Introduction to sessions

> This article was originally published at https://gist.github.com/joepie91/cf5fd6481a31477b12dc33af453f9a1d.

*While a lot of Node.js guides recommend using JWT as an alternative to session cookies (sometimes even mistakenly calling it "more secure than cookies"), this is a terrible idea. JWTs are absolutely **not** a secure way to deal with user authentication/sessions, and this article goes into more detail about that.*

Secure user authentication requires the use of *session cookies*.

Cookies are small key/value pairs that are usually sent by a server, and stored on the client (often a browser). The client then sends this key/value pair back with every request, in a HTTP header. This way, unique clients can be identified between requests, and client-side settings can be stored and used by the server.

Session cookies are cookies containing a unique *session ID* that is generated by the server. This session ID is used by the server to identify the client whenever it makes a request, and to associate *session data* with that request.

Session data is arbitrary data that is stored on the server side, and that is associated with a session ID. The client can't see or modify this data, but the server can use the session ID from a request to associate session data with that request.

Altogether, this allows for the server to store arbitrary data for a session (that the user can't see or touch!), that it can use on every subsequent request in that session. This is how a website remembers that you've logged in.

Step-by-step, the process goes something like this:

1. **Client** requests login page.
2. **Server** sends login page HTML.
3. **Client** fills in the login form, and submits it.
4. **Server** receives the data from the login form, and verifies that the username and password are correct.
5. **Server** creates a new session in the database, containing the ID of the user in the database, and generates a unique session ID for it (which is *not* the same as the user ID!)
6. **Server** sends the session ID to the user as a cookie header, alongside a "welcome" page.
7. **Client** receives the session ID, and saