the contents of the provided zip into the `lab16b` folder.

2 If you wish to only do the Node exercises, then create a folder named `lab16b` in your Node development location on your development machine. Copy the contents of the provided `node` subfolder into the `lab16b` folder.

HTTP AUTHENTICATION

While HTTP authentication is quite rare on public sites, many intranet sites still make use of it. Let's begin with it so you can get some practice with examining HTTP headers.

Exercise 16b.1 — HTTP AUTHENTICATION

1 Examine `lab16b-ex01-tester.html` in a code editor.

This tester file is going to request a page from the server that will make a HTTP authentication request.

2 Add the following code to `lab16b-ex01.php`.

```
<?php
if ( !isset($_SERVER['PHP_AUTH_USER']) ) {
   header('WWW-Authenticate: Basic realm="Members Only"');
   header('HTTP/1.0 401 Unauthorized');
   exit;
} else {
   echo "<h1>Hello {$_SERVER['PHP_AUTH_USER']}</h1>";
   echo "<p>Your password was {$_SERVER['PHP_AUTH_PW']}</p>";
   echo "<a href='lab16b-ex01b.php'>Test another page</a>";
}
?>
```

3 If using Chrome, request `lab16b-ex01-tester.html` using an Incognito Tab. If using Firefox or Safari, make your request using a Private Browser Window.

Once you complete a HTTP Authentication, there is no way to logout. By using an Incognito Tab, this prevents this header from sticking around in future requests.

4 View the Network tab within the Dev Tools in your browser. You will need to re-request the page and then you will be able to view the request headers for your previous request (see Figure16b.2).

5 Keep the Dev Tools visible and then click on the link in `lab16b-ex01-tester.html`. If you saved the code in Step 2, the browser should display a dialog asking you to supply a user name and password for the Members Only realm (see Figure16b.1).

6 Supply a fictious user name and password to the dialog and click the Sign In/OK button.

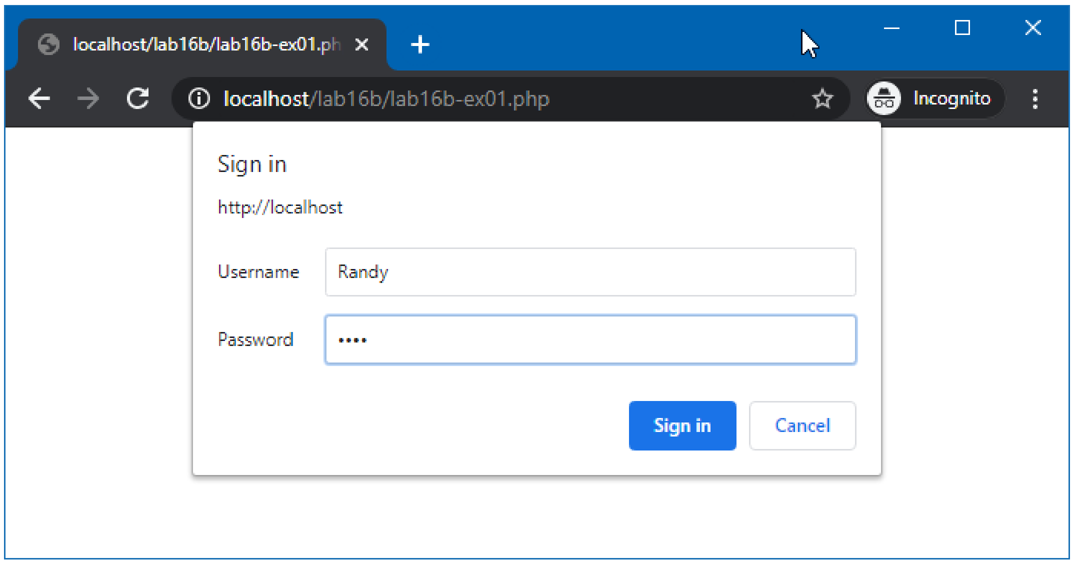
The page will echo back the username and password entered.

