Local Authentication Using Passport in Node.js

By Beardscript

JavaScript

April 8, 2020

A common requirement when building a web app is to implement a login system, so that users can authenticate themselves before gaining access to protected views or resources. Luckily for those building Node apps, there's a middleware called Passport that can be dropped into any Express-based web application to provide authentication mechanisms in only a few commands.

In this tutorial, I'll demonstrate how to use Passport to implement local authentication (that is, logging in with a username and password) with a MongoDB back end. If you're looking to implement authentication via the likes of Facebook or GitHub, please refer to this tutorial.

As ever, all of the code for this article is available for download on GitHub.

Prerequisites

To follow along with this tutorial, you'll need to have Node and MongoDB installed on your machine.

You can install **Node** by heading to the official Node download page and grabbing the correct binaries for your system. Alternatively, you can use a version manager

— a program that allows you to install multiple versions of Node and switch between them at will. If you fancy going this route, please consult our quick tip, "Install Multiple Versions of Node.js Using nvm".

MongoDB comes in various editions. The one we're interested in is the MongoDB Community Edition.

The project's home page has excellent documentation and I won't try to replicate that here. Rather, I'll offer you links to instructions for each of the main operating systems:

- Install MongoDB Community Edition on Windows
- Install MongoDB Community Edition on macOS
- Install MongoDB Community Edition on Ubuntu

If you use a non-Ubuntu–based version of Linux, you can check out this page for installation instructions for other distros. MongoDB is also normally available through the official Linux software channels, but sometimes this will pull in an outdated version.

Note: You don't need to enter your name and address to download MongoDB. If prompted, you can normally dismiss the dialog.

If you'd like a quick refresher on using MongoDB, check out our beginner's guide, "An Introduction to MongoDB".

Authentication Strategies: Session vs JWT

Before we begin, let's talk briefly about authentication choices.

Many of the tutorials online today will opt for token-based authentication using JSON Web Tokens (JWTs). This approach is probably the simplest and most

popular one nowadays. It relegates part of the authentication responsibility to the client and makes them sign a token that's sent with every request, to keep the user authenticated.

Session-based authentication has been around longer. This method relegates the weight of the authentication to the server. It uses cookies and sees the Node application and database work together to keep track of a user's authentication state.

In this tutorial, we'll be using session-based authentication, which is at the heart of the passport-local strategy.

Both methods have their advantages and drawbacks. If you'd like to read more into the difference between the two, this Stack Overflow thread might be a good place to start.

Creating the Project

Once all of the prerequisite software is set up, we can get started.

We'll begin by creating the folder for our app and then accessing that folder on the terminal:

```
mkdir AuthApp
cd AuthApp
```

To create the node app, we'll use the following command:

```
npm init
```

You'll be prompted to provide some information for Node's `package.json`. Just keep hitting `Return` to accept the default configuration (or use the `-y` flag).

Setting up Express

Now we need to install Express. Go to the terminal and enter this command:

```
npm install express
```

We'll also need to install the body-parser middleware which is used to parse the request body that Passport uses to authenticate the user. And we'll need to install the express-session middleware.

Let's do that. Run the following command:

```
npm install body-parser express-session
```

When that's done, create an `index.js` file in the root folder of your app and add the following content to it:

```
/*  EXPRESS SETUP  */

const express = require('express');
const app = express();

app.use(express.static(__dirname));

const bodyParser = require('body-parser');
const expressSession = require('express-session')({
  secret: 'secret',
  resave: false,
  saveUninitialized: false
});
```

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(expressSession);

const port = process.env.PORT || 3000;
app.listen(port, () => console.log('App listening on port ' + port));
```

First, we `require` Express and create our Express app by calling express(). Then we define the directory from which to serve our static files.

The next line sees us `require` the body-parser middleware, which will help us parse the body of our requests. We're also adding the express-session middleware to help us save the session cookie.

As you can, see we're configuring express-session with a `secret` to sign the session ID cookie (you should choose a unique value here), and two other fields, resave and saveUninitialized. The `resave` field forces the session to be saved back to the session store, and the `saveUninitialized` field forces a session that is "uninitialized" to be saved to the store. To learn more about them, check out their documentation, but for now it's enough to know that for our case we want to keep them `false`.

Then, we use `process.env.PORT` to set the port to the environment port variable if it exists. Otherwise, we'll default to `3000`, which is the port we'll be using locally. This gives you enough flexibility to switch from development, directly to a production environment where the port might be set by a service provider like, for instance, Heroku. Right below that, we called app.listen() with the port variable we set up and a simple log to let us know that it's all working fine and on which port is the app listening.

That's all for the Express setup. Now it's on to setting up `Passport`.

Setting up Passport

First, we install Passport with the following command:

```
npm install passport
```

Then we need to add the following lines to the bottom of the `index.js` file:

```
/*  PASSPORT SETUP  */

const passport = require('passport');

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

Here, we require `passport` and initialize it along with its session authentication middleware, directly inside our Express app.

