Authenticate Users With Node ExpressJS and Passport.js

Many Node.js applications require users to authenticate in order to access private content. The authentication process must be both functional and secure, and creating one from scratch can be lengthy and cumbersome. Because of this, most modern-day developers opt to use trusted libraries or outside services. Passport.js is a popular Express middleware specifically created to facilitate the login process. It is flexible, trusted by many organizations worldwide, and easy to integrate into your ExpressJS code.

In this tutorial we'll:

- Create a login form for a Node application using Passport
- Use the session authentication strategy with Passport
- Connect Passport to a MongoDB database to store user data
- Authorize only logged-in users to access a page

By the end of this tutorial, you will learn how to create a functional login page complete with authentication and authorization.

This tutorial is part 7 of 7 tutorials that walk through using Express.js for user authentication.

Goal

Create an application that performs basic authentication and authorization tasks.

Prerequisites

- What Is the Difference between Authorization and Authentication?
- Process a Login Form with ExpressJS
- Setup Express Session Authentication

Passport.js strategies

Passport is a library of authentication functions that is adaptable to the needs of most applications. What makes Passport so flexible? The main reason is that developers can choose from a cornucopia of implementation strategies.

Passport strategies are separate modules created to suit individual means of authentication. Available Passport strategies include:

- **passport-local**: local username and password authentication
- **passport-facebook**: authenticate with Facebook via OAuth
- **passport-twitter:** authenticate with Twitter via OAuth
- **passport-jwt**: use JSON web tokens instead of sessions

There are hundreds of strategies. You can even find modules tailored to specific technologies and databases. For example, in this project, we will use local authentication with MongoDB. To concentrate on the authentication code, we'll use the MongoDB wrapper Mongoose and the passport-local-mongoose module.

MongoDB

MongoDB is a NoSQL database that uses JSON-formatted documents to store data. Some NodeJS developers prefer MongoDB because of familiarity of JSON, which can be easily manipulated into JavaScript objects. Here's an example of MongoDB data:

```
{ name: 'Lori Fields',
  address: {
    street: 123 Palm Trace
    city: Miami, FL
    zip code: 33101
  }
},
{ name: 'Harry Humbug',
  address: {
    street: 6 SW 1st Street
    city: Deerfield Beach, FL
    zip code: 33442
  }
}
```

Authentication application with Passport and ExpressJS

Now that you've had a brief overview of Passport and MongoDB, we are ready to begin the project. Here's the application's file structure:

```
.
└── login
    ├── package.json
    ├── server.js
    ├── static
    │   ├── index.html
    │   ├── login.html
    │   └── secret-page.html
    └── user.js
```

As you can see, our primary project folder is named *login*. Inside *login* we have:

- A folder named *static* that contains HTML files

- *server.js* which is the root to our application and contains all our ExpressJS server code, including our routes
- *user.js* which uses Mongoose to connect to the database and create our user model
- *package.json*, the configuration file

The code in this tutorial builds on code in 2 previous tutorials:

1. How to Make a Form with ExpressJS
2. How to Set Up Express Session

If you do not understand ExpressJS or express-session, please view the previous tutorials.

Download MongoDB and create a database

Navigate to Install MongoDB Community Edition and click the link for your operating system (Linux, macOS, or Windows). Follow the directions to install and start MongoDB.

Once MongoDB is installed and started, go to the MongoDB command shell.

```
mongo
```

Then go to the database "users" (and create it, if it doesn't already exist):

```
> use users
```

At any time, you can exit the MongoDB command shell by typing the quit command.

```
> quit ()
```

Create a *package.json* file

Assuming you have already have NodeJS installed, create a package.json file inside the main project folder.

```
npm init -y
```

Install ExpressJS and dependencies

Before we start writing code, we must install the necessary dependencies.

Let's start by installing ExpressJS (server framework), body-parser (parses incoming request bodies), and express-session (cookie-based session middleware).

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

Next, we install Passport and passport-local. Passport is the authentication library and passport-local is our core authentication strategy.

