Handling Authentication in NodeJS (Express) with Passport Part 1 — Project Setup

Sijuade Ajagunna
May 29, 2020 · 4 min read

It is what it says it is

This is intended to be a series of articles to help you set up authentication and authorization in your server-side NodeJS applications using PassportJS. We'll start with the Local Strategy (The common username and password method of authentication), and then move on to other Strategies (such as signing up and logging in via Twitter and other popular services).

What we'll cover over this series:

- Setting up your Express app to use ES6 imports
- Setting up and connecting a MongoDB database to your app
- Creating Mongoose Schemas
- Passport Local Strategy
- Creating and Handling JSON Web Tokens (JWTs)

- Passport Google and Twitter strategies
- Handling cookies and protected routes
- Handling Errors

**Note that these are not necessarily in order.*

In this project, we'll set up our project to make use of JavaScript ES6 imports and add some error handling middleware.

What is Passport?

In the simplest terms, Passport is a middleware that handles authentication (registration and logging in) and authorization (access to protected routes) for your application. Other than with these two functions, Passport does not interfere with the functionalities of your app — In other words, it is "unobstructive".

Passport handles authentication in two ways: Session and server-side sessions. We'll be making use of JWTs in this article. To learn more about the difference between these two, you can check this Stack Overflow thread.

Let's get started:

Your Toolkit

You should have NodeJS installed on your computer. You can follow this guide to set it up. You should also have MongoDB installed (or you can create a Mongoose Atlas account instead).

Folder Setup

Create your project folder, `cd` into it and initialize a new package with `npm init -y`. Your folder should contain a package.json file. Create a `src` folder and follow the folder structure below.

```
├──src
    ├──db
    ├──controllers
    ├──helpers
    ├──middleware
    ├──models
```

```
└──routes
├──models
└──services
index.js
app.js
```

You'll probably agree with me that `import` and `export` look better than `require` and `modules.exports`, so let's set up Babel so our app can use ES6 imports syntax.

In your terminal, (make sure you're in the root folder this time and not the src folder), run the following commands:

```
npm i @babel/register @babel/runtime

npm i -D @babel/cli @babel/core @babel/node @babel/preset-env
```

Still in the root folder, create a `.babelrc` file and paste the following code in it

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "useBuiltIns": "usage",
        "corejs": 3
      }
    ]
  ]
}
```

With babel correctly configured, it's time to do some work in the `src` folder.

First, we need to install some more dependencies:

```
npm i express cookie-parser cross-env debug dotenv morgan nodemon
```

And then some error handling

We'll write a function to catch asynchronous errors, another to handle errors and error messages depending on our environment, and then we'll write some classes that extend the JavaScript Error class to make our error messages more specific.

Catch Async Errors

Create a new file in your middleware folder named `catchAsync.js` and copy the following code into it.

```
export default catchAsync => async (request, response, next) => {
  try {
    await catchAsync(request, response, next);
  } catch (error) {
    return next(error);
  }
};
```

catchAsync.js hosted with ♥ by GitHub view raw

In the same middleware folder, create a `errorHandler.js` file and add this code:

```
import { config } from "dotenv";
import debug from "debug";

config();

const DEBUG = debug("dev");

export default (err, request, response, next) => {
  const isProduction = process.env.NODE_ENV === "production";
  let errorMessage = {};

  if (response.headersSent) {
    return next(err);
  }

  if (!isProduction) {
    DEBUG(err.stack);
    errorMessage = err;
  }

  return response.status(err.statusCode || 500).json({
    status: "error",
    error: {
      message: err.message,
```

```
      ...(err.errors && { errors: err.errors }),
      ...(!isProduction && { trace: errorMessage }),
    },
  });
};
```

errorHandler.js hosted with ♥ by GitHub view raw

And finally, for our error classes, create a `errors.js` file in the helpers folder. It should look like this

```
export class ApplicationError extends Error {
  constructor(statusCode, message = "an error occurred", errors) {
    super(message);
    this.statusCode = statusCode || 500;
    this.message = message;
    this.errors = errors;
  }
}

export class NotFoundError extends ApplicationError {
  constructor(message) {
    super(404, message || "resource not found");
  }
}
```

errors.js hosted with ♥ by GitHub view raw

For now, we only have two error classes but we can increase it if needed.

Then, copy the following code into your app.js file.

```
import express from "express";
import logger from "morgan";
import { config } from "dotenv";
import errorHandler from "./middleware/errorHandler";
import { NotFoundError } from "./helpers/errors";

config()
const app = express();

if (["development", "production"].includes(process.env.NODE_ENV)) {
  app.use(logger("dev"));
}

app.get("/", (_, res) => {
```

```
  res.status(200).json({
    status: "success",
    message: "Bonjour, Welcome, E Kaabo",
  });
});

app.all("*", (_, res) => {
  throw new NotFoundError('Resource not found on this server')
});

app.use(express.json());
app.use(express.urlencoded({ extended: false }));

app.use(errorHandler);

export default app;
```

app.js hosted with ♥ by GitHub view raw

Finally, in our index.js file, we'll import our app.js file and create an HTTP server.

```

import http from "http";
import debug from "debug";
import { config } from "dotenv";
import app from "./app";

config();

const DEBUG = debug("dev");
const PORT = process.env.PORT || 5000;

const server = http.createServer(app);

process.on("uncaughtException", (error) => {
  DEBUG(`uncaught exception: ${error.message}`);
  process.exit(1);
});

process.on("unhandledRejection", (err) => {
  DEBUG(err);
  DEBUG("Unhandled Rejection:", {
    name: err.name,
    message: err.message || err,
  });
  process.exit(1);
});
```

```
27
28  server.listen(PORT, () => {
29    DEBUG(
30      `server running on http://localhost:${PORT} in ${process.env.NODE_ENV} mode`
31    );
32  });
```

index.js hosted with ♥ by GitHub view raw

Now, create a `.env` file in your root folder and set up your environment variables.

```
PORT=5000
NODE_ENV=development
```

Next come The Scripts! No, not multiple versions of the band but your package.json scripts to run your app. We'll be creating `dev`, `start`, `build`, `clean` and `build-babel` scripts to enable us run our app in a development or production environment.

In your package.json file, add the following commands to the scripts property:

```
"build": "npm run clean && npm run build-babel",
"build-babel": "babel -d ./build ./src -s",
"clean": "rm -rf .nyc_output build coverage && mkdir build",
"dev": "cross-env DEBUG=dev nodemon --exec babel-node src/index.js",
"start": "node ./build/index.js",
```

- The `dev` script runs your app in development mode, which is what we'll be using.
- The build script compiles your app into the build folder, transforming ES6+ code into a backwards-compatible version of JavaScript in current and older environments.
- The start script starts your compiled NodeJS app.

In the second article, we'll integrate a MongoDB database and set up PassportJS in our app.

This project is available on GitHub, you can also follow me on Twitter.

Nodejs Passportjs Express Authorization JavaScript