Handling Authentication in NodeJS (Express) with Passport Part 2 — MongoDB and Passport

Sijuade Ajagunna
May 29, 2020 · 3 min read

So far, we've created our project boilerplate to make development easier for us. In this part, we'll set up our database and create an authentication and authorization system.

The database…

We'll be setting up a MongoDB database using the popular Mongoose ORM. First, let's install the dependency:

```
npm i mongoose bcrypt jsonwebtoken validator
```

We'll need each of those packages for various things while we manage entries in our database. Bcrypt is useful for hashing passwords, JSON Web Tokens are will serve as our means of authentication when communicating with the server and validator will be used for validating data to be saved to our database.

Creating a Mongoose connection

First, in your `.env` file, add a new key titled `DEV_DB`. This will serve as our MongoDB database name. MongoDB databases use the MongoDB protocol, so a to create a database named 'passport_auth_dev', your `DEV_DB` value will look somewhat like this:

```
DEV_DB=mongodb://localhost:27017/passport_auth_dev
```

In the **db** folder, create a mongoose.js file, and set up a mongoose connection so:

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

config();
const DEBUG = debug("dev");

const { NODE_ENV, DEV_DB } = process.env;

const options = {
  useNewUrlParser: true,
  useCreateIndex: true,
  useFindAndModify: false,
  useUnifiedTopology: true,
};

mongoose
  .connect(DEV_DB, options)
  .then(() => {
    DEBUG("MongoDB is connected");
  })
  .catch((err) => {
    DEBUG("MongoDB connection unsuccessful");
    DEBUG(err);
  });
```

mongoose.js hosted with ♥ by GitHub view raw

Import your mongoose file into your index.js and restart your app, you'll know if your database is successfully created via the message on your console.

```
import './db/mongoose'
```

Creating a collection schema

We'll only be creating a User collection in this project with 3 fields: email, userName and password.

In the model folder, create a user.model.js file and update it with this code:

```
import mongoose from "mongoose";
import bcrypt from "bcrypt";
import jwt from "jsonwebtoken";
import { config } from "dotenv";
import validator from "validator";

config();

const jwtPrivateSecret = process.env.JWT_PRIVATE_SECRET.replace(/\\n/g, "\n");

const userSchema = new mongoose.Schema({
  email: {
    type: String,
    validate: [validator.isEmail, "Please provide a valid email address"],
    required: [true, 'Email is required'],
    unique: true

  },
  userName: {
    type: String,
    required: [true, 'userName is required'],
    unique: true
  },
  password: {
    type: String,
    required: [true, "password is required"],
    minlength: 8,
  },
});

userSchema.pre("save", async function (next) {
  if (!this.password || !this.isModified("password")) return next;

  this.password = await bcrypt.hash(
    this.password,
    parseInt(process.env.HASH)
  );
  next();
});
```

```
40
41  userSchema.methods.toJSON = function () {
42    const user = this;
43
44    const userObj = user.toObject();
45    delete userObj.password;
46    return userObj;
47  };
48
49  userSchema.methods.comparePassword = async function (password) {
50    return await bcrypt.compare(password, this.password);
51  };
52
53  userSchema.methods.generateVerificationToken = function () {
54    return jwt.sign({ id: this._id }, jwtPrivateSecret, {
55      expiresIn: "10d",
56      algorithm: "RS256",
57    });
58  };
59
60  userSchema.statics.checkExistingField = async (field, value) => {
61      const checkField = await User.findOne({ [`${field}`]: value });
62
63      return checkField;
64  };
65
66  const User = mongoose.model("User", userSchema);
67
68  export default User;
```

You'll need to update your `.env` file with three more keys:

- A HASH key to indicate how much time bcrypt should take while creating a hash. A higher number makes increases the security of your passwords but can adversely affect your server response time. 10 is acceptable.
- JWT_PUBLIC_SECRET and JWT_PRIVATE_SECRET: There are different algorithms for generating JWTs, I'm using the RS256, which may be a bit more complicated as it requires a private and public RSA key pair (Note which key is public and which is private). You can generate a key pair here. Copy both keys to an editor, e.g., VSCode, Use the find and replace(`ctrl+F`) functionality to replace ALL newline characters(`ctrl+Enter` in VSCode)with `\n` , and assign to the correct JWT key.

Passport

We're ready to integrate PassportJS into our app. But first, some *more* dependencies:

```
npm i passport passport-jwt passport-local
```

Now, let's get started.

In the services folder, let's create a subfolder named passport. This subfolder should have two files: `passport-local.js` and `config.js`. The first file handles our signup and login functionalities while the second contains the settings for retrieving and decoding our JWT.

```
1  import debug from 'debug';
2  import { Strategy } from 'passport-local';
3  import passport from 'passport';
4
5  import User from '../../models/user.model';
6
7  const DEBUG = debug('dev');
8
9  const authFields = {
10   usernameField: 'email',
11   passwordField: 'password',
12   passReqToCallback: true,
13 };
14
15 passport.use(
16   'login',
17   new Strategy(authFields, async (req, email, password, cb) => {
18     try {
19       const user = await User.findOne({
20         $or: [{ email }, { userName: email }],
21       });
22
23       if (!user || !user.password) {
24         return cb(null, false, { message: 'Incorrect email or password.' });
25       }
26
27       const checkPassword = await user.comparePassword(password);
28
29       if (!checkPassword) {
30         return cb(null, false, { message: 'Incorrect email or password.' });
```

```
      }
      return cb(null, user, { message: 'Logged In Successfully' });
    } catch (err) {
      DEBUG(err);
      return cb(null, false, {statusCode: 400, message: err.message});
    }
  }),
);

passport.use(
  'signup',
  new Strategy(authFields, async (req, email, password, cb) => {
    try {
    const checkEmail = await User.checkExistingField('email', email);

      if (checkEmail) {
        return cb(null, false, {
          statusCode: 409,
          message: 'Email already registered, log in instead',
        });
      }

    const checkUserName = await User.checkExistingField('userName', req.body.userName);
      if (checkUserName) {
        return cb(null, false, {
          statusCode: 409,
          message: 'Username exists, please try another',
        });
      }

      const newUser = new User();
      newUser.email = req.body.email;
      newUser.password = req.body.password;
      newUser.userName = req.body.userName;
      await newUser.save();
      return cb(null, newUser);
    } catch (err) {
        DEBUG(err)
      return cb(null, false, {statusCode: 400, message: err.message});
    }
  }),
);
```

You can set a 'username' and 'password' field if yours, for some reason, won't be the same as the default. Our 'username' field is the 'email' in our request body. Our users can sign in via userName or email, so while the 'username' field remains 'email', we can configure our passport-local strategy to look through both fields in our database for the user's data.

Passport error callbacks have 3 arguments, the third of which is the 'info'. This is an object which can transfer information such as status code or error messages to the next middleware in the stack or to your error handler.

```
import passport from 'passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { config } from 'dotenv';
import User from '../../models/user.model';

config()

const jwtPublicSecret = process.env.JWT_PUBLIC_SECRET.replace(/\\n/g, '\n');

const cookieExtractor = req => {
  let token = null;
  if (req && req.cookies.jwt) {
    token = req.cookies.jwt;
  }

  return token;
};

const options = {
  secretOrKey: jwtPublicSecret,
  algorithms: ['RS256'],
  passReqToCallback: true,
};

options.jwtFromRequest = ExtractJwt.fromExtractors([
  ExtractJwt.fromAuthHeaderAsBearerToken(),
  req => cookieExtractor(req),
]);

passport.use(
  new Strategy(options, (req, jwtPayload, done) => {
    User.findOne({ _id: jwtPayload.id })
      .then(user => {
        if (user) {
          delete user._doc.password;
```

```
            done(null, user);
          } else {
            done(null, false);
          }
        })
        .catch(err => {
          if (err) {
            return done(err, false);
          }
        });
    }),
  );

  export default passport;
```

config is hosted with ♥ by GitHub view raw

Passport provides us with a number of ways to extract JWTs from request headers, in this case, we combine two different ways — `ExtractJWT.fromAuthHeaderAsBearerToken()` which extracts the token when it's prefixed with 'Bearer '(note the space), and a custom Cookie Extractor function which extracts the token from a named token. You can check out other ways of extracting tokens in the passport-jwt documentation.

The cookie is parsed by the cookie-parser package, hence it should be added to the express middleware stack. You should also initialize Passport in the app.js file so your app.js should look like this now:

```
import express from "express";
import logger from "morgan";
import { config } from "dotenv";
import passport from 'passport';
import cookieParser from 'cookie-parser';
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.use(express.json());
```

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

passport.initialize()

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

We are yet to check if our configuration works and we'll complete the Passport local strategy in the next article by creating our authentication middleware, controllers and routes.

The project repository is available on GitHub, you can follow me on the bird app too while at it.

Passportjs Mongodb Nodejs Authentication Authorization