# NodeJS

## AUTHENTICATION BASICS

Creating users and allowing them to log in and out of your web apps is a crucial functionality that we are finally ready to learn! There is quite a bit of setup involved here, but thankfully none of it is too tricky. You'll be up and running in no time! In this lesson, we're going to be using passportJS an excellent middleware to handle our authentication and sessions for us.

We're going to be building a very minimal express app that will allow users to sign up, log in, and log out. For now, we're just going to keep everything except the views in one file to make for easier demonstration, but in a real-world project, it is best practice to split our concerns and functionality into separate modules.

## Learning Outcomes

By the end of this lesson, you should be able to do the following:

### PassportJS

- Understand the use order for the required middleware.
- Describe what Strategies are.
- Use the LocalStrategy to authenticate users.
- Explain the purpose of cookies in authentication.
- Refreshed on prior learning material (routes, templates, middleware).
- Use PassportJS to set up user authentication with Express.

### Data Security/Safety

- Describe what bcrypt is and its use.
- Explain the importance of password hashing.
- Describe bcrypt's `compare` function.

## Set Up

We're going to be using another Mongo database, so before we begin log in to your mongo provider and create a new database and save its URL string somewhere handy.

To begin, let's set up a very minimal express app with a single MongoDB model for our users. Create a new directory and use `npm init` to start the package.json file then run the following to install all the dependencies we need:

```
npm install express express-session mongoose passport passport-local ejs
```

Next, let's create our `app.js`:

**IMPORTANT NOTE**: For the moment we are saving our users with just a plain text password. This is a *really* bad idea for any real-world project. At the end of this lesson, you will learn how to properly secure these passwords using bcrypt. Don't skip that part.

```
/////// app.js

const express = require("express");
const path = require("path");
const session = require("express-session");
const passport = require("passport");
const LocalStrategy = require("passport-local").Strategy;
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const mongoDb = "YOUR MONGO URL HERE";
mongoose.connect(mongoDb, { useUnifiedTopology: true, useNewUrlParser: true });
const db = mongoose.connection;
db.on("error", console.error.bind(console, "mongo connection error"));

const User = mongoose.model(
  "User",
  new Schema({
    username: { type: String, required: true },
    password: { type: String, required: true }
  })
);

const app = express();
app.set("views", __dirname);
app.set("view engine", "ejs");

app.use(session({ secret: "cats", resave: false, saveUninitialized: true }));
app.use(passport.initialize());
app.use(passport.session());
app.use(express.urlencoded({ extended: false }));

app.get("/", (req, res) => res.render("index"));

app.listen(3000, () => console.log("app listening on port 3000!"));
```

Most of this should look familiar to you by now, except for the new imported middleware for express-session and passport. We are not actually going to be using express-session directly, it is a dependency that is used in the background by passport.js. You can take a look at what it does here.

To keep things simple, our view engine is set up to just look in the main directory, and it's looking for a template called `index.ejs` so go ahead and create that:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
```

```
</head>
<body>
  <h1>hello world!</h1>
</body>
</html>
```

## Creating Users

The first thing we need is a sign up form so we can actually create users to authenticate! In the Library Tutorial website, you learned about validating and sanitizing inputs. This is a *really good idea*, but for the sake of brevity, we're going to leave that out here. Don't forget to include sanitation and validation when you get to the project.

Create a new template called `sign-up-form`, and a route for `/sign-up` that points to it:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
  <h1>Sign Up</h1>
  <form action="" method="POST">
    <label for="username">Username</label>
    <input name="username" placeholder="username" type="text" />
    <label for="password">Password</label>
    <input name="password" type="password" />
    <button>Sign Up</button>
  </form>
</body>
</html>
```

```
//// app.js

app.get("/sign-up", (req, res) => res.render("sign-up-form"));
```

Next, create an `app.post` for the sign up form so that we can add users to our database (remember our notes about sanitation, and using plain text to store passwords...).

```
app.post("/sign-up", (req, res, next) => {
  const user = new User({
    username: req.body.username,
    password: req.body.password
  }).save(err => {
    if (err) {
      return next(err);
    };
    res.redirect("/");
  });
});
```

Let's reiterate: this is not a particularly safe way to create users in your database... BUT you should now be able to visit `/sign-up`, and submit the form. If all goes well it'll redirect you to the index and you will be able to go see your newly created user inside your database.

## Authentication

Now that we have the ability to put users in our database, let's allow them to log-in to see a special message on our home page! We're going to step through the process one piece at a time, but first, take a minute to glance at the passportJS website the documentation here has pretty much everything you need to get set up. You're going to want to refer back to this when you're working on your project.

PassportJS uses what they call *Strategies* to authenticate users. They have over 500 of these strategies, but we're going to focus on the most basic (and most common), the username-and-password, or what they call the `LocalStrategy` (documentation here). We have already installed and required the appropriate modules so let's set it up!

We need to add 3 functions to our app.js file, and then add an app.post for our `/log-in` path. Add them somewhere before the line that initializes passport for us: `app.use(passport.initialize())`.

### Function one : setting up the LocalStrategy

```
passport.use(
  new LocalStrategy((username, password, done) => {
    User.findOne({ username: username }, (err, user) => {
      if (err) {
        return done(err);
      };
      if (!user) {
        return done(null, false, { message: "Incorrect username" });
      }
      if (user.password !== password) {
        return done(null, false, { message: "Incorrect password" });
      }
      return done(null, user);
    });
  })
);
```

This function is what will be called when we use the `passport.authenticate()` function later. Basically, it takes a username and password, tries to find the user in our DB, and then makes sure that the user's password matches the given password. If all of that works out (there's a user in the DB, and the passwords match) then it authenticates our user and moves on! We will not be calling this function directly, so you won't have to supply the `done` function. This function acts a bit like a middleware and will be called for us when we ask passport to do the authentication later.

## Functions two and three: Sessions and serialization

To make sure our user is logged in, and to allow them to *stay* logged in as they move around our app passport will use some data to create a cookie which is stored in the user's browser. These next two functions define what bit of information passport is looking for when it creates and then decodes the cookie. The reason they require us to define these functions is so that we can make sure that whatever bit of data it's looking for actually exists in our Database! For our purposes, the functions that are listed in the passport docs will work just fine.

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  User.findById(id, function(err, user) {
    done(err, user);
  });
});
```

Again, we aren't going to be calling these functions on our own, they're used in the background by passport.

## Log in form

To keep things nice and simple let's go ahead and add the login form directly to our index template. The form will look just like our sign-up form, but instead of `POST`ing to `/sign-up` we'll add an `action` to it so that it `POST`s to `/log-in` instead. Add the following to your index template:

```
<h1>please log in</h1>
<form action="/log-in" method="POST">
  <label for="username">Username</label>
  <input name="username" placeholder="username" type="text" />
  <label for="password">Password</label>
  <input name="password" type="password" />
  <button>Log In</button>
</form>
```

… and now for the magical part! Add this route to your app.js file:

```
app.post(
  "/log-in",
  passport.authenticate("local", {
    successRedirect: "/",
    failureRedirect: "/"
  })
);
```

As you can see, all we have to do is call `passport.authenticate()`. This middleware performs numerous functions behind the scenes. Among other things, it looks at the request body for parameters named `username` and `password` then runs the `LocalStrategy` function that we defined earlier to see if the username and password are in the database. It then creates a session cookie that gets stored in the user's browser, and that we can access in all future requests to see whether or not that user is logged in. It can also redirect you to different routes based on whether the login is a success or a failure. If we had a separate login page we might want to go back to that if the login failed, or we might want to take the user to their user dashboard if the login is successful. Since we're keeping everything in the index we want to go back to "/" no matter what.

If you fill out and submit the form now, everything should technically work, but you won't actually SEE anything different on the page… let's fix that.

The passport middleware checks to see if there is a user logged in (by checking the cookies that come in with the `req` object) and if there is, it adds that user to the request object for us. So, all we need to do is check for `req.user` to change our view depending on whether or not a user is logged in.

Edit your `app.get("/")` to send the user object to our view like so:

```
app.get("/", (req, res) => {
  res.render("index", { user: req.user });
});
```

and then edit your view to make use of that object like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
  <% if (user) {%>
    <h1>WELCOME BACK <%= user.username %></h1>
    <a href="/log-out">LOG OUT</a>
  <% } else { %>
    <h1>please log in</h1>
    <form action="/log-in" method="POST">
      <label for="username">Username</label>
      <input name="username" placeholder="username" type="text" />
      <label for="password">Password</label>
      <input name="password" type="password" />
      <button>Log In</button>
    </form>
  <%}%>
</body>
</html>
```

So, this code checks to see if there is a user defined… if so it offers a welcome message, and if NOT then it shows the login form. Neat!

As one last step… let's make that log out link actually work for us. As you can see it's simply sending us to `/log-out` so all we need to do is add a route for that in our app.js. Conveniently, the passport middleware adds a logout function to the `req` object, so logging out is as easy as this:

```
app.get("/log-out", (req, res) => {
  req.logout();
  res.redirect("/");
});
```

You should now be able to visit `/sign-up` to create a new user, then log-in using that user's username and password, and then log out by clicking the log out button!

### A quick tip

In express, you can set and access various local variables throughout your entire app (even in views) with the `locals` object. We can use this knowledge to write ourselves a custom middleware that will simplify how we access our current user in our views.

Middleware functions are simply functions that take the `req` and `res` objects, manipulate them, and pass them on through the rest of the app.

```
app.use(function(req, res, next) {
  res.locals.currentUser = req.user;
  next();
});
```

If you insert this code somewhere between where you instantiate the passport middleware and before you render your views, you will have access to the `currentUser` variable in all of your views, and you won't have to manually pass it into all of the controllers in which you need it.

## Securing passwords with bcrypt

Now, let's go back and learn how to securely store user passwords so that if anything ever goes wrong, or if someone gains access to our database, our user passwords will be safe. This is *insanely* important, even for the simplest apps, but luckily it's also really simple to set up.

First `npm install bcryptjs`. There is another module called `bcrypt` that does the same thing, but it is written in C++ and is sometimes a pain to get installed. The C++ `bcrypt` is technically faster, so in the future it might be worth getting it running, but for now, the modules work the same so we can just use `bcryptjs`.

Once it's installed you need to require it at the top of your app.js and then we are going to put it to use where we save our passwords to the DB, and where we compare them inside the LocalStrategy.

### Storing hashed passwords:

Edit your `app.post("/sign-up")` to use the bcrypt.hash function which works like this:

```
bcrypt.hash("somePassword", 10, (err, hashedPassword) => {
  // if err, do something
  // otherwise, store hashedPassword in DB
});
```

The hash function is somewhat slow, so all of the DB storage stuff needs to go inside the callback. Check to see if you've got this working by signing up a new user with a simple password, then go look at your DB entries to see how it's being stored. If you've done it right, your password should have been transformed into a really long random string.

It's important to note that *how* hashing works is beyond the scope of this lesson. Password hashes are the result of passing the user's password through a one-way hash function.

### Comparing hashed passwords:

We will use the `bcrypt.compare()` function to validate the password input. The function compares the plain-text password in the request object to the hashed password.

Inside your `LocalStrategy` function we need to replace the `user.password !== password` expression with the `bcrypt.compare()` function.

```
bcrypt.compare(password, user.password, (err, res) => {
  if (res) {
    // passwords match! log user in
    return done(null, user)
  } else {
    // passwords do not match!
    return done(null, false, { message: "Incorrect password" })
  }
```

You should now be able to log in using the new user you've created (the one with a hashed password). Unfortunately, users that were saved BEFORE you added bcrypt will no longer work, but that's a small price to pay for security! (and a good reason to include bcrypt from the start on your next project)

## Additional Resources

This section contains helpful links to other content. It isn't required, so consider it supplemental if you need to dive deeper into something.

- This article goes into great detail about the passport local strategy and brings the magic that happens behind the scenes into the light. It provides a comprehensive foundation for how session-based authentication works using browser cookies along with backend sessions to manage users.
- If you like video content, watch this Youtube Playlist by the same author who wrote the article above. You just need to watch the first 6 videos.

Are you interested in accelerating your web development learning experience?

Get started

Thinkful

5-6 months Job Guarantee 1-on-1 Mentorship

THE ODIN PROJECT

About
FAQ
Blog
Success Stories
Contribute
Terms of Use