7 Examine the Request Headers in the Dev Tools. You should see an Authorization header that contains a Base64 encoded string of whatever you entered as your username and password.

8 Click on the “Test another page” link. Examine the Request Headers in the Dev Tools. Notice that the Authorization header is still the same. Now that you are “logged-in”, you will stay logged in, no matter which page you visit in this domain.

9 Copy the encoded string after the word “Basic” in your Dev Tools and then visit `https://www.base64decode.org/` or `https://base64.guru/converter/decode`. Paste in the encoded string and decode.

You should see your username and password (see Figure16b.3). This is not secure!



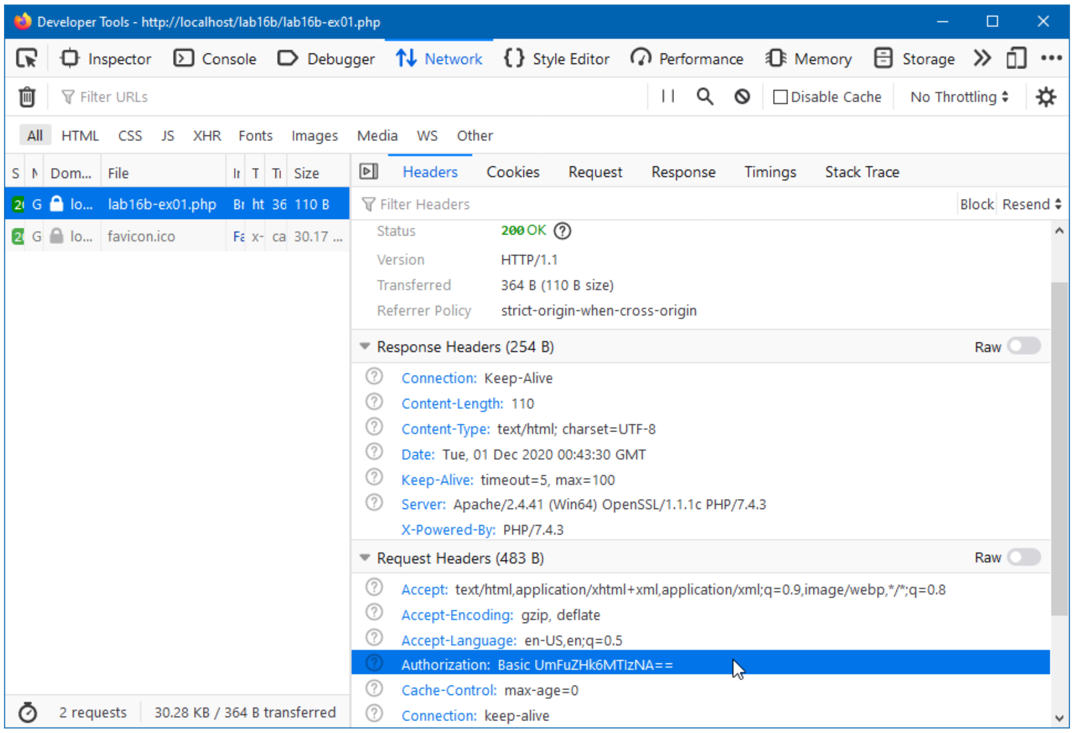
localhost/lab16b/lab16b-ex01.ph
localhost/lab16b/lab16b-ex01.php
Incognito
Sign in
Cancel

Sign in

http://localhost

Username	Randy
Password	••••

Figure 16b.1 – HTTP Authorization login form generated by the browser.