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

Now we install connect-ensure-login, authorization middleware that makes it easy to restrict access to pages. It is built to work hand-in-hand with Passport, and with one function call, we can authorize routes to only logged-in users.

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

Next, we install Mongoose, an object data mapper (ODM) used to integrate MongoDB with NodeJS. The library simplifies MongoDB data modeling, facilitating the creation of JavaScript objects and database document persistence. The asynchronous nature of NodeJS query functions can make data modeling cumbersome. Using Mongoose in this project will enable us to concentrate on the authentication code instead of document creation and database integration.

```
npm install mongoose
```

Finally, we install passport-local-mongoose. This strategy integrates Mongoose with the passport-local strategy.

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

Create HTML files

Let's make sure we have all our HTML files properly written and saved in the static directory.

Example *index.html*:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Welcome Home</title>
  </head>
  <body>
    <a href="/login">Please Login Here!</a>
  </body>
</html>
```

Example *login.html*:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Example Login Form</title>
  </head>
  <body>
    <form action="//members.osiolabs.com/login" method="post">
      <!-- user input-->
      Username:<br>
      <input type="text" name="username" placeholder="Username" re
      Password:<br>
      <input type="password" name="password" placeholder="Password
      <!-- submit button -->
      <input type="submit" value="login">
    </form>
  </body>
</html>
```

Example *secret-page.html*:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Secret Page</title>
  </head>
  <body>
    <p>This is a secret page that is only available to logged-in u
     Congratulations!!!  You are in on our secret!</p>
     <a href="/logout">Log Out</a>
  </body>
</html>
```

Create *user.js*

Let's get the database code out of the way and create *user.js*. We will:

- Connect to the database
- Create our Passport user model
- Create a MongoDB document called 'userData'

Let's start by including Mongoose and passport-local-mongoose.

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

Next, we connect to the database "users" on our local server. If the database server is not on your local machine, instead of "localhost", you will write the hostname or IP address.

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

- **Connection URL**: Our connection URL is `'mongodb://localhost/users'`. 'localhost' is the host and 'users' is the database. If the database is on an outside server you may have to include more parameters, e.g.: `'mongodb://username:password@host:port/database?options...'`
- **useNewUrlParser**: This is set to avoid deprecation warnings from the native MongoDB driver; it's opt-in to facilitate conversions for older applications. Set this to `true` for new development. See Mongoose documentation for details.
- **useUnifiedTopology**: Also set to avoid deprecation warnings; briefly, the connection management engine was upgraded. Set to `true` for new development and see Mongoose documentation for details.

Now we create our *User* model, containing 2 properties: username and password. They are both String datatype.

```
const Schema = mongoose.Schema;

const User = new Schema({
  username: String,
  password: String
});
```

Finally, we are ready to export our model.

1. Using the passport-local-mongoose library, we will export `'User'` in a usable form to the passport code we are about to write.

2. We export our model as a module, so we can use it in *server.js* and create a document called `'userData'`. userData will store our user information (id, username, hash). Passport-local-mongoose automatically salts and hashes passwords.

The completed *user.js* file should look like the code below:

```
// dependencies
const mongoose = require('mongoose');
const passportLocalMongoose = require('passport-local-mongoose');
// connect to database
mongoose.connect('mongodb://localhost/users',{
  useNewUrlParser: true,
  useUnifiedTopology: true
});
// Create Model
const Schema = mongoose.Schema;

const User = new Schema({
  username: String,
  password: String
});
// Export Model
User.plugin(passportLocalMongoose);

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

Create some dummy user data

To test our application, we will need test data. Test data is temporary data we insert into the database to test code. In a live application, users would register via a sign-up form.

To test our authentication code, let's register two users. Create a file named *dummy-data.js* with the following code:

```
const UserDetails = require('./user');

UserDetails.register({ username: 'candy', active: false }, 'cane')
UserDetails.register({ username: 'starbuck', active: false }, 'red
```

This will create two users, *candy* and *starbuck*, with passwords *cane* and *redeye*.

While in the early stages of development, this is a quick way to add temporary test data to the temporary test database. You would never use this method on a production server. Other methods of inserting data include running a migration from the command line, using a form (such as a sign-up form), and placing the code above in a data migration file.

