Passport | Sign Up, Login, Logout

SELF GUIDED

Learning Goals

After this lesson, you will be able to:

- configure Passport as a middleware in our app,
- allow users to log in to our application using Passport Local Strategy,
- create protected routes,
- manage errors during the login process using the `connect-flash` package,
- allow users to logout from our application using Passport.

Introduction

Passport is a flexible and modular authentication middleware for Node.js. Remember that authentication is the process where a user logs on a website by indicating their credentials (these can be a username or email and password).

If we can use a username/email and password to log on a website, why should we use a Passport?

Well, the Passport package also gives us a set of support strategies for authentication using Facebook, Twitter, and more.

Setup

In this first lesson, we will see how we can signup, login, and logout from our Express-based web application by using Passport. We will create a project with `ironhack_generator`.

First, we have to execute the following commands:

```
$ irongenerate localPassport
```

Remember to install all the packages:

```
$ npm install
```

Finally, we should run the `npm run dev` command:

```
$ npm run dev
```

Signup

We said Passport is a modular **authentication** middleware. So how do we build this authentication functionality into our application? Starting with an app generated with `ironhack-generator`, we will create users with username and password and authentication functionality using the passport.

Model

In the `models` folder create `User.model.js` file inside it. In `models/User.model.js`, we will define the Schema with `username` and `password` as follows:

```
// models/user.js

const { Schema, model } = require('mongoose');

const userSchema = new Schema(
  {
    username: String,
    passwordHash: String
  },
  {
    timestamps: true
  }
);

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

Copy

Routes File

The routes file will be defined in the `routes/auth.routes.js`, and we will set the necessary packages and code to signup in the application:

```
// routes/auth.routes.js

const { Router } = require('express');
const router = new Router();

// User model
```

Copy

```
const User = require('../models/User.model.js');

// Bcrypt to encrypt passwords
const bcrypt = require('bcrypt');
const bcryptSalt = 10;

router.get('/signup', (req, res, next) => res.render('auth/signu

router.post('/signup', (req, res, next) => {
  const { username, password } = req.body;

  // 1. Check username and password are not empty
  if (!username || !password) {
    res.render('auth/signup', { errorMessage: 'Indicate username
    return;
  }

  User.findOne({ username })
    .then(user => {
      // 2. Check user does not already exist
      if (user !== null) {
        res.render('auth/signup', { message: 'The username alrea
        return;
      }

      // Encrypt the password
      const salt = bcrypt.genSaltSync(bcryptSalt);
      const hashPass = bcrypt.hashSync(password, salt);

      //
      // Save the user in DB
      //

      const newUser = new User({
        username,
        password: hashPass
      });

      newUser
        .save()
        .then(() => res.redirect('/'))
        .catch(err => next(err));
    })
    .catch(err => next(err));
});

module.exports = router;
```

Don’t forget to install the package `bcrypt`

```
$ npm install bcrypt
```

Form

We also need a form to allow our users to signup in the application. We will put the `hbs` file in the following path: `views/auth/signup.hbs` path. Create the `/views/auth/` folder and place the `signup.hbs` file inside it. The form will look like this:

Passport | Sign Up, Login, Logout

```
{{! views/auth/signup.hbs }}

<h2>Signup</h2>

<form action="/signup" method="POST" id="form-container">
 <div>
  <label for="username">Username</label>
  <input id="username" type="text" name="username">
 </div>

 <div>
  <label for="password">Password</label>
  <input id="password" type="password" name="password">
 </div>

 {{#if errorMessage}}
  <div class="error-message">{{errorMessage}}</div>
 {{/if}}

 <div>
  <button>Create account</button>
 </div>

 <p class="account-message">
  Do you already have an account?
  <a href="/login">Login</a>
 </p>

</form>

```

Routes File

Last, but not least, we will have to define the routes in the `app.js` file. We will mount our authentication routes on the `/` path.

```
// app.js

// ...

// Routes
const router = require('./routes/auth-routes');
app.use('/', router);

module.exports = app;
```

If we execute the server and open the browser with `http://localhost:3000/signup` URL, we will be able to signup in our app.

Login

We have created the user model to access the website through a username and password. Now we are going to use Passport to log in to our app. The first thing we have to do is to choose the Strategy we are going to use. **A strategy defines how we will authenticate the user**.

There are 500+ strategies available through Passport. In this case, we will use a local strategy which is based on using the username and password to authenticate users.

Before we start coding, we have to configure the Passport npm package in our app.

Passport configuration

Passport works as a middleware in our application, so we should know how to add the basic configuration to it.

First, we have to install the packages we need:

- `passport`,
- `passport-local` (which will allow us to use username and password to log in),
- `express-session` and
- `connect-mongo`:

```
$ npm install passport passport-local express-s
```

Once the packages are installed, we have to require them in the `app.js` file:

```
// app.js
// ... skipped existing imports at the end of file

const session = require('express-session');
const MongoStore = require('connect-mongo')(session);

const bcrypt = require('bcrypt');
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// ... the rest of app.js stays untouched
```

Next up, we have to configure the session middleware. First of all, we have to configure the `express-session`, indicating which is the secret key it will use to be generated:

```
// app.js

// ...
// add the session right before the routes middleware (toward th
app.use(
  session({
    secret: process.env.SESSION_SECRET,
    resave: true,
    saveUninitialized: false, // <== false if you don't want to 
    cookie: {
      sameSite: 'none',
      httpOnly: true,
      maxAge: 60000 // 60 * 1000 ms === 1 min
    },
    store: new MongoStore({
      mongooseConnection: mongoose.connection
    })
  })
);
```

Inside `.env` file, create a variable `SESSION_SECRET` and save some string there. Example:

```
# .env file
SESSION_SECRET = "our-passport-local-strategy-app"
```

Then, we have to initialize passport and passport session, both of them like a middleware:

```
// app.js

// ...
// add the following lines after the session

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

// routes middleware
```

Copy

The following step is to define three methods that Passport needs actually to work. These methods are:

- the Strategy,
- the user serializer and
- the user deserializer.

You can find the descriptions of all the methods in the **Passport documentation**.

We use each of these configurations as follows:

- **Strategy** - defines which strategy we are going to use, and its configuration, that includes error control as well.
- **User serialize** and **User deserialize** - these methods will define which data is kept in the session, and how to recover this information from the database.

❗ The following code **needs to be placed before the `passport.initialize()` method**.

```
// app.js

// ...

const User = require('./models/user.js');
```

Copy

```

// ...

passport.serializeUser((user, cb) => cb(null, user._id));

passport.deserializeUser((id, cb) => {
  User.findById(id)
    .then(user => cb(null, user))
    .catch(err => cb(err));
});

passport.use(
  new LocalStrategy(
    { passReqToCallback: true },
    {
      usernameField: 'username', // by default
      passwordField: 'password' // by default
    },
    (username, password, done) => {
      User.findOne({ username })
        .then(user => {
          if (!user) {
            return done(null, false, { message: 'Incorrect userna
          }

          if (!bcrypt.compareSync(password, user.password)) {
            return done(null, false, { message: 'Incorrect passwo
          }

          done(null, user);
        })
        .catch(err => done(err));
    }
  )
);
// ...
```

💡 `usernameField` and `passwordField` options allow you to name your `username` and `password` `<input>` fields differently in your `<form>`, see: http://www.passportjs.org/docs/username-password/#parameters

💡 For more info about `passport.serialize`/`deserialize`, see Passport's doc and also that Stack Overflow answer.

This is all the middleware configuration we need to add to our application to be able to use Passport. The next step is to configure the passport to support *logging in*.

Routes

First, we have to require the `passport` package since we need to use it in our routes. We will add this line at the beginning of the file:

```
// routes/auth-routes.js

...

const passport = require('passport');

...
```

Then, we have to define the routes and the corresponding functionality associated with each one. The `GET` route has one mission, and that is to load the view we will use. Meanwhile, the `POST` will contain the Passport functionality. The routes are in `routes/auth-routes.js`, and we have to add the following:

```
// routes/auth-routes.js

// ...

router.get('/login', (req, res, next) => res.render('auth/login'

router.post(
  '/login',
  passport.authenticate('local', {
    successRedirect: '/',
    failureRedirect: '/login'
  })
);
```

💡 If you need more control over the flow, you can also:

```
router.post('/login', (req, res, next) => {                Copy
  passport.authenticate('local', (err, theUser, failureDeta
    if (err) {
      // Something went wrong authenticating user
      return next(err);
    }

    if (!theUser) {
      // Unauthorized, `failureDetails` contains the error
      res.render('auth/login', { errorMessage: 'Wrong passw
      return;
    }

    // save user in session: req.user
    req.login(theUser, err => {
      if (err) {
        // Session save went bad
        return next(err);
      }

      // All good, we are now logged in and `req.user` is 
      res.redirect('/');
    });
  })(req, res, next);
});
```

Side note: calling `passport.authenticate("local", ...)` will call our previously defined `LocalStrategy`.

Cool, huh? We don't have to do anything else to be able to start a session with the Passport! We need just 5 lines of code. Let's create the form to be able to log in.

Login form

Following the same file pattern we have used until now, we will create the form view in the `/views/auth/login.hbs` path. It will contain the form, with username and password fields:

```
{{! views/auth/login.hbs }}

<form action="/login" method="POST">
 <div>
 <label for="username">Username:</label>
 <input type="text" name="username">
 </div>

 <div>
 <label for="password">Password:</label>
 <input type="password" name="password">
 </div>

 {{#if errorMessage}}
 <div class="error-message">{{errorMessage}}</div>
 {{/if}}

 <div>
 <button>Log in</button>
 </div>
</form>

```

If we start the server, we will be able to log in. How can we prove we are logged in? Let's create a **protected route** to be 100% sure what we have done is working fine.

Authentication page

Once logged-in, the passport sets the `req.user` to our DB user. We can use that to decide if some pages will be accessible to the user or not. If a user is in the session (if req.user exists), then the user can access a page. Otherwise, they can't.

```
// routes/auth-routes.js

// ...

router.get('/private-page', (req, res) => {
  if (!req.user) {
    res.redirect('/login'); // can't access the page, so go and
    return;
  }

  // ok, req.user is defined
  res.render('private', { user: req.user });
});
```

As you can see, we are rendering a page that we should define in the `views/private.hbs` path. This page will just contain the following:

```
{{! views/private.hbs }}

<h1>Private page</h1>

<p>Welcome {{user.username}}</p>

```

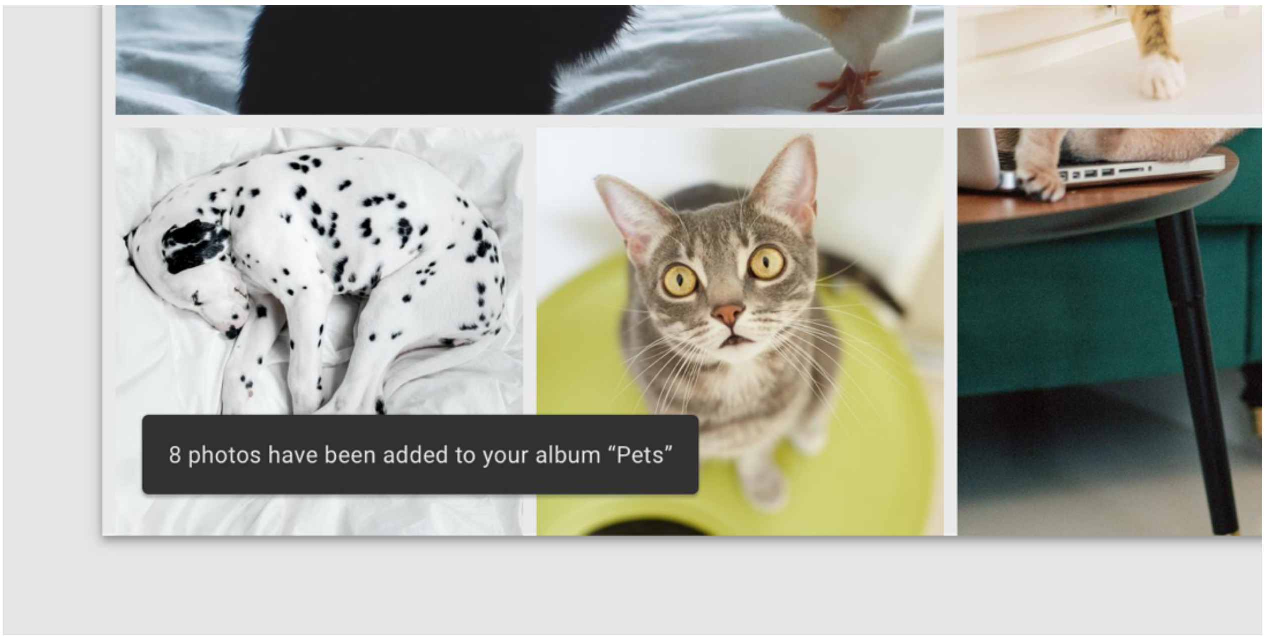
If we try to access the page without being logged in, the application should redirect us to `/login` page.

Once you are logged in, you should be able to access the page.

Error control

The package `connect-flash` is used to manage flash messages.

A flash message is a brief message that will only appear once in our app :



First, we have to install the package in our project:

```
$ npm install connect-flash
```

Copy

Once it's installed, we have to require it at the beginning of the `app.js` and `app.use()` it:

```
// app.js

...

const flash = require('connect-flash');

// ...

app.use(flash());

...
```

Copy

In the `routes/auth-route.js`, let's add `failureFlash` option:

```js
// routes/auth-route.js

// ...

router.post(
  '/login',
  passport.authenticate('local', {
    successRedirect: '/',
    failureRedirect: '/login',
    failureFlash: true // !!!
  })
);
```

As you can see above, we set a property called `failureFlash` to true. This is what will allow us to use flash messages in our application. We just have to redefine the `GET` method to send the errors to our view:

```js
// routes/auth-route.js

// ...

router.get('/login', (req, res, next) => {
  res.render('auth/login', { errorMessage: req.flash('error') });
});
```

Once we have added the error control, the login process is completed. To complete the basic authorization process, we have to create the logout method.

Logout

Passport exposes a `logout()` function on `req` object that can be called from any route handler which needs to terminate a login session. We will declare the `logout` route in the `auth-routes.js` file as it follows:

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

Copy

To finish up with this section, we just have to add a link requesting the `/logout` route in the browser, so we allow users to log out from our application.

Summary

In this learning unit, we have seen that the Passport is used to authenticate users in our application, but not for authorization.

We have reviewed how we can authorize users in our application and how to combine this functionality with passport authentication.

We have also seen how we can protect routes and handle errors during the login process with different npm packages we have to install and configure.

Finally, we created the functionality to allow users to log out of our application.

Extra Resources

- Passport documentation
- NPM `connect-flash` package
- Local Strategy example in Github (old example)

Mark as completed

PREVIOUS
class code

NEXT
Passport | Strategies and Social Login