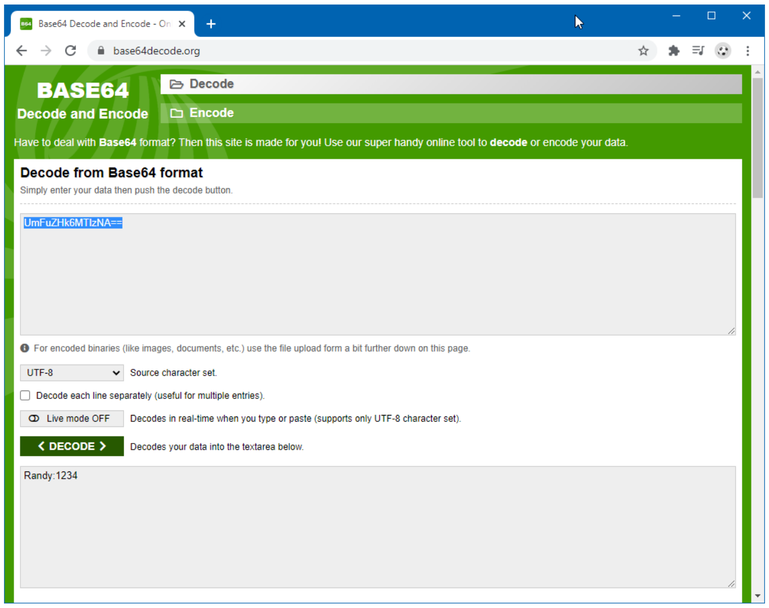
```
lab16b-ex01.php   110 B
favicon.ico       30.17 ...
2 requests   30.28 KB / 364 B transferred

Status            200 OK
Version           HTTP/1.1
Transferred       364 B (110 B size)
Referrer Policy   strict-origin-when-cross-origin

Response Headers (254 B)
Connection: Keep-Alive
Content-Length: 110
Content-Type: text/html; charset=UTF-8
Date: Tue, 01 Dec 2020 00:43:30 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.41 (Win64) OpenSSL/1.1.1c PHP/7.4.3
X-Powered-By: PHP/7.4.3

Request Headers (483 B)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Authorization: Basic UmFuZHk6MTIzNA==
Cache-Control: max-age=0
Connection: keep-alive
```

Figure 16b.2 – Examining the HTTP Request Headers



Have to deal with **Base64** format? Then this site is made for you! Use our super handy online tool to **decode** or encode your data.

Decode from Base64 format

Simply enter your data then push the decode button.

```
UmFuZHk6MTIzNA==
```

For encoded binaries (like images, documents, etc.) use the file upload form a bit further down on this page.

UTF-8 Source character set.

Decode each line separately (useful for multiple entries).

Live mode OFF Decodes in real-time when you type or paste (supports only UTF-8 character set).

< DECODE > Decodes your data into the textarea below.

```
Randy:1234
```

Figure 16b.3 – Decoding a Base64 string

Form Authentication with PHP

Due to all of its drawbacks, HTTP authentication is rarely used for public-facing sites. Form authentication is used instead. The next several exercises all require a use of a database on the server-side, so if you haven't completed Lab14a, you should complete Lab14a first.

Exercise 16b.2 — Using Bcrypt in PHP

1 If you have completed Lab14a and still have the art, travel, or stocks databases available you have the necessary database for this lab.

If you no longer have those databases available (or you prefer to do this lab's exercises with a dedicated database), you have been provided with a variety of SQL import files that you can use with MySQL for the next several exercises. You have also been provided with a SQLite file if you prefer to use that instead.

Using either phpMyAdmin, the MySQL command window, or SQLiteStudio, examine the structure of the `userslogin` table in the `Travel` database, or the `customerLogon` table in the Art database, the `users` table in the `Stock` database, or the `users` table in the `security` SQLite database (all four have almost the same structure).

2 Try running one of the following queries from the supplied databases:

```
// art database
SELECT * FROM customerlogon WHERE Username = 'hholy@gmail.com'

// security database
SELECT * FROM users WHERE email = 'hholy@gmail.com'

// travel database
SELECT * FROM userslogin WHERE UserName = 'hholy@gmail.com'

// stocks database
SELECT * FROM users WHERE email = 'hholy@gmail.com'
```

Why the different field names? These databases have been created over several years for different courses and I never noticed the small differences between them!

I do hope, however, that you notice the content of the password field in the different tables. The main password field contains a digest from the bcrypt hash function. We will use the Salt and Password_sha256 fields in a future exercise.

3 Examine `art-config.inc.php` in a text editor. Notice that it provides connection details for both MySQL and SQLite. Comment or uncomment out the relevant connection string for your development environment. If you are going to use a different database, be sure to change the `DBNAME` and `DBCONNSTRING` variables as needed.

4 Examine `lab16b-ex02.php` first in a browser, then in your text editor.

With this form, you are going to authenticate the login using the bcrypt hashing function. For each record, the plain text password is actually ***mypassword****.*

5 Add the following code to the top of `lab16b-ex02.php`.

```
<?php
include 'art-config.inc.php';
$msg = "Demonstrates how to use form authentication";
try {

} catch (Exception $e) { die( $e->getMessage() ); }
?>
```

6 Modify the markup as follows:

```
<body>
   <section class="container">
      <div class="formData">
         <h2>Login</h2>
         <p><?= $msg ?></p>
         <form method="post" action="lab16b-ex02.php">
```

Form error messages are going to be contained with the $msg variable. Notice also that the form action is the same file as the form page itself.

7 Add the following code to the `try..catch` in `lab16b-ex02.php`.

```
try {
  if ( isset($_POST['email']) && isset($_POST['pass']) ) {
    // 1. First see if this email is in the database
    $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
    $pdo->setAttribute(PDO::ATTR_ERRMODE,
                       PDO::ERRMODE_EXCEPTION);
    // run the query
    $sql = "select * from customerlogon where UserName=?";
    $statement = $pdo->prepare($sql);
    $statement->bindValue(1,  $_POST["email"] );
    $statement->execute();
     // retrieve the data and close connection
    $data = $statement->fetch(PDO::FETCH_ASSOC);
    $pdo = null;
     // if $data is empty then supplied email was not found
    if ( isset($data['Pass']) ) {
       // 2. Does this password match the digest saved in the
       //    Password field in database for this username?
      if ( password_verify($_POST['pass'], $data['Pass']) ) {
         // 3. We have a match, log-in user.
         //    For now, simply redirect
         header("Location: login-success.php");
         exit();
      } else {
         $msg = "Password incorrect";
      }
    } else {
       $msg = "Email not found";
    }
  }
} catch (Exception $e) { die( $e->getMessage() ); }
```

8 Test. Use `hholy@gmail.com` as the email and `mypassword` as the password. It should redirect to `login-success.php` file. Try experimenting with an incorrect password and an incorrect email.

We can simplify the code by encapsulating the database access using a gateway class and database helper from Chapter 14.

Exercise 16b.3 — Encapsulating Data Access

1 Examine `lab16b-db-classes.inc.php`. It already has a `CustomerLogonDB` class defined.

2 Add the following function to `lab16b-ex03.php`.

```
function isLoginDataPresent() {
   if ( isset($_POST['email']) && isset($_POST['pass']) )
      return true;
   else
      return false;
}
```

3 Add the following code to `lab16b-ex03.php`.

```
$msg = "Demonstrates how to use form authentication";
if ( isLoginDataPresent() ) {
  try {
     // 1. First see if this email exists
     $connection =
         DatabaseHelper::createConnection($connectionDetails);
     $gate = new CustomerLogonDB ($connection);
     $data = $gate->getByUserName($_POST["email"]);
     $connection = null;
     // if $data is empty then supplied email was not found
     if ( isset($data['Pass']) ) {
       // 2. Does this password match the digest saved in the
       //    Password field in database for this username?
       if ( password_verify($_POST['pass'], $data['Pass']) ) {
          // 3. we have a match, log-in user
          header("Location: login-success.php");
          exit();
       } else {
          $msg = "Password incorrect";
       }
    } else {
         $msg = "Email not found";
    }
  } catch (Exception $e) { die( $e->getMessage() ); }
}
```

4 Test.

This should still work the same but the code is simpler.

As you learned in Lab15, sessions are a mechanism for maintaining state on the server about a user session. One of the most common things stored in session state is login status, which is covered in the next exercise.