Run the code:

```
node dummy-data.js
```

Create a *server.js*

Now we are ready to create our application's root *server.js* file.

We start by including our dependencies.

```
const express = require('express'); // server software
const bodyParser = require('body-parser'); // parser middleware
const session = require('express-session');  // session middleware
const passport = require('passport');  // authentication
const connectEnsureLogin = require('connect-ensure-login'); //auth
```

And don't forget to include our User model.

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

Then let's create our Express app.

```
const app = express();
```

We must configure express-session, as the passport-local-mongoose module does not do it for you. To learn about configuring express-session, please review Setup Express Session Authentication.

```
app.use(session({
  secret: 'r8q,+&1LM3)CD*zAGpx1xm{NeQhc;#',
  resave: false,
  saveUninitialized: true,
  cookie: { maxAge: 60 * 60 * 1000 } // 1 hour
}));
```

We have to configure a bit more middleware:

- **bodyParser**: We are parsing URL-encoded data from the body
- **passport.initialize**: Middleware to use Passport with Express
- **passport.session**: Needed to use express-session with passport

```
app.use(bodyParser.urlencoded({ extended: false }));
app.use(passport.initialize());
```

Before doing any authenticating, your strategy must be declared and configured. Our strategy module, *passport-local-mongoose*, does the heavy lifting of creating the strategy for us, so all we need is the code below.

```
passport.use(User.createStrategy());
```

If we were just using *passport-local* and not *passport-local-mongoose*, we would have to configure the login strategy. See the example local strategy configuration below (not used in today's project).

```
passport.use(new LocalStrategy(
  // function of username, password, done(callback)
  function(username, password, done) {
    // look for the user data
    User.findOne({ username: username }, function (err, user) {
      // if there is an error
      if (err) { return done(err); }
      // if user doesn't exist
      if (!user) { return done(null, false, { message: 'User not f
      // if the password isn't correct
      if (!user.verifyPassword(password)) { return done(null, fals
      message: 'Invalid password.' }); }
      // if the user is properly authenticated
      return done(null, user);
    });
  }
));
```

To authenticate, Passport first looks at the user's login details, then invokes a verified callback (done). If the user gets properly

authenticated, pass the `user` into the callback. If the user does not get appropriately authenticated, pass `false` into the callback. You also have the option to pass a specific message into the callback.

When you use sessions with Passport, as soon as a user gets appropriately authenticated, a new session begins. When this transpires, we serialize the user data to the session and the user ID is stored in `req.session.passport.user`. To access the user data it is deserialized, using the user ID as its key. The user data is queried and attached to `req.user`.

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

So far, *server.js* should look like the code below:

```
const express = require('express'); // server software
const bodyParser = require('body-parser'); // parser middleware
const session = require('express-session');  // session middleware
const passport = require('passport');  // authentication
const connectEnsureLogin = require('connect-ensure-login');// auth

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

const app = express();

// Configure Sessions Middleware
app.use(session({
  secret: 'r8q,+&1LM3)CD*zAGpx1xm{NeQhc;#',
  resave: false,
  saveUninitialized: true,
  cookie: { maxAge: 60 * 60 * 1000 } // 1 hour
}));

// Configure More Middleware
app.use(bodyParser.urlencoded({ extended: false }));
```

```
// Passport Local Strategy
passport.use(User.createStrategy());

// To use with sessions
passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());
```

Define GET routes

We are ready to append the *server.js* code with the GET routes.

We start with the route to our homepage. It is accessible to everybody.

```
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/static/index.html');
})
```

Next, we create the route to our application's login page. It is also accessible to everybody.

```
app.get('/login', (req, res) => {
  res.sendFile(__dirname + '/static/login.html');
});
```

Now we create the dashboard route. It is only accessible to logged-in users.

```
app.get('/dashboard', connectEnsureLogin.ensureLoggedIn(), (req,
  res.send(`Hello ${req.user.username}. Your session ID is ${req.s
    and your session expires in ${req.session.cookie.maxAge}
    milliseconds.<br><br>
    <a href="/logout">Log Out</a><br><br>
    <a href="/secret">Members Only</a>`);
});
```

As you can see above, we call the `connectEnsureLogin.ensureLoggedIn()` function in our route handler to ensure only logged-in users can access the page.

Once navigated to that route, we send a message to our logged-in user. The message contains:

- Their username: `req.user.username` (further explained above when creating the *user.js* schema file)
- Their session ID: `req.sessionID`
- The maximum age of the session cookie: `req.session.cookie.maxAge`
- A link to log out of the application: `/logout`
- A link to a members-only page: `/secret`

Our secret page route is also only accessible to logged-in users.

```
app.get('/secret', connectEnsureLogin.ensureLoggedIn(), (req, res
  res.sendFile(__dirname + '/static/secret-page.html');
});
```

Finally, we create our route to log out users.

```
app.get('/logout', function(req, res) {
  req.logout();
  res.redirect('/login');
});
```

- `req.logout` : logs out user
- `req.redirect` : Redirects the user to another page. In this case, the user gets redirected to /login.

Add a login POST route

Our application contains one POST route, */login*. We are ready to make the authentication process work by creating this route.

```
app.post('/login', passport.authenticate('local', { failureRedire
    console.log(req.user)
    res.redirect('/dashboard');
});
```

In our route handler we call `passport.authenticate` , to which we pass 2 parameters:

- `'local'` : The primary strategy we are using
- `failureRedirect` : If the user does not authenticate, they get redirected to the homepage.

If the user fails to authenticate, they are sent back to the homepage. If they do authenticate, the code inside the callback will execute:

- `console.log(req.user)` : When a user authenticates, user properties are attached to `req.user` . When you test this application, we will see those properties in your terminal: ID,

username, salt, and hash. Depending on how you design your User, these properties can differ.

- `res.redirect('/dashboard')` : Once the user authentications, they are redirected to */dashboard*.

Assign a port for the application

We are ready to finish *server.js* by assigning a port.

```
// assign port
const port = 3000;
app.listen(port, () => console.log(`This app is listening on port
```

Our completed *server.js* file should look like the code below:

```
const express = require('express'); // server software
const bodyParser = require('body-parser'); // parser middleware
const session = require('express-session');  // session middleware
const passport = require('passport');  // authentication
const connectEnsureLogin = require('connect-ensure-login');// auth

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

const app = express();

// Configure Sessions Middleware
app.use(session({
  secret: 'r8q,+&1LM3)CD*zAGpx1xm{NeQhc;#',
  resave: false,
  saveUninitialized: true,
  cookie: { maxAge: 60 * 60 * 1000 } // 1 hour
}));

// Configure Middleware
app.use(bodyParser.urlencoded({ extended: false }));
app.use(passport.initialize());
app.use(passport.session());
```

```
passport.use(User.createStrategy());

// To use with sessions
passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());

// Route to Homepage
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/static/index.html');
});

// Route to Login Page
app.get('/login', (req, res) => {
  res.sendFile(__dirname + '/static/login.html');
});

// Route to Dashboard
app.get('/dashboard', connectEnsureLogin.ensureLoggedIn(), (req, r
  res.send(`Hello ${req.user.username}. Your session ID is ${req.s
  and your session expires in ${req.session.cookie.maxAge}
  milliseconds.<br><br>
  <a href="/logout">Log Out</a><br><br><a href="/secret">Members C
});

// Route to Secret Page
app.get('/secret', connectEnsureLogin.ensureLoggedIn(), (req, res)
  res.sendFile(__dirname + '/static/secret-page.html');
});

// Route to Log out
app.get('/logout', function(req, res) {
  req.logout();
  res.redirect('/login');
});

// Post Route: /login
app.post('/login', passport.authenticate('local', { failureRedirec
    console.log(req.user)
    res.redirect('/dashboard');
});

// assign port
```

Test the code

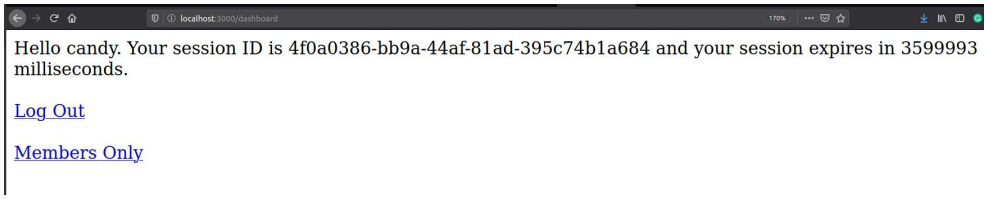
Our project is complete, and we are ready to test the code. Go to your terminal, navigate to the main project folder (login), and start the server.

```
node server.js
```

In your web browser, go to http://localhost:3000/. You should see the home page.

Click on the *Please Login Here!* link to navigate to the login page.

Now enter valid login credentials. Then click the login button to navigate to */dashboard*.



Next, click the *Members Only* link to navigate to /secret.

Finally, click the *Log Out* link to make sure you don't have access to /dashboard or /secret when you are not logged in. You can also make sure you are redirected to the homepage when you enter incorrect login details.

Recap

Passport.js is a widely used authentication middleware that facilitates the creation of applications. Developers have a wide assortment of authentication strategies to choose from, suiting most applications' needs. In this project, we used passport-local to verify usernames and passwords locally.

For our database, we used MongoDB, a popular NoSQL database that stores data in JSON documents. To simplify data modeling, we used Mongoose to handle NodeJS and MongoDB interaction. There are hundreds of specific Passport strategy modules available, and we chose to use passport-local-mongoose.

To use Passport with ExpressJS, you must:

- Connect to the Database
- Create a User model
- Configure the appropriate middleware
- Configure the Passport strategy
- Call the function `passport.authenticate` in your login POST route

Once authenticated, `req.user` is created, allowing you to access user properties.

Further your understanding

- Besides username and password, what other user properties might you want to use?
- What are the advantages and disadvantages of using an ODM or ORM?
- Why isn't it advisable to store user data in a server file?

Additional resources

- Passport.js Documentation (passportjs.org)
- MongoDB Documentation (mongodb.com)
- Mongoose Documentation (mongoosejs.com)
- Passport-Local Mongoose (npmjs.com)

to gain instant access to our entire library.

Was this helpful? Yes No

« Set Up ExpressJS Session Authentication for Node Applications

Back to Contents

Data Brokering with Node.js

- ETL: Transform Data with Node.js
- What Is the Node.js fs (File System) Module?
- Read/Write JSON Files with Node.js
- ETL: Load Data to Destination with Node.js
- What Is a Node.js Stream?
- Use Streams to Extract, Transform, and Load CSV Data
- Stream to an HTTP Response with Node.js
- Overview: How to Manage Node.js Locally
- Install Node.js Locally with Node Version Manager (nvm)
- How to Use Environment Variables in Node.js
- Set up and Test a Dot Env (.env) File in Node
- What Is an API Proxy?
- What Is the Express Node.js Framework?
- How to Add a Route to an Express Server in Node.js
- Express Middleware in Node.js
- Use Express to Create an API Proxy Server in Node.js
- How to Set up an Express.js Server in Node.js
- Organize Your Node.js Code into Modules
- Set up Routes for Your API in Node.js
- Optimize an Express Server in Node.js
- Add Compression to Express in Node.js
- Add Response Caching to a Node.js Express Server
- How the Event Loop Works in Node.js
- Explore the Timers Phase of Node's Event Loop
- Explore the I/O Callbacks Phase of the Node.js Event Loop
- Explore the Immediate Callbacks Phase of Node's Event Loop
- What Is the Difference Between Authorization and Authentication?
- What Are Form Validation and Sanitization?
- Process a User Login Form with ExpressJS
- How to Validate and Sanitize an ExpressJS Form

- What Is the Difference Between Sessions and JSON Web Tokens (JWT) Authentication?
- Set Up ExpressJS Session Authentication for Node Applications

> Authenticate Users With Node ExpressJS and Passport.js