Hashing and JWTs with Node

Springboard

Download Demo Code

Goals

- Hash passwords with Bcrypt
- Using JSON web tokens for API authentication
- Use middleware to simplify route security

Password Hashing with Bcrypt

Similar to Flask, but with asynchronous API.

To use, install library:

```
$ npm install bcrypt
```

Import bcrypt:

```
const bcrypt = require("bcrypt");
```

bcrypt.hash(*password-to-hash*, *work-factor*)

Hash password, using work factor (12 is a good choice).

Returns *promise* — resolve to get hashed password.

bcrypt.compare(*password*, *hashed-password*)

Check if password is valid.

Returns *promise* — resolve to get boolean.

Hashing Password

demo/auth-api/routes/auth.js

```
/** Register user.
 *   {username, password} => {username} */

router.post("/register", async function (req, res, next) {
  try {
    const { username, password } = req.body;
    const hashedPassword = await bcrypt.hash(
      password, BCRYPT_WORK_FACTOR);
    const result = await db.query(
      `INSERT INTO users (username, password)
             VALUES ($1, $2)
             RETURNING username`,
      [username, hashedPassword]);

    return res.json(result.rows[0]);
  } catch (err) {
    return next(err);
  }
});
```

Logging in

- Try to find user first
  - If exists, compare hashed password to hash of login password
- ***bcrypt.compare()*** resolves to boolean—if ***true***, passwords match!

demo/auth-api/routes/auth.js

```
/** Login: returns {message} on success. */

router.post("/login-1", async function (req, res, next) {
  try {
    const { username, password } = req.body;
    const result = await db.query(
      `SELECT password FROM users WHERE username = $1`,
      [username]);
    const user = result.rows[0];

    if (user) {
      if (await bcrypt.compare(password, user.password) === true) {
        return res.json({ message: "Logged in!" });
      }
    }
    throw new ExpressError("Invalid user/password", 400);
  } catch (err) {
    return next(err);
  }
});
```

JSON Web Tokens

Authentication in Flask

- Make request with username/password to login route
- Server authenticates & puts user info session
  - Session is encoded & signed with Flask-specific scheme
- Session info is sent back to browser in cookie
- Session info is automatically resent with every request via cookie

- This works well for traditional web apps & is straightforward to do
- What if
  - We didn't want to send auth info with certain requests?
  - We wanted to share authentication info across multiple APIs / hostnames?
- We'll use a more API-server friendly process!

Authentication Via Tokens

For our Express API apps, we'll handle authentication differently:

- Make request with username/password to AJAX login route
- Server authenticates & returns token in JSON
  - Token is encoded & signed with open standard, "JSON Web Token"
- Front-end JavaScript receives token & stores *(in var or localStorage)*
- For every future request, browser sends token in request
  - Server gets token from request & validates token

JSON Web Tokens

Homepage of JSON Web Tokens

JWTs are an open standard and are implemented across technology stacks, making it simple to share tokens across different services.

JWTs can store any arbitrary "payload" of info, which are "signed" using a secret key, so they can be validated later (similar to Flask's session).

The JWT token itself is a string comprising three parts:

- **Header:** metadata about token *(signing algorithm used & type of token)*
- **Payload:** data to be stored in token *(typically an object)*
  - Often, this will store things like the user ID
  - This is *encoded*, not *encrypted* — don't put secret info here!
- **Signature:** version of header & payload, signed with secret key
  - Uses algorithm specified in header *(we'll use default, "HMAC-SHA256")*

Note: JWTs Versus Flask sessions

JWTs do the same process as a Flask session: encode the payload and sign it using a secret key. Flask's built-in session uses a Flask-specific encoding and signing algorithm, but there are add-on products for Flask to use JWTs as the encoding/signing scheme for sessions.

The bigger difference is in how this is transmitted: Flask's standard sessions are transmitted via cookies, so they are passed automatically between the server and the browser. The JWT standard isn't involved itself with when a how tokens are sent — this is up to the application developer. We'll be doing so by sending these in the request manually, and retrieving them manually from the request in the server.