Exercise 16b.4 — Integrating Sessions

1 This exercise will work with the same files from the previous exercise. Add the following to the top of `lab16b-ex03.php`.

```
session_start();
$msg = "Demonstrates how to use form authentication";
```

2 Modify the login code as follows:

```
if ( password_verify($_POST['pass'], $data['Pass']) ) {
   // 3. we have a match, log-in user
   $_SESSION['user'] = $data['CustomerID'];
   header("Location: login-success.php");
```

3 Add the following code to `login-success.php`.

```
session_start();
// if user id not in session then redirect back to login screen
if ( ! isset($_SESSION['user']) ) {
   header('Location: lab16b-ex03.php');
   exit();
}
…
<h1>Login succeeded! UserID = <?= $_SESSION['user'] ?></h1>
```

Storing a bcrypt digest is currently a best practice in terms of credential storage. However, there are many legacy web applications that instead use an explicit salt field and a fast hashing algorithm such as SHA256. The next exercise illustrates how one would implement a login using this older approach. *Do remember that passwords in this older approach are vulnerable to brute force decoding if the table is leaked.*

Exercise 16b.5 — Salting a Password

1 Examine `lab16b-ex05.php`. It already has much of the code from the previous two exercises in it.

2 Add the following code then test.

```
// add your code here
$extendedPassword = $_POST['pass'] . $data['Salt'];
$calculatedDigest = hash("sha256", $extendedPassword );
$digestInDatabase = $data['Password_sha256'];
// if these match then password is correct so can log in user
if ( $calculatedDigest == $digestInDatabase ) {
   // 3. we have a match, log-in user
   $_SESSION['user'] = $data['CustomerID'];
   header("Location: login-success.php");
   exit();
} else {
   $msg = "Password incorrect";
}
```

Notice that you first create an extended password as input into the hashing algorithm that consists of the password entered in the form with the value from the Salt field for this user record.

In the next exercise, you will see how to save updated user credentials to the database.

Exercise 16b.6 — Registering a User

1 Examine `lab16b-ex06.php`. It has a form and some of the code already defined.

2 Add the following code.

```
$msg = "Demonstrates how to generate and store a digest";
if ( isLoginDataPresent() ) {
   if ( $_POST['pass1'] != $_POST['pass2'] )
      $msg = "Passwords must match";
   else {
      try {
         // 1. first see if this email is in the database
         $connection = DatabaseHelper::
                          createConnection($connectionDetails);
         $gate = new CustomerLogonDB($connection);
         $data = $gate->getByUserName($_POST["email"]);
         $connection = null;

        // if $data is empty then supplied email was not found
         if ( isset($data['Pass']) ) {

           // calculate the bcrypt digest using cost = 12
            $digestBcrypt = password_hash($_POST['pass1'],
                                 PASSWORD_BCRYPT, ['cost' => 12]);
           // also let's show how you would do the same with
           // salt + SHA256 approach
            $extendedPassword = $_POST['pass1'] . $data['Salt'];
            $digestSHA256 = hash("sha256", $extendedPassword );

           // save the updated passwords then redirect
            $gate->updatePassword($data['CustomerID'],
                       $digestBcrypt, $digestSHA256);
            header("Location: password-changed.php");
            exit();
         } else {
            $msg = "Email not found";
         }
      } catch (Exception $e) { die( $e->getMessage() ); }
   }
}
```

3 Add the following method to `CustomerLogonDB` in `lab16b-db-classes.inc.php`.

```
public function updatePassword($id, $pass, $shaPass) {
    $sql = "UPDATE customerlogon SET Pass=?, Password_sha256=?
              WHERE CustomerID=?";
    $statement = DatabaseHelper::runQuery($this->pdo, $sql,
                      Array($pass, $shaPass, $id));
}
```

4 Test `lab16b-ex06.php` in browser. Be sure to verify changed password works when logging in.

USER AUTHENTICATION IN NODE 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.7 — INSTALLING ADDITIONAL PACKAGES

1 Install several packages via the following commands.

```
npm init
npm install express express-session cookie-parser
npm install mongoose dotenv ejs
npm install bcrypt passport passport-local express-flash
```

This installs a mechanism for encrypting/decrypting bcrypt digests as well as the Mongoose ORM for working with MongoDB. This installs the relevant passport packages. We will use passport as our authentication library.

2 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.

3 Examine `mongoDataConnector.js` in the `scripts` folder. In the `dbName` field, specify the name of your database.

Notice that the mongoose.connect call uses the connection string from the previous step.

4 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.

5 Test your MongoDB connection via:

```
node mongo-tester
```

If successful, you should see message "connected to mongo". With the Free Tier of MongoDB Atlas, your database will become unavailable after a period of non-usage. If you are getting an error message here, verify your cluster is available.

6 Examine `login.ejs` in `views` folder, and then add the following to it.

```
<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.8 — Defining and Testing The Mongoose Model

1 Examine the `users.json` file again 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 app = express();
const connector =
   require('./scripts/mongoDataConnector.js').connect();

const UserModel = require('./scripts/User.js');

app.get("/", (req, resp) => {
   UserModel.findOne({email: "zpochet2@apple.com" })
      .then(data => {
         console.log('-- User found ---');
         resp.json(data);
       })
      .catch(err => {
         console.log('user not found');
      });
});
const port = process.env.port || 8080;
app.listen(port, () => {
  console.log("Server running at port= " + port);
});
```

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

4 Test by running `node user-tester` and then requesting `http://localhost:8080/`.

This should display the user data for the specified email. With the Free Tier of Mongodb Atlas, your database will become unavailable after a period of non-usage. If you are getting an error message here, verify your cluster is available and that you can view the data in the user collection.

Exercise 13b.9 — Implementing 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 })
    .then((user) => done(null, user));
});
```

Exercise 13b.10 — 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;
}
module.exports = mongoose.model("User", userSchema, "users");
```

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 resources');
   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.11 — 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 Test by running `node book-server` and then 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.

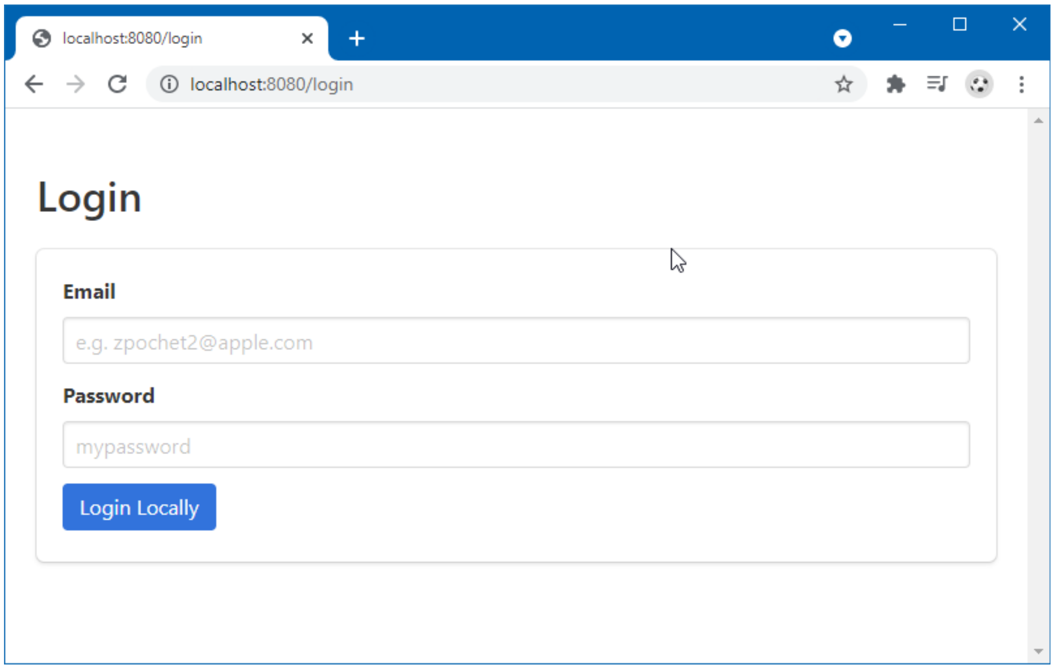


Figure 13.4 – The login page