



-  an hour
-  Completed

# Implementing Session-Based Authentication

In this reading, you'll learn how to implement session-based authentication in an Express application.

By the end of this article, you should be able to:

- Understand what authentication and authorization are and how they differ from each other;
- Understand how to support user self-registration and login within a web application; and
- Add session-based authentication to an Express website including user flows to support self-registration and login.

## Overview of authentication and authorization

---

So far, all of the users of your applications have been anonymous. While the server can use a feature like sessions to determine if a request is from a client that's been served before, the server still doesn't know the identity of that user.

If an application can't identify its users, it doesn't have a way to associate data with specific users. The ability to know whose data belongs to whom is an extremely important feature.

Think about popular web applications, such as Facebook. Being able to identify the current user is necessary so Facebook can determine who your friends are, what posts to display to you, and if you post, who to associate the post with.

# What is authentication?

Authentication is the process of identifying a user. To authenticate a user, a `key` and a `secret` is required.

- The `key` is typically a username or an email address.
- The `secret` is typically a password.

The user's `key` and `secret` fields are stored in the application's database as a user account.

A user's account can also contain other information about the user. For example, an account can contain personal information such as a birth date, a mobile phone number, or a physical address.

Application preferences are also stored in a user account. These preferences can include how many items to include in lists, the designation of a default home page, or color scheme (light vs dark mode).

## Authentication process

When a user authenticates, they provide their username and password via an HTML form. The form then posts to the server. The login route handler attempts to retrieve the user from the database using their username. If a user is found, then the user account's *hashed* password is checked against the provided password. Passwords are stored in the database as encrypted *hashes*.

The user's provided password needs to be hashed first before it can be compared to the password hash from the database. If the password hashes match, then the user is logged in. With session-based authentication, the user's ID is stored in the session. Subsequent requests to the application can then check if the session contains a user ID. If a user ID is available, then the user is logged in!

The application can then retrieve the user's account information and make it available to other middleware, the route handler, and the view.

## What is authorization?

Authorization is the process of determining if the currently logged in user has access to an application's data or features. Sometimes, data is associated directly with a user. For example, a table in the database for storing posts or comments would include

a `userId` column. The `userId` column would have the primary key `id` value from the `Users` table.

Relating data to specific users allows the application to retrieve just the user's own records. Sometimes data should only be accessed by users that are assigned a specific user function or role.

For example, a user might belong to an "Admin" role. Any user in the "Admin" role has permissions to add, update, or even delete other application users. "Admin" role users may also have additional access to certain application features. Don't let that power go to your head!

We'll explore how role-based access control works in a future article.

## Supporting user self-registration

---

Before a user can login to an application or website, the application needs to know who that user is... they need to register!

## Getting the starter project

Clone the starter project:

```
git clone https://github.com/appacademy-starters/express-reading-list-with-auth-starter.git
```

Install the project's dependencies:

```
npm install
```

Add an `.env` file containing the environment variables from the `.env.example` file:

```
PORT=8080
DB_USERNAME=reading_list_app
DB_PASSWORD=«the reading_list_app user password»
DB_DATABASE=reading_list
DB_HOST=localhost
```

Create the database and database user (if needed). Open `psql` by running the command `psql` (to use the currently logged in user) or `psql -U «super user username»` to specify the username of the super user to use. Then execute the following SQL statements:

```
create database reading_list;
```

```
create user reading_list_app with encrypted password '«a strong
password for the reading_list_app user»';
grant all privileges on database reading_list to
reading_list_app;
```

Make sure that the `.env` file contains the correct database user password!

Start and test the application:

```
npm start
```

## Creating the User model

The User model should include the following properties:

- `emailAddress` - A string representing the user's email address;
- `firstName` - A string representing the user's first name;
- `lastName` - A string representing the user's last name; and
- `hashedPassword` - A string representing the user's hashed password.

From the terminal, run the following command to use the Sequelize CLI to generate the User model:

```
npx sequelize model:generate --name User --attributes
"emailAddress:string, firstName:string, lastName:string,
hashedPassword:string"
```

If the command succeeds, you'll see the following output in the console:

```
New model was created at [path to the project
folder]/db/models/user.js .
New migration was created at [path to the project
folder]/db/migrations/20200410231702-User.js
```

This confirms that two files were generated: a file for the User model and a file for a database migration to add the Users table to the database.

Open the `./db/models/user.js` file and update the User model's attribute data types and nullability:

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define('User', {
    emailAddress: {
      type: DataTypes.STRING(255),
      allowNull: false,
      unique: true
    },
    firstName: {
      type: DataTypes.STRING(50),
      allowNull: false
```

```

    },
    lastName: {
      type: DataTypes.STRING(50),
      allowNull: false
    },
    hashedPassword: {
      type: DataTypes.STRING.BINARY,
      allowNull: false
    }
  }, {});
User.associate = function(models) {
  // associations can be defined here
};
return User;
};

```

Take a moment to notice that the `emailAddress` attribute is configured to be unique. This means that there is a *unique constraint* on that column in the database. The *unique constraint* ensures that each user in the database will have a unique `emailAddress`. Without the assurance of knowing that each user has a unique `emailAddress`, you wouldn't be able to reliably identify a user by their email address.

The Sequelize `STRING` data type has an available property of `BINARY`. Setting an attribute to have a datatype of `STRING.BINARY` means that the string is stored as raw-byte data. According to the PostgreSQL documentation, a [binary string](#) is a sequence of octets (or bytes).

Make sure to also set the `hashedPassword` attribute with a datatype of `STRING.BINARY` in the migration file:

```

'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Users', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      emailAddress: {
        type: Sequelize.STRING(255),
        allowNull: false,
        unique: true
      },
      firstName: {
        type: Sequelize.STRING(50),
        allowNull: false
      },
      lastName: {
        type: Sequelize.STRING(50),
        allowNull: false
      },

```

```

    hashedPassword: {
      type: Sequelize.STRING.BINARY,
      allowNull: false
    },
    createdAt: {
      allowNull: false,
      type: Sequelize.DATE
    },
    updatedAt: {
      allowNull: false,
      type: Sequelize.DATE
    }
  }
});
},
down: (queryInterface, Sequelize) => {
  return queryInterface.dropTable('Users');
}
};

```

Then apply the migration:

```
npx dotenv sequelize db:migrate
```

In the console, you should see something similar to the following output:

```

Loaded configuration file "config/database.js".
Using environment "development".
== 20200410231702-create-user: migrating =====
== 20200410231702-create-user: migrated (0.028s)

```

To confirm the creation of the `Users` table, you can run the following command from within `psql`:

```
\d "Users"
```

*Be sure that you're connected to the `reading_list` database in `psql`. If you are, the cursor should read `reading_list=#`. If you're not connected to the correct database, you can run the command `\c reading_list` to connect to the `reading_list` database.*

After running the `\d "Users"` command, you should see the following output within `psql`:

Column		Type	Table "public.Users"		
	Default		Collation	Nullable	
-----+-----+-----+-----+-----+-----					
id		integer		not null	
nextval('"Users_id_seq"::regclass)					
emailAddress		character varying(255)		not null	
firstName		character varying(50)		not null	
lastName		character varying(50)		not null	
hashedPassword		bytea		not null	

```

  createdAt      | timestamp with time zone |          | not null |
  updatedAt      | timestamp with time zone |          | not null |
Indexes:
  "Users_pkey" PRIMARY KEY, btree (id)
  "Users_emailAddress_key" UNIQUE CONSTRAINT, btree
("emailAddress")

```

## Adding the user registration form

Now you'll want to create a form that collects the required account information from the user.

To start, now that the application will have two resources, "books" and "users", let's do a little refactoring! Create a `routes` folder in the root of the project and then rename the `routes.js` file as `book.js` and move the file into the `routes` folder.

Update the `routes` module reference in the `app.js` file:

```

const bookRoutes = require('./routes/book');

// Code removed for brevity.

app.use(bookRoutes);

```

Then move the `csrfProtection` and `asyncHandler` variable declarations to a new `./routes/utils` module:

```

// ./routes/utils.js

const csrf = require('csurf');

const csrfProtection = csrf({ cookie: true });

const asyncHandler = (handler) => (req, res, next) =>
  handler(req, res, next).catch(next);

module.exports = {
  csrfProtection,
  asyncHandler,
};

```

And update the `./routes/book.js` file to import those items from the `utils` module:

```

const { csrfProtection, asyncHandler } = require('./utils');

```

Add a module named `user` to the `routes` folder containing the following code to define the routes for the "Register" page:

```

const express = require('express');
const { check, validationResult } = require('express-validator');

const db = require('../db/models');

```

```

const { csrfProtection, asyncHandler } = require('./utils');

const router = express.Router();

router.get('/user/register', csrfProtection, (req, res) => {
  const user = db.User.build();
  res.render('user-register', {
    title: 'Register',
    user,
    csrfToken: req.csrfToken(),
  });
});

const userValidators = [
  // TODO Define the user validators.
];

router.post('/user/register', csrfProtection, userValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      firstName,
      lastName,
      password,
    } = req.body;

    const user = db.User.build({
      emailAddress,
      firstName,
      lastName,
    });

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      await user.save();
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) =>
error.msg);
      res.render('user-register', {
        title: 'Register',
        user,
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  }));

```



```
module.exports = router;
```

Rename the `textField` mixin to `field` in the `utils.pug` template file:

```
mixin field(labelText, fieldName, fieldValue, fieldType =  
'text')  
  div(class='form-group')  
    label(for=fieldName)= labelText  
    input(type=fieldType id=fieldName name=fieldName  
value=fieldValue class='form-control')
```

Notice that a `fieldType` parameter was added that defaults to a value of `text`. Adding this parameter will give you a way to add a password field to the register form.

Then add the `user-register.pug` template file to the `views` folder containing the following code:

```
//- ./views/user-register.pug  
  
extends layout.pug  
  
include utils.pug  
  
block content  
  +validationErrorSummary(errors)  
  form(action='/user/register' method='post')  
    input(type='hidden' name='_csrf' value=csrfToken)  
    +field('First Name', 'firstName', user.firstName)  
    +field('Last Name', 'lastName', user.lastName)  
    +field('Email Address', 'emailAddress', user.emailAddress)  
    +field('Password', 'password', user.password, 'password')  
    +field('Confirm Password', 'confirmPassword', '',  
'password')  
    div(class='py-4')  
      button(type='submit' class='btn btn-primary') Register  
      a(href='/' class='btn btn-warning ml-2') Cancel
```

Notice that there are two "password" fields: "Password" and "Confirm Password". When using the `input` element with a `type` attribute of `password`, the field will replace typed characters with bullets to hide what has been typed into the field. This is a great protection from prying eyes, but you need a way for users to be able to confirm the password that they're providing. Making them type their password twice is doing exactly that.

**Note:** After renaming the `textField` mixin to `field`, be sure to update the `book-form-fields.pug` template to call the mixin using the new name!

## Validating the user registration form

Implement the following validation rules:

- `firstName`

- Not null or empty
- Not longer than 50 characters
- lastName
  - Not null or empty
  - Not longer than 50 characters
- emailAddress
  - Not null or empty
  - Not longer than 255 characters
  - Is a valid email address
- password
  - Not null or empty
  - Not longer than 50 characters
- confirmPassword
  - Not null or empty
  - Not longer than 50 characters

Here's what the initial pass at setting up the validators looks like using the `express-validator` library:

```
const userValidators = [
  check('firstName')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for First Name')
    .isLength({ max: 50 })
    .withMessage('First Name must not be more than 50
characters long'),
  check('lastName')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Last Name')
    .isLength({ max: 50 })
    .withMessage('Last Name must not be more than 50 characters
long'),
  check('emailAddress')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Email Address')
    .isLength({ max: 255 })
    .withMessage('Email Address must not be more than 255
characters long')
    .isEmail()
    .withMessage('Email Address is not a valid email'),
  check('password')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Password')
    .isLength({ max: 50 })
    .withMessage('Password must not be more than 50 characters
long'),
  check('confirmPassword')
    .exists({ checkFalsy: true })
```

```

    .withMessage('Please provide a value for Confirm Password')
    .isLength({ max: 50 })
    .withMessage('Confirm Password must not be more than 50
characters long'),
];

```

You can use a regular expression to enforce password complexity:

```

check('password')
  .exists({ checkFalsy: true })
  .withMessage('Please provide a value for Password')
  .isLength({ max: 50 })
  .withMessage('Password must not be more than 50 characters
long')
  .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*])/, 'g')
  .withMessage('Password must contain at least 1 lowercase
letter, uppercase letter, number, and special character (i.e.
"!@#$%^&*")')
  ^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*])

```

This article from [The Polyglot Developer](#) breaks down the RegEx that tests the password strength in the validation above. To recap:

- The hat operator ^ is used to start matching at the beginning of the password.
- The expression (?!.\*[a-z]) is used to check that the password contains at least one lowercase character.
- The expression (?!.\*[A-Z]) is used to check that the password contains at least one uppercase character.
- The expression (?!.\*[0-9]) is used to check that the password contains at least one numeric character.
- The expression (?!.\*[!@#\$%^&\*]) is used to check that the password contains at least one special character.

The validation below checks that the two passwords should match:

```

check('confirmPassword')
  .exists({ checkFalsy: true })
  .withMessage('Please provide a value for Confirm Password')
  .isLength({ max: 50 })
  .withMessage('Confirm Password must not be more than 50
characters long')
  .custom((value, { req }) => {
    if (value !== req.body.password) {
      throw new Error('Confirm Password does not match
Password');
    }
    return true;
  })

```

You can use another custom validator to check if the provided email address is already in use by another account:

```
check('emailAddress')
  .exists({ checkFalsy: true })
  .withMessage('Please provide a value for Email Address')
  .isLength({ max: 255 })
  .withMessage('Email Address must not be more than 255
characters long')
  .isEmail()
  .withMessage('Email Address is not a valid email'),
  .custom((value) => {
    return db.User.findOne({ where: { emailAddress: value } })
      .then((user) => {
        if (user) {
          return Promise.reject('The provided Email Address is
already in use by another account');
        }
      });
  }),
```

## Hashing user passwords

To keep your user's personal information as secure as possible, you need to avoid storing user passwords in clear text. You can do this by encrypting the password with BCrypt, a password hashing function.

Install the `bcryptjs` npm package:

```
npm install bcryptjs
```

Update the POST `/user/register` route handler to use `bcrypt` to hash the user's password:

```
const bcrypt = require('bcryptjs');

// Code removed for brevity.

router.post('/user/register', csrfProtection, userValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      firstName,
      lastName,
      password,
    } = req.body;

    const user = db.User.build({
```

```

    emailAddress,
    firstName,
    lastName,
  });

  const validatorErrors = validationResult(req);

  if (validatorErrors.isEmpty()) {
    const hashedPassword = await bcrypt.hash(password, 10);
    user.hashedPassword = hashedPassword;
    await user.save();
    res.redirect('/');
  } else {
    const errors = validatorErrors.array().map((error) =>
error.msg);
    res.render('user-register', {
      title: 'Register',
      user,
      errors,
      csrfToken: req.csrfToken(),
    });
  }
}
}));

```

Notice that the asynchronous method `bcrypt.hash()` is called to hash the `password` variable. The hashed value returned from the method call is used to set the `user.hashedPassword` property.

## Adding the user routes to the `app` module

In the `app` module, add the user routes:

```

const bookRoutes = require('./routes/book');
const userRoutes = require('./routes/user');

// Code removed for brevity.

app.use(bookRoutes);
app.use(userRoutes);

```

## Testing user registration

Run the application (`npm start`) and test the `/user/register` route by filling out and submitting the form to create a new user.

Using `psql`, view the user in the database by running the following `SELECT SQL` statement:

```
select * from "Users";
```

You should see a user with a hashed password like in this example:

```
id | emailAddress | firstName | lastName |
hashedPassword
| createdAt | updatedAt
-----+-----+-----+-----+-----
1 | james@smashdev.com | James | Churchill |
\x24326124303824446c32684c546e676f49524f322e2e7335704d42697558554955
617730325264796d58534e544d3552542e6d686b365a46336e724f | 2020-04-10
18:54:07.635-07 | 2020-04-10 18:54:07.635-07
(1 row)
```

## Supporting user login

Now that you have a way to support user registration, you need a way to allow existing users to log in using their email address and password.

## Adding the user login form

Start by adding the routes and validators for the "Login" page to the `./routes/user` module just below the existing routes for the "Register" page:

```
router.get('/user/login', csrfProtection, (req, res) => {
  res.render('user-login', {
    title: 'Login',
    csrfToken: req.csrfToken(),
  });
});

const loginValidators = [
  check('emailAddress')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Email Address'),
  check('password')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a value for Password'),
];
```

```

router.post('/user/login', csrfProtection, loginValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      password,
    } = req.body;

    let errors = [];
    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      // TODO Attempt to login the user.
    } else {
      errors = validatorErrors.array().map((error) => error.msg);
    }

    res.render('user-login', {
      title: 'Login',
      emailAddress,
      errors,
      csrfToken: req.csrfToken(),
    });
  }));

```

Notice the slightly different approach of declaring the `errors` array outside of the `else` block. Using this approach will allow you to manually add an error message to the `errors` array if the user login process fails (you'll implement this process in just a bit). Then add the `user-login.pug` template file to the `views` folder containing the following code:

```

// - ./views/user-login.pug

extends layout.pug

include utils.pug

block content
  +validationErrorSummary(errors)
  form(action='/user/login' method='post')
    input(type='hidden' name='_csrf' value=csrfToken)
    +field('Email Address', 'emailAddress', emailAddress)
    +field('Password', 'password', null, 'password')
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Login
      a(href='/' class='btn btn-warning ml-2') Cancel

```

# Implementing the user login process

To implement the user login process, the following steps need to be followed:

1. Attempt to retrieve the user from the database using the supplied email address.

```
// Attempt to get the user by their email address.  
const user = await db.User.findOne({ where: { emailAddress } });
```

2. If a user was found in the database, then use the `bcrypt.compare()` method to compare the supplied password to the user's hashed password.

```
// Attempt to get the user by their email address.  
const user = await db.User.findOne({ where: { emailAddress } });  
  
if (user !== null) {  
  // If the user exists then compare their password  
  // to the provided password.  
  const passwordMatch = await bcrypt.compare(password,  
    user hashedPassword.toString());  
}
```

3. If the hashed passwords match (i.e. the `bcrypt.compare()` method returns `true`), then login the user and redirect them to the default route (i.e. `/`).

```
// Attempt to get the user by their email address.  
const user = await db.User.findOne({ where: { emailAddress } });  
  
if (user !== null) {  
  // If the user exists then compare their password  
  // to the provided password.  
  const passwordMatch = await bcrypt.compare(password,  
    user hashedPassword.toString());  
  
  if (passwordMatch) {  
    // If the password hashes match, then login the user  
    // and redirect them to the default route.  
    // TODO Login the user.  
    return res.redirect('/');  
  }  
}
```

**Note:** We'll implement a method to "login" the user in just a bit. For now, add a `TODO` comment as a placeholder for the actual method call.

4. If a user wasn't found in the database or the hashed passwords don't match, then add a validation error message and render the `user-login` view to let the user know that the login process failed.

```
// Attempt to get the user by their email address.  
const user = await db.User.findOne({ where: { emailAddress } });  
  
if (user !== null) {  
  // If the user exists then compare their password  
  // to the provided password.
```



```

    const passwordMatch = await bcrypt.compare(password,
user.hashPassword.toString());

    if (passwordMatch) {
        // If the password hashes match, then login the user
        // and redirect them to the default route.
        // TODO Login the user.
        return res.redirect('/');
    }
}

// Otherwise display an error message to the user.
errors.push('Login failed for the provided email address and
password');

```

Notice that you're not letting the user know if the supplied email address or password is to blame for the failed login attempt. This is intentional! Not providing this information, while potentially frustrating to end users, makes it more difficult for hackers to guess at email address and password combinations.

Here's the completed POST /user/login route:

```

router.post('/user/login', csrfProtection, loginValidators,
    asyncHandler(async (req, res) => {
        const {
            emailAddress,
            password,
        } = req.body;

        let errors = [];
        const validatorErrors = validationResult(req);

        if (validatorErrors.isEmpty()) {
            // Attempt to get the user by their email address.
            const user = await db.User.findOne({ where: { emailAddress
} }));

            if (user !== null) {
                // If the user exists then compare their password
                // to the provided password.
                const passwordMatch = await bcrypt.compare(password,
user.hashPassword.toString());

                if (passwordMatch) {
                    // If the password hashes match, then login the user
                    // and redirect them to the default route.
                    // TODO Login the user.
                    return res.redirect('/');
                }
            }
        }
    })

```

```

        // Otherwise display an error message to the user.
        errors.push('Login failed for the provided email address
and password');
    } else {
        errors = validatorErrors.array().map((error) => error.msg);
    }

    res.render('user-login', {
        title: 'Login',
        emailAddress,
        errors,
        csrfToken: req.csrfToken(),
    });
}));

```

## Improving the user navigation

One small change to the `user-register` and `user-login` views before you test the user login form. As a convenience for the user, add links below the user registration and login forms that allow the user to easily navigate between the "Register" and "Login" pages:

```

// - ./views/user-register.pug

extends layout.pug

include utils.pug

block content
    +validationErrorSummary(errors)
    form(action='/user/register' method='post')
        input(type='hidden' name='_csrf' value=csrfToken)
        +field('First Name', 'firstName', user.firstName)
        +field('Last Name', 'lastName', user.lastName)
        +field('Email Address', 'emailAddress', user.emailAddress)
        +field('Password', 'password', user.password, 'password')
        +field('Confirm Password', 'confirmPassword', '',
'password')
        div(class='py-4')
            button(type='submit' class='btn btn-primary') Register
            a(href='/' class='btn btn-warning ml-2') Cancel
        div
            p: a(href='/user/login') Already have an account?
// - ./views/user-login.pug

extends layout.pug

```

```

include utils.pug

block content
  +validationErrorSummary(errors)
  form(action='/user/login' method='post')
    input(type='hidden' name='_csrf' value=csrfToken)
    +field('Email Address', 'emailAddress', emailAddress)
    +field('Password', 'password', null, 'password')
    div(class='py-4')
      button(type='submit' class='btn btn-primary') Login
      a(href='/' class='btn btn-warning ml-2') Cancel
    div
      p: a(href='/user/register') Don't have an account?

```

## Testing user login

Run the application (if it's not already running) and browse to the `/user/login` route. Test the user login form by completing the following actions:

- Submit the form with no values.
  - You should see two validation messages asking you to provide values.
- Submit the form with an email address that isn't associated with a user record in the database and a password (doesn't matter if the password is correct or not).
  - You should see a validation message letting you know that the login attempt failed.
- Submit the form with an email address that's associated with a user record in the database **but with an incorrect password**.
  - You should see a validation message letting you know that the login attempt failed.
- Submit the form with an email address that's associated with a user record in the database **and with a correct password**.
  - This time you should be redirected to the "Home" page.

The login process succeeded, but a crucial piece is still missing: persisting the user's login state.

## Persisting user login state

Now it's time to handle persisting the user's login state after they've successfully logged into the website!

Remember that HTTP is a stateless protocol. Each HTTP request is independent from other requests that were executed before or after. Once the server has processed an incoming request and returned a response, it forgets about the client.

To persist to user's login state, we can implement and use sessions!

## Configuring Express to use sessions

Install the `express-session` npm package.

```
npm install express-session
```

Add a new environment variable named `SESSION_SECRET` to the `.env` file:

```
SESSION_SECRET=f1f079b1-68fe-4324-8010-0a5cff63a288
```

Don't forget to update the `.env.example` file too:

```
SESSION_SECRET=«strong session secret»
```

After updating the `.env` file, update the `config` module to export a property named `sessionSecret` initialized to

the `process.env.SESSION_SECRET` property value:

```
// ./config/index.js

module.exports = {
  environment: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 8080,
  sessionSecret: process.env.SESSION_SECRET,
  db: {
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE,
    host: process.env.DB_HOST,
  },
};
```

Now it's time to add the `express-session` middleware to the app module. Start with importing the `express-session` module:

```
const session = require('express-session');
```

Then use the `require()` function to get the session secret environment variable from the `config` module:

```
const { sessionSecret } = require('./config');
```

Now you can add the `session` middleware to the application just after the call to the `app.use()` method that adds the `cookieParser` middleware:

```
app.use(cookieParser(sessionSecret));
app.use(session({
  name: 'reading-list.sid',
  secret: sessionSecret,
  resave: false,
  saveUninitialized: false,
```

```
});
```

Some things to note about the above code:

- Notice that the `sessionSecret` config value is passed to the `cookieParser` middleware. If your application is using both `cookie-parser` and `express-session` they need to use the same secret value.
- The `name` option is set to `reading-list.sid` so that session cookies for the Reading List application won't affect any other applications that are using the general `localhost` domain.

Here's what the top portion of the `app` module should look like now:

```
// ./app.js

const express = require('express');
const morgan = require('morgan');
const cookieParser = require('cookie-parser');
const session = require('express-session');

const { sessionSecret } = require('./config');
const bookRoutes = require('./routes/book');
const userRoutes = require('./routes/user');

const app = express();

app.set('view engine', 'pug');
app.use(morgan('dev'));
app.use(cookieParser(sessionSecret));
app.use(session({
  name: 'reading-list.sid',
  secret: sessionSecret,
  resave: false,
  saveUninitialized: false,
}));
app.use(express.urlencoded({ extended: false }));
app.use(bookRoutes);
app.use(userRoutes);

// Code removed for brevity.
```

**Note:** Remember that the default in-memory session store (`MemoryStore`) is only meant for local development. While the in-memory session store works for the purposes of this article, you'd want to replace it with a more robust option (i.e. `connect-pg-simple` and `connect-session-sequelize`) before deploying your application to a production environment.

## Using sessions to persist a user's login state

Now that you've configured sessions in Express application, you can persist the user's login state using a session.

Add a new module named `auth` to the root of your project and add the following code to the module:

```
// ./auth.js

const loginUser = (req, res, user) => {
  req.session.auth = {
    userId: user.id,
  };
};

module.exports = {
  loginUser,
};
```

Putting all of the authentication related code in its own module helps to keep things organized in the project. It also helps to keep your modules focused on solving a single problem or group of related problems. All of this will improve the readability and maintainability of your project.

Now update the `./routes/user` module to import the `loginUser()` function from the `auth` module:

```
const { loginUser } = require('../auth');
```

Then within the `POST /user/login` route handler add a call to the `loginUser()` function just before redirecting the user to the default route if the password matched:

```
// Code removed for brevity.

if (passwordMatch) {
  // If the password hashes match, then login the user
  // and redirect them to the default route.
  loginUser(req, res, user);
  return res.redirect('/');
}
```

You can also login the user after a new user has registered. In the `POST /user/register` route handler add a call to the `loginUser()` function after saving the user to the database but before redirecting then to the default route:

```
// Code removed for brevity.

if (validatorErrors.isEmpty()) {
  const hashedPassword = await bcrypt.hash(password, 10);
  user.hashedPassword = hashedPassword;
  await user.save();
  loginUser(req, res, user);
  res.redirect('/');
} else {
  // Code removed for brevity.
```

```
}
```

For your reference, here are the updated POST /user/register and /user/login route handlers:

```
router.post('/user/register', csrfProtection, userValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      firstName,
      lastName,
      password,
    } = req.body;

    const user = db.User.build({
      emailAddress,
      firstName,
      lastName,
    });

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      const hashedPassword = await bcrypt.hash(password, 10);
      user.hashedPassword = hashedPassword;
      await user.save();
      loginUser(req, res, user);
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) =>
error.msg);
      res.render('user-register', {
        title: 'Register',
        user,
        errors,
        csrfToken: req.csrfToken(),
      });
    }
  }));

// Code removed for brevity.

router.post('/user/login', csrfProtection, loginValidators,
  asyncHandler(async (req, res) => {
    const {
      emailAddress,
      password,
    } = req.body;

    let errors = [];
    const validatorErrors = validationResult(req);
```

```

    if (validatorErrors.isEmpty()) {
      // Attempt to get the user by their email address.
      const user = await db.User.findOne({ where: { emailAddress
    } });

    if (user !== null) {
      // If the user exists then compare their password
      // to the provided password.
      const passwordMatch = await bcrypt.compare(password,
user hashedPassword.toString());

      if (passwordMatch) {
        // If the password hashes match, then login the user
        // and redirect them to the default route.
        loginUser(req, res, user);
        return res.redirect('/');
      }
    }

    // Otherwise display an error message to the user.
    errors.push('Login failed for the provided email address
and password');
  } else {
    errors = validatorErrors.array().map((error) => error.msg);
  }

  res.render('user-login', {
    title: 'Login',
    emailAddress,
    errors,
    csrfToken: req.csrfToken(),
  });
}));

```

## Testing user login state persistence

Run the application (if it's not already running) and use the "Register" and "Login" pages to register a new user and login an existing user. Everything should work as it did before, but the user's login state is being persisted in session.

At this point in the project, there isn't any visual indication if the user is logged in or not (that's something that you'll fix in a bit). If you open the DevTools in Chrome and view the "Application" tab, you can view the cookies for `http://localhost:8080`. After



registering a new user or logging in an existing user, you should see a cookie named `reading-list.sid`. That's the session cookie!

## Restoring the authenticated user from session

---

Now that you're persisting a user's login state to session, you need to make that user's information easily accessible to your application when it's processing requests.

In the `auth` module, define a middleware function named `restoreUser()` to retrieve the user's information from the database if they're authenticated:

```
// ./auth.js

const db = require('./db/models');

const loginUser = (req, res, user) => {
  req.session.auth = {
    userId: user.id,
  };
};

const restoreUser = async (req, res, next) => {
  // Log the session object to the console
  // to assist with debugging.
  console.log(req.session);

  if (req.session.auth) {
    const { userId } = req.session.auth;

    try {
      const user = await db.User.findByPk(userId);

      if (user) {
        res.locals.authenticated = true;
        res.locals.user = user;
        next();
      }
    } catch (err) {
      res.locals.authenticated = false;
      next(err);
    }
  } else {
    res.locals.authenticated = false;
    next();
  }
}
```

```

    }
  };

module.exports = {
  loginUser,
  restoreUser,
};

```

The `restoreUser()` middleware function starts by logging the `req.session` object to the console. Doing this will help with testing and debugging.

Then the function checks if the `req.session.auth` property is defined to determine if there's an authenticated user. If there is, then it uses destructuring to extract the `userId` from the `req.session.auth` property and calls the `db.User.findByPk()` method to retrieve the user from the database.

If the user is successfully retrieved from the database, then the `res.locals` object is used to define and set two properties:

- `authenticated` - Set to `true` to indicate that the current request has an authenticated user; and
- `user` - Set to the user that was just retrieved from the database.

The `res.locals` object is scoped to the current request and available to anything that follows the `restoreUser()` middleware function, including middleware and route handler functions and any views that are rendered as part of the current request/response cycle. It's a convenient way to pass values to other middleware, route handlers, or views.

If the `req.session.auth` property isn't defined or if the `db.User.findByPk()` method call throws an error then

the `res.locals.authenticated` property is set to `false` to indicate that the current request doesn't have an authenticated user (i.e. it's an anonymous request).

After defining the `restoreUser()` function and exporting it from the `auth` module, you need to import it into the `app` module:

```
const { restoreUser } = require('./auth');
```

Then add the `restoreUser()` middleware function to the application just before the routes are added:

```

app.use(restoreUser);
app.use(bookRoutes);
app.use(userRoutes);

```

Retrieving the user's information from the database on every request, instead of storing the user's information in the session, ensures that the user's information doesn't get stale. While it's possible to refresh the session if the user were to change their information using the application, using that approach could break if a user can change their information using other means (e.g. a mobile app).

## Displaying the user's login state

It's helpful to display to the end user whether or not they're currently logged in. A common approach is to display login and registration links or a welcome message in the header of the website.

If the user isn't logged in, they would see links to log in or register:

*Login / Register*

If the user is logged in, they would be welcomed and have access to logging out:

*Welcome «current user name»! / Logout*

To do that, update your `./views/layout.pug` template to the following:

```
doctype html
html
  head
    meta(charset='utf-8')
    meta(name='viewport' content='width=device-width, initial-scale=1, shrink-to-fit=no')
    link(rel='stylesheet' href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css' integrity='sha384-Vkoo8x4CGs03+Hhvxv8T/Q5PaXtkKtu6ug5T0eNV6gBiFeWPGFN9MuhOf23Q9Ifjh' crossorigin='anonymous')
    title Reading List - #{title}
  body
    nav(class='navbar navbar-expand-lg navbar-dark bg-primary')
      a(class='navbar-brand' href='/') Reading List
      button(class='navbar-toggler' type='button' data-toggle='collapse' data-target='#navbarText' aria-controls='navbarText' aria-expanded='false' aria-label='Toggle navigation')
        span(class='navbar-toggler-icon')
      div(class='collapse navbar-collapse' id='navbarText')
        ul(class='navbar-nav mr-auto')
          //- Empty menu keeps the content that follows the
          //- unordered list correctly positioned on the
          //- right side of the navbar.
          if locals.authenticated
            span(class='navbar-text px-4') Welcome
            #{user.firstName}!
            form(class='form-inline pr-4' action='/user/logout' method='post')
              button(class='btn btn-sm btn-warning' type='submit') Logout
          else
            span(class='navbar-text px-4')
              a(class='btn btn-sm btn-dark mr-2' href='/user/login') Login
```

```

        a(class='btn btn-sm btn-dark'
href='/user/register') Register
    .container
        h2(class='py-4') #{title}
        block content
        script(src='https://code.jquery.com/jquery-
3.4.1.slim.min.js' integrity='sha384-
J6qa4849bLE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ
+n' crossorigin='anonymous')
        script(src='https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist
/umd/popper.min.js' integrity='sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfoo
Ao' crossorigin='anonymous')
        script(src='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1
/js/bootstrap.min.js' integrity='sha384-
wfSDF2E50Y2D1uUdj003uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl30g8ifw
B6' crossorigin='anonymous')

```

A good portion of the new code is related to styling the Bootstrap Navbar component. The section that's responsible for displaying the user's login state is this bit:

```

if locals.authenticated
    span(class='navbar-text px-4') Welcome #{user.firstName}!
    form(class='form-inline pr-4' action='/user/logout'
method='post')
        button(class='btn btn-sm btn-warning' type='submit') Logout
else
    span(class='navbar-text px-4')
        a(class='btn btn-sm btn-dark mr-2' href='/user/login')
Login
        a(class='btn btn-sm btn-dark' href='/user/register')
Register

```

Notice that the view is using the `locals.authenticated` property to determine if the current user is logged in or not. `locals` within a view is the same object that's available via `res.locals`. Remember that the `restoreUser` function defined in the `auth` module is responsible for determining if there's an authenticated user stored in session and defining the initializing the `res.locals.authenticated` property to the appropriate value.

If the current user isn't logged in, then links (styled as buttons using Bootstrap CSS classes) are rendered to the "Login" and "Register" pages. If the current user is logged in, then a short, friendly "welcome" message is rendered along with a simple form that contains a single "Logout" submit button.

Now if you run and test your application, you'll see the current user's login state displayed in the header! If you log in and click the "Logout" button in the header, you'll receive a "Page Not Found" error. This is occurring because the `POST /user/logout` route doesn't exist. Time to fix that!

## Implementing user logout

Before adding the new route to logout a user, define and export a `logoutUser()` function in the `auth` module:

```
const logoutUser = (req, res) => {  
  delete req.session.auth;  
};
```

The `logoutUser()` function uses the [JavaScript delete operator](#) to remove the `auth` property from the `req.session` object which destroys the user's persisted login state.

Now you're ready to add the `POST /user/logout` to the `./routes/user` module to process `POST` requests from the logout form. Start by importing the `logoutUser()` function from the `auth` module:

```
const { loginUser, logoutUser } = require('../auth');
```

Then add the new route:

```
router.post('/user/logout', (req, res) => {  
  logoutUser(req, res);  
  res.redirect('/user/login');  
});
```

Notice that the route doesn't use the `csrfProtection` middleware.

The `POST /user/logout` route isn't modifying any of the user's data in the database, so for simplicity's sake the route isn't requiring a valid CSRF token to be present on the request.

After calling the `logoutUser()` function to logout the user, the user is redirected to the `/user/login` route. In some situations, it'd be more appropriate to redirect the user to the default route, but in just a bit you're going to update the default route to only be visible if the current user is logged in. Given that, it's a better option to redirect the user to the "Login" page.

## Testing the latest changes

Run the application (if it's not already running) and use the "Login" page to login an existing user. After logging into the website, you should now see the user's first name displayed in the header.

Remember that the `restoreUser()` function logs the `req.session` object to the console. After logging in, you should see something like this logged to the console:

```
Session {  
  cookie: { path: '/', _expires: null, originalMaxAge: null,  
httpOnly: true },  
  auth: { userId: 1 }  
}
```

The `session.auth.userId` property value is the user ID of the currently logged in user.

Now if you click the "Logout" button, you'll be redirected to the "Login" page. In the console you'll see that the `session.auth` property is no longer defined on the `session` object:

```
Session {
  cookie: { path: '/', _expires: null, originalMaxAge: null,
httpOnly: true }
}
```

## Protecting a route

---

Now that the Reading List application supports user self-registration and login, it's time to restrict access to the routes that should only be accessible to authenticated users.

For some applications, you might need to restrict access to one or two routes. For the Reading List application, you'll restrict access to all of the routes in the `./routes/book` module so that the user needs to log in view their list of books (in just a bit you'll update the `Book` model so that each book record will be associated with a user) or to add, update, or delete a book.

## Defining a middleware function to require an authenticated user

Add a new function named `requireAuth()` to the `auth` module. Update the `requireAuth()` function to redirect the user to the "Login" page if the `res.locals.authenticated` property is set to `false`, otherwise pass control to the next middleware function by calling the `next()` method:

```
// ./auth.js

const db = require('./db/models');

const loginUser = (req, res, user) => {
  // Code removed for brevity.
};

const logoutUser = (req, res) => {
  delete req.session.auth;
};
```

```

const requireAuth = (req, res, next) => {
  if (!res.locals.authenticated) {
    return res.redirect('/user/login');
  }
  return next();
};

const restoreUser = async (req, res, next) => {
  // Code removed for brevity.
};

module.exports = {
  loginUser,
  logoutUser,
  requireAuth,
  restoreUser,
};

```

Import the `requireAuth` function into the `./routes/book` module:

```
const { requireAuth } = require('../auth');
```

Then add the `requireAuth` to every route:

```

// ./routes/book.js

const express = require('express');
const { check, validationResult } = require('express-validator');

const db = require('../db/models');
const { csrfProtection, asyncHandler } = require('../utils');
const { requireAuth } = require('../auth');

const router = express.Router();

router.get('/', requireAuth, asyncHandler(async (req, res) => {
  // Code removed for brevity.
})));

router.get('/book/add', requireAuth, csrfProtection, (req, res)
=> {
  // Code removed for brevity.
});

const bookValidators = [
  // Code removed for brevity.
];

router.post('/book/add', requireAuth, csrfProtection,
bookValidators,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

```

```

router.get('/book/edit/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

router.post('/book/edit/:id(\\d+)', requireAuth, csrfProtection,
  bookValidators,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

router.get('/book/delete/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

router.post('/book/delete/:id(\\d+)', requireAuth,
  csrfProtection,
  asyncHandler(async (req, res) => {
    // Code removed for brevity.
  }));

module.exports = router;

```

Run the application (if it's not already running) and browse to the default route (/). You'll be redirected to the "Login" page. Go ahead and login using an existing account. After logging in, you'll be redirected to the default route (/) to view your list of books.

## Associating data with a user

---

Now that you've restricted access to all of the book related routes so that a logged or authenticated user is required, you can update the `Book` model so that each book record will be associated with a user.

### Defining an association

To start, run the following command in a terminal from the root of your project to remove all of the seed data from the database:

```
npx dotenv sequelize db:seed:undo:all
```

Then update the `Book` and `User` models as follows:



```
// ./db/models/book.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const Book = sequelize.define('Book', {
    // Code removed for brevity.
  }, {});
  Book.associate = function(models) {
    Book.belongsTo(models.User, {
      as: 'user',
      foreignKey: 'userId'
    });
  };
  return Book;
};

// ./db/models/user.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define('User', {
    // Code removed for brevity.
  }, {});
  User.associate = function(models) {
    User.hasMany(models.Book, {
      as: 'books',
      foreignKey: 'userId'
    });
  };
  return User;
};
```

Updating the models in this way creates a one-to-many association between the `User` and `Book` models (i.e. a user can have one or more books).

After defining the association in the models, you need to create a new migration to add the a foreign key column to the `Books` table in the database. Run the following command to add a skeleton migration file:

```
npx sequelize migration:generate --name update-book
```

Then update the contents of the `./db/migrations/[timestamp]-update-book.js` file to this:

```
'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.addColumn('Books', 'userId', {
      allowNull: false,
      references: {
        model: 'Users',
        key: 'id',
      },
      type: Sequelize.INTEGER,
    });
  },
```

```

    },
    down: (queryInterface, Sequelize) => {
    }
  };
};

```

The above migration uses the `queryInterface.addColumn()` method to add a new column to the `Books` table. In this specific case, you're adding a column named `userId` that's a foreign key to the `Users` table `id` column.

Now apply the pending migration:

```
npx dotenv sequelize db:migrate
```

## Updating the seed data

Open the `./db/seeder/[timestamp]-test-data.js` file and update its contents to this:

```

'use strict';

const bcrypt = require('bcryptjs');

module.exports = {
  up: async (queryInterface, Sequelize) => {
    const users = await queryInterface.bulkInsert('Users', [
      {
        emailAddress: 'john@smith.com',
        firstName: 'John',
        lastName: 'Smith',
        hashedPassword: bcrypt.hashSync('P@ssw0rd', 10),
        createdAt: new Date(),
        updatedAt: new Date()
      }
    ], { returning: true });

    return queryInterface.bulkInsert('Books', [
      {
        userId: users[0].id,
        title: 'The Martian',
        author: 'Andy Weir',
        releaseDate: new Date('2014-02-11'),
        pageCount: 384,
        publisher: 'Crown',
        createdAt: new Date(),
        updatedAt: new Date()
      },
    ], {

```

```

      userId: users[0].id,
      title: 'Ready Player One',
      author: 'Ernest Cline',
      releaseDate: new Date('2011-08-16'),
      pageCount: 384,
      publisher: 'Crown',
      createdAt: new Date(),
      updatedAt: new Date()
    },
    {
      userId: users[0].id,
      title: 'Harry Potter and the Sorcerer\'s Stone',
      author: 'J.K. Rowling',
      releaseDate: new Date('1998-10-01'),
      pageCount: 309,
      publisher: 'Scholastic Press',
      createdAt: new Date(),
      updatedAt: new Date()
    },
  ], {});
},
down: async (queryInterface, Sequelize) => {
  await queryInterface.bulkDelete('Books', null, {});
  return queryInterface.bulkDelete('Users', null, {});
}
};

```

An additional call to the `queryInterface.bulkInsert()` method has been added to seed a test user, "John Smith", into the `Users` database table. Notice that an object literal has been supplied to the `queryInterface.bulkInsert()` method to specify the `returning` option. The `returning` option configures the bulk insert to return the newly inserted data. This gives you a way to set the `userId` foreign key column when bulk inserting the test data into the `Books` table.

To seed the database, run the following command:

```
npx dotenv sequelize db:seed:all
```

## Updating the book related routes

Now that books are associated with a user, you can update the default route to retrieve the list of books for the currently authenticated user:

```

router.get('/', requireAuth, asyncHandler(async (req, res) => {
  const books = await db.Book.findAll({ where: { userId:
    res.locals.user.id }, order: [['title', 'ASC']] });

```

```
res.render('book-list', { title: 'Books', books });
}));
```

The options object passed into the `db.Book.findAll()` method call has been updated with a `where` property. The `where` property is set to an object literal that defines the properties to filter the query results by. In this case, you're filtering by the `userId` column, using the `res.locals.user.id` property value.

When users initially create an account, they won't have any books in their reading list. You can update the `book-list` view to display a friendly message when that occurs:

```
// ./views/book-list.pug

extends layout.pug

block content
  div(class='py-3')
    a(class='btn btn-success' href='/book/add' role='button')
Add Book
    if books && books.length > 0
      table(class='table table-striped table-hover')
        thead(class='thead-dark')
          tr
            th(scope='col') Title
            th(scope='col') Author
            th(scope='col') Release Date
            th(scope='col') Page Count
            th(scope='col') Publisher
            th(scope='col')
        tbody
          each book in books
            tr
              td= book.title
              td= book.author
              td= book.releaseDate
              td= book.pageCount
              td= book.publisher
              td
                a(class='btn btn-primary'
href='/book/edit/${book.id}` role='button') Edit
                a(class='btn btn-danger ml-2'
href='/book/delete/${book.id}` role='button') Delete
            else
              p: em You don't have any books in your reading list!
```

When a user adds a new book, you need to update the book's `userId` property with the authenticated user's `id`:

```
router.post('/book/add', requireAuth, csrfProtection,
bookValidators,
  asyncHandler(async (req, res) => {
    const {
      title,
      author,
```

```

        releaseDate,
        pageCount,
        publisher,
    } = req.body;

    const book = db.Book.build({
        userId: res.locals.user.id,
        title,
        author,
        releaseDate,
        pageCount,
        publisher,
    });

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
        await book.save();
        res.redirect('/');
    } else {
        const errors = validatorErrors.array().map((error) =>
error.msg);
        res.render('book-add', {
            title: 'Add Book',
            book,
            errors,
            csrfToken: req.csrfToken(),
        });
    }
}
}));

```

And lastly, you need to ensure that the current can only edit or delete their own books. To be clear, as long as the user uses the application's user interface to edit and delete books, this would never be an issue. This situation would only arise if a user maliciously edited their current URL to attempt to edit or delete a book that belonged to another user.

Luckily, this issue is easy to prevent. In the `./routes/book` module, add the following function above the route definitions:

```

const checkPermissions = (book, currentUser) => {
    if (book.userId !== currentUser.id) {
        const err = new Error('Illegal operation. ');
        err.status = 403; // Forbidden
        throw err;
    }
};

```

The `checkPermissions()` function accepts a book and current user and throws an error if the book's associated user doesn't match the current user.

Then update each of the edit and delete routes to call the `checkPermissions()` function:

```
router.get('/book/edit/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const book = await db.Book.findByPk(bookId);

    checkPermissions(book, res.locals.user);

    res.render('book-edit', {
      title: 'Edit Book',
      book,
      csrfToken: req.csrfToken(),
    });
  }));

router.post('/book/edit/:id(\\d+)', requireAuth, csrfProtection,
  bookValidators,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const bookToUpdate = await db.Book.findByPk(bookId);

    checkPermissions(bookToUpdate, res.locals.user);

    const {
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    } = req.body;

    const book = {
      title,
      author,
      releaseDate,
      pageCount,
      publisher,
    };

    const validatorErrors = validationResult(req);

    if (validatorErrors.isEmpty()) {
      await bookToUpdate.update(book);
      res.redirect('/');
    } else {
      const errors = validatorErrors.array().map((error) =>
error.msg);
      res.render('book-edit', {
```

```

        title: 'Edit Book',
        book: { ...book, bookId },
        errors,
        csrfToken: req.csrfToken(),
    });
}
}));

router.get('/book/delete/:id(\\d+)', requireAuth, csrfProtection,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const book = await db.Book.findByPk(bookId);

    checkPermissions(book, res.locals.user);

    res.render('book-delete', {
      title: 'Delete Book',
      book,
      csrfToken: req.csrfToken(),
    });
  }));

router.post('/book/delete/:id(\\d+)', requireAuth,
  csrfProtection,
  asyncHandler(async (req, res) => {
    const bookId = parseInt(req.params.id, 10);
    const book = await db.Book.findByPk(bookId);

    checkPermissions(book, res.locals.user);

    await book.destroy();
    res.redirect('/');
  }));

```

## Testing one more time

Now you're ready to do your final testing!

Login using the test user that you defined in your seed data. You should see the test user's list of books. Now logout the test user and login as another user. You should now see that user's list of books. If the user doesn't have any books in their reading list, add a new book. You should now only be seeing this user's books.

Congrats on completing the Reading List application... now with authentication!

# The importance of using HTTPS

---

As a reminder, form fields are submitted in clear text to the server! This means that passwords can be *sniffed* or hijacked if the communication between the client and the server isn't encrypted. This brings us to why HTTPS (hypertext transfer protocol secure) is important. HTTP uses a digital certificate for authentication known as an SSL certificate to make a web application more secure. TLS and SSL protocol are often lumped together, as TLS is another protocol that encrypts communication between the client and server.

The use of HTTPS instead of HTTP means that TLS/SSL encryption is being used to keep form posts away from prying eyes!

In order to implement HTTPS, a domain requires an SSL certificate. SSL certificates can be purchased from companies that verify your identity so that you can be issued you a certificate for your domain. Once you have the certificate (which is simply a digital file), you then install the certificate on the server.

Typically, this server isn't your Node or Express application, but another web server that serves as a proxy server for your application. A proxy server receives all of the HTTP requests from the internet to your application's domain and forwards those requests to your application. Your application will then send responses to the proxy server, which then forwards them on to the client.

## Next steps

---

There's so much more to learn about authentication! For example, it is important to know about implementing user roles as well as other user authentication flows. Examples of other user authentication processes include email confirmation, allowing users to reset their password, and two-factor authentication.

You'll dive into more components of user authentication as you learn more about web security and authentication!

Did you find this lesson helpful?