None of this is inherently specific to Flask or Express — there are cookie-based authentication add-ons for Express, and there are JWT libraries for Python, so Flask could emit JWTs for API-based server.

For more information, here's a good discussion of JWTs versus server-side sessions.

Using JWTs

Install JSON web token:

```
$ npm install jsonwebtoken
```

Creating Tokens

jwt.sign(*payload*, *secret-key*, *jwt-options*)

- ***payload***: object to store as payload of token
- ***secret-key***: secret string used to "sign" token
- ***jwt-options*** is optional object of settings for making the token

This returns the token (a string)

```
const jwt = require("jsonwebtoken");

const SECRET_KEY = "oh-so-secret";
const JWT_OPTIONS = { expiresIn: 60 * 60 };  // 1 hour

let payload = {username: "jane"};
let token = jwt.sign(payload, SECRET_KEY, JWT_OPTIONS);
```

Decoding / Verifying Tokens

jwt.decode(*token*)

Return the payload from the token (works *without secret key*. Remember, the tokens are signed, not enciphered!)

jwt.verify(*token*, *secret-key*)

Verify token signature and return payload is valid. If not, raise error.

```
jwt.decode(token);                // {username: "jane"}

jwt.verify(token, SECRET_KEY);    // {username: "jane"}

jwt.verify(token, "WRONG");       // error!
```

Using JWTs in Express

Login

demo/auth-api/routes/auth.js

```
/** (Fixed) Login: returns JWT on success. */

router.post("/login", async function (req, res, next) {
  try {
    const { username, password } = req.body;
    const result = await db.query(
      "SELECT password FROM users WHERE username = $1",
      [username]);
    let user = result.rows[0];

    if (user) {
      if (await bcrypt.compare(password, user.password) === true) {
        let token = jwt.sign({ username }, SECRET_KEY);
        return res.json({ token });
      }
    }
    throw new ExpressError("Invalid user/password", 400);
  } catch (err) {
    return next(err);
  }
});
```

Protected Routes

After client receives token, they should send with every future request that needs authentication.

For our demo, we'll look in ***req.body*** for a token called ***_token***

Front End JS

```
// get token from login route
let resp = await axios.post(
    "/login", {username: "jane", password: "secret"});
let token = resp.data;

// use that taken for future requests
await axios.get("/secret", {params: {_token: token}});
await axios.post("/other", {_token: token});
```

Verifying a token

demo/auth-api/routes/auth.js

```
/** Secret-1 route than only users can access */

router.get("/secret-1", async function (req, res, next) {
  try {
    // try to get the token out of the body
    const tokenFromBody = req.body._token;

    // verify this was a token signed with OUR secret key
    // (jwt.verify raises error if not)
    jwt.verify(tokenFromBody, SECRET_KEY);

    return res.json({ message: "Made it!" });
  }

  catch (err) {
    return next({ status: 401, message: "Unauthorized" });
  }
});
```

That works, but can we refactor this?

- We don't want to repeat this one every route
- How can we intercept the request and verify the token?
- **Middleware!**

Middleware

Authentication Middleware

demo/auth-api/middleware/auth.js

```
/** Auth JWT token, add auth'd user (if any) to req. */

function authenticateJWT(req, res, next) {
  try {
    const tokenFromBody = req.body._token;
    const payload = jwt.verify(tokenFromBody, SECRET_KEY);
    req.user = payload;
    return next();
  } catch (err) {
    // error in this middleware isn't error -- continue on
    return next();
  }
}
```

(Stores user data on ***req*** for later requests)

Using Middleware on All Routes

demo/auth-api/app.js

```
const express = require("express");
const app = express();
const routes = require("./routes/auth");
const ExpressError = require("./expressError");
const { authenticateJWT } = require("./middleware/auth");

app.use(express.json());
app.use(authenticateJWT);
```

Middleware runs on all routes defined after this line.