Creating a MongoDB Data Store

Since we're assuming you've already installed Mongo, you should be able to start the Mongo shell using the following command:

```
mongo
```

Within the shell, issue the following command:

```
use MyDatabase;
```

This simply creates a datastore named `MyDatabase`.

Leave the terminal there; we'll come back to it later.

Connecting Mongo to Node with Mongoose

Now that we have a database with records in it, we need a way to communicate with it from our application. We'll be using Mongoose to achieve this. Why don't we just use plain Mongo? Well, as the Mongoose devs like to say ,A href="https://mongoosejs.com/docs/unstable/index.html">on their website:

> writing MongoDB validation, casting and business logic boilerplate is a drag.

Mongoose will simply make our lives easier and our code more elegant.

Let's go ahead and install it with the following command:

```
npm install mongoose
```

We'll also be using passport-local-mongoose, which will simplify the integration between Mongoose and Passport for local authentication. It will add a `hash` and `salt` field to our Schema in order to store the hashed password and the salt value. This is great, as passwords should never be stored as plain text in a database.

Let's install the package:

```
npm install passport-local-mongoose
```

Now we have to configure Mongoose. Hopefully you know the drill by now: add the following code to the bottom of your `index.js` file:

```
/* MONGOOSE SETUP */
```

```
const mongoose = require('mongoose');
const passportLocalMongoose = require('passport-local-mongoose');

mongoose.connect('mongodb://localhost/MyDatabase',
  { useNewUrlParser: true, useUnifiedTopology: true });

const Schema = mongoose.Schema;
const UserDetail = new Schema({
  username: String,
  password: String
});

UserDetail.plugin(passportLocalMongoose);
const UserDetails = mongoose.model('userInfo', UserDetail, 'userInfo');
```

Here we require the previously installed packages. Then we connect to our database using `mongoose.connect` and give it the path to our database. Next, we're making use of a Schema to define our data structure. In this case, we're creating a `UserDetail` schema with `username` and `password` fields.

Finally, we add `passportLocalMongoose` as a plugin to our Schema. This will work part of the magic we talked about earlier. Then, we create a model from that schema. The first parameter is the name of the collection in the database. The second one is the reference to our Schema, and the third one is the name we're assigning to the collection inside Mongoose.

That's all for the Mongoose setup. We can now move on to implementing our Passport strategy.

Implementing Local Authentication

And finally, this is what we came here to do! Let's set up the local authentication. As you'll see below, we'll just write the code that will set it up for us:

```
/* PASSPORT LOCAL AUTHENTICATION */

passport.use(UserDetails.createStrategy());

passport.serializeUser(UserDetails.serializeUser());
passport.deserializeUser(UserDetails.deserializeUser());
```

There's quite some magic going on here. First, we make `passport` use the local strategy by calling `createStrategy()` on our `UserDetails` model — courtesy of `passport-local-mongoose` — which takes care of everything so that we don't have to set up the strategy. Pretty handy.

Then we're using `serializeUser` and `deserializeUser` callbacks. The first one will be invoked on authentication, and its job is to serialize the user instance with the information we pass on to it and store it in the session via a cookie. The second one will be invoked every subsequent request to deserialize the instance, providing it the unique cookie identifier as a "credential". You can read more about that in the Passport documentation.

Routes

Now let's add some routes to tie everything together. First, we'll add a final package. Go to the terminal and run the following command:

```
npm install connect-ensure-login
```

The connect-ensure-login package is middleware that ensures a user is logged in. If a request is received that is unauthenticated, the request will be redirected to a login page. We'll use this to guard our routes.

Now, add the following to the bottom of `index.js`:

```
/* ROUTES */

const connectEnsureLogin = require('connect-ensure-login');

app.post('/login', (req, res, next) => {
  passport.authenticate('local',
  (err, user, info) => {
    if (err) {
      return next(err);
    }

    if (!user) {
      return res.redirect('/login?info=' + info);
    }

    req.logIn(user, function(err) {
      if (err) {
        return next(err);
      }

      return res.redirect('/');
    });

  })(req, res, next);
});

app.get('/login',
  (req, res) => res.sendFile('html/login.html',
  { root: __dirname })
);

app.get('/',
  connectEnsureLogin.ensureLoggedIn(),
  (req, res) => res.sendFile('html/index.html', {root: __dirname})
);

app.get('/private',
  connectEnsureLogin.ensureLoggedIn(),
  (req, res) => res.sendFile('html/private.html', {root: __dirname})
);

app.get('/user',
  connectEnsureLogin.ensureLoggedIn(),
```

```
  (req, res) => res.send({user: req.user})
);
```

At the top, we're requiring `connect-ensure-login`. We'll come back to this later.

Next, we set up a route to handle a POST request to the `/login` path. Inside the handler, we use the passport.authenticate method, which attempts to authenticate with the strategy it receives as its first parameter — in this case `local`. If authentication fails, it will redirect us to `/login`, but it will add a query parameter — `info` — that will contain an error message. Otherwise, if authentication is successful, it will redirect us to the `'/'` route.

Then we set up the `/login` route, which will send the login page. For this, we're using res.sendFile() and passing in the file path and our root directory, which is the one we're working on — hence the `__dirname`.

The `/login` route will be accessible to anyone, but our next ones won't. In the `/` and `/private` routes we'll send their respective HTML pages, and you'll notice something different here. Before the callback, we're adding the `connectEnsureLogin.ensureLoggedIn()` call. This is our route guard. Its job is validating the session to make sure you're allowed to look at that route. Do you see now what I meant earlier by "letting the server do the heavy lifting"? We're authenticating the user every single time.

Finally, we'll need a `/user` route, which will return an object with our user information. This is just to show you how you can go about getting information from the server. We'll request this route from the client and display the result.

Talking about the client, let's do that now.

The Client

The client should be quite simple. We'll create some `HTML` pages and a `CSS` file. Let's begin with the home page, or index. In your project root, create a folder called `html` and add a file called `index.html`. Add the following to it:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title> Home </title>
  <link rel="stylesheet" href="css/styles.css">
</head>

<body>
  <div class="message-box">
    <h1 id="welcome-message"></h1>
    <a href="/private">Go to private area</a>
  </div>

  <script>
    const req = new XMLHttpRequest();
    req.onreadystatechange = function () {
      if (req.readyState == 4 && req.status == 200) {
        const user = JSON.parse(req.response).user;
        document.getElementById("welcome-message").innerText = `Welcome 
      }
    };
    req.open("GET", "http://localhost:3000/user", true);
    req.send();
  </script>
</body>
</html>
```

Here we have an empty `h1` tag where we'll place our welcome message and, below that, a link to `/private`. The crucial part here is the `script` tag at the bottom where we'll handle getting the username to create the welcome message.

This is divided into four parts:

1. We instantiate the request object using `new XMLHttpRequest()`.
2. We set the `onreadystatechange` property with the function that will be called after we get our answer. In the callback, we're checking if we got a successful response and if so, we parse the response, get the user object (the one we sent in the `/user` route, remember?), and we find the `welcome-message` element to set its `innerText` to our `user.username`.
3. We `open()` the `GET` request to the user `URL` and we set the last parameter to `true` to make it `asynchronous`.
4. Finally, we `send()` the request.

Now we'll create the login page. As before, in the HTML folder create a file called `login.html` and add the following content to it:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title> Login </title>
  <link rel="stylesheet" href="css/styles.css">
</head>

<body>
  <form action="/login" method="post">
    <div class="title">
      <h3>Login</h3>
    </div>
    <div class="field">
      <label>Username:</label>
      <input type="text" name="username" />
      <br />
    </div>
    <div class="field">
      <label>Password:</label>
      <input type="password" name="password" required />
    </div>
    <div class="field">
      <input class="submit-btn" type="submit" value="Submit" required />
    </div>
    <label id="error-message"></label>
```

```
  </form>

  <script>
    const urlParams = new URLSearchParams(window.location.search);
    const info = urlParams.get('info');

    if(info) {
      const errorMessage = document.getElementById("error-message");
      errorMessage.innerText = info;
      errorMessage.style.display = "block";
    }
  </script>
</body>
</html>
```

On this page, we have a simple login form, with `username` and `password` fields, as well as a *Submit* button. Below that, we have a label where we'll display any error messages. Remember, these are contained in the query string.

The `script` tag at the bottom is far simpler this time. We're instantiating a `URLSearchParams` object passing the `window.location.search` property, which contains the parameters string in our URL. Then we use the `URLSearchParams.get()` method, passing in the parameter name we're looking for.

At this point, we either have an info message or not. So if we do, we get the `error-message` element and set its `innerText` to whatever that message is, and then set its `style.display` property to `block`. This will make it visible, given that by default it has a `display: "none"` value.

Let's set up the private page now. Again, create a file in the HTML folder with the name `private.html` and add the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <title> Private </title>
  <link rel="stylesheet" href="css/styles.css">
</head>

<body>
  <div class="message-box">
    <h2>This is a private area</h2>
    <h3>Only you can see it</h3>
    <a href="/">Go back</a>
  </div>
</body>
</html>
```

Super simple. Just a simple message and a `Go back` link which takes us back to the homepage.

That's it for the HTML, but as you probably noticed, we're referencing a `CSS` file the `head` tags. Let's add that file now. Create a folder called `css` in the root of our project and add a `styles.css` file to it, with the following content:

```
body {
  display: flex;
  align-items: center;
  background: #37474F;
  font-family: monospace;
  color: #cfd8dc;
  justify-content: center;
  font-size: 20px;
}

.message-box {
  text-align: center;
}

a {
  color: azure;
}

.field {
  margin: 10px;
}
```

```
input {
  font-family: monospace;
  font-size: 20px;
  border: none;
  background: #1c232636;
  color: #CFD8DC;
  padding: 7px;
  border: #4c5a61 solid 2px;
  width: 300px;
}

.submit-btn {
  width: 100%
}

.title {
  margin: 10px 0px 20px 10px
}

#error-message {
  color: #E91E63;
  display: block;
  margin: 10px;
  font-size: large;
  max-width: fit-content;
}
```

This will make our pages look decent enough. Let's check it out!

Grab a terminal pointing to the project root and run the following command:

```
node index.js
```

Now navigate to http://localhost:3000/ in your browser. You should be redirected to the login page. If you try to go to http://localhost:3000/private, it should redirect you to the login page again. There's our route guard doing its job.

Press `Ctrl` + `C` in the terminal window to stop our server. Then head back to the `index.js` file and, at the bottom of the file, add the following lines:

```
/* REGISTER SOME USERS */

UserDetails.register({username:'paul', active: false}, 'paul');
UserDetails.register({username:'jay', active: false}, 'jay');
UserDetails.register({username:'roy', active: false}, 'roy');
```

This uses the passport-local-mongoose `register` method to salt the password for us. We just have to pass it in in plain text.

Now we run `node index.js`. The users will be created. You should comment those last lines now.

Remember the MongoDB shell terminal we left open? Go back to it and type:

```
db.userInfo.find()
```

This should show your three users and, as you can see, the salt and hash now occupy a good portion of the space on the terminal.

That's all we need for the app to work. We're done!

Head back to the browser, try to log in with one of the credentials we entered and you'll see the login message with the given username in it.

Next Steps

We only added the necessary modules for this app to work — nothing more, nothing less. For a production app, you'll need to add other middleware and

separate your code in modules. You can take that as a challenge to set up a clean and scalable environment and grow it into something useful!

The first and easiest thing you should try is adding the `logout`, using Passport's req.logout() method.

Then you could try implementing a register flow. You'll need a registration form and a route to talk to. You should use the `UserDetails.register()` we added earlier as a template. For email confirmation, you should check out nodemailer.

Another thing you could do is try to apply these concepts to a single page application. Perhaps using Vue.js and its router. And there goes your weekend!

Conclusion

Well, we're finally at the end. In this article, we learned how to implement local authentication using `Passport` in a `Node.js` application. In the process, we also learned how to connect to `MongoDB` using `Mongoose`.

Perhaps this wasn't as easy for you as I tried to paint it, but at least you got to see that it gets easier with these tools that work some magic in the background, letting us worry only about what we're trying to build.

"Magic" tools are not always ideal, but reputable and actively maintained tools help us write less code — and code you don't write is code you don't maintain, and code you don't maintain is code you don't break.

Also, keep in mind that if a tool is actively maintained by a core team, chances are they know what they're doing better than any of us. Delegate whenever possible.

I hope you enjoyed this tutorial, and maybe got some inspiration for your next project. Happy coding!

Beardscript

Aside from being a software developer, I am also a massage therapist, an enthusiastic musician, and a hobbyist fiction writer. I love traveling, watching good quality TV shows and, of course, playing video games.