Authorization Middleware

demo/auth-api/middleware/auth.js

```
/** Require user or raise 401 */

function ensureLoggedIn(req, res, next) {
  if (!req.user) {
    const err = new ExpressError("Unauthorized", 401);
    return next(err);
  } else {
    return next();
  }
}
```

We can have more specific authorization requirements.

Here's a version that requires the username be "admin":

demo/auth-api/middleware/auth.js

```
/** Require admin user or raise 401 */

function ensureAdmin(req, res, next) {
  if (!req.user || req.user.username != "admin") {
    const err = new ExpressError("Unauthorized", 401);
    return next(err);
  } else {
    return next();
  }
}
```

Note: Better version

This is a simple check, useful for just understanding the idea that you might have more specialized authorization middleware.

A more realistic implementation of "make they're admins" might use an ***is_admin*** field in the user table, and the JWT would include that field along with the username. Then, this middleware would check whether that value is in the JWT.

Using Middleware on Specific Routes

You typically need flexibiity in route authorization; not just "all".

Some routes, like */register* and */login*, need to be open

demo/auth-api/routes/auth.js

```
/** Secret route: only users can access */

router.get("/secret",
  ensureLoggedIn,
  async function (req, res, next) {
    try {
      return res.json({ message: "Made it!" }
    } catch (err) {
      return next(err);
    }
  });
```

demo/auth-api/routes/auth.js

```
/** Secret-admin route: only admins can access */

router.get("/secret-admin",
  ensureAdmin,
  async function (req, res, next) {
    try {
      return res.json({ message: "Made it!" });
    } catch (err) {
      return next(err);
    }
  });
```

You can add as many middleware functions as you want

Common Configuration

- As application scales, variables like ***SECRET_KEY*** used all over.
  - Don't redefine in every file — tedious and bug-prone!
- Create a file, ***config.js***, at app route, and export vars from there

demo/auth-api/config.js

```
/** Common settings for auth-api app. */

const DB_URI = (process.env.NODE_ENV === "test")
  ? "postgresql:///express_hashing_jwts_test"
  : "postgresql:///express_hashing_jwts";

const SECRET_KEY = process.env.SECRET_KEY || "secret";

const BCRYPT_WORK_FACTOR = 12;

module.exports = {
  DB_URI,
  SECRET_KEY,
  BCRYPT_WORK_FACTOR
};
```

Testing Auth

- Before each request, create test users and dervice tokens for them
- Store these tokens in global variables that can be used across tests

Before hook

demo/auth-api/routes/auth.test.js

```
let testUserToken;
let testAdminToken;

beforeEach(async function () {
  const hashedPassword = await bcrypt.hash(
    "secret", BCRYPT_WORK_FACTOR);
  await db.query(`INSERT INTO users VALUES ('test', $1)`,
    [hashedPassword]);
  await db.query(`INSERT INTO users VALUES ('admin', $1)`,
    [hashedPassword]);

  // we'll need tokens for future requests
  const testUser = { username: "test" };
  const testAdmin = { username: "admin" };
  testUserToken = jwt.sign(testUser, SECRET_KEY);
  testAdminToken = jwt.sign(testAdmin, SECRET_KEY);
});
```

Protected Routes

demo/auth-api/routes/auth.test.js

```
describe("GET /secret success", function () {
  test("returns 'Made it'", async function () {
    const response = await request(app)
      .get(`/secret`)
      .send({ _token: testUserToken });
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ message: "Made it!" });
  });
});
```

demo/auth-api/routes/auth.test.js

```
describe("GET /secret failure", function () {
  test("returns 401 when logged out", async function () {
    const response = await request(app)
      .get(`/secret`); // no token being sent!
    expect(response.statusCode).toBe(401);
  });

  test("returns 401 with invalid token", async function () {
    const response = await request(app)
      .get(`/secret`)
      .send({ _token: "garbage" }); // invalid token!
    expect(response.statusCode).toBe(401);
  });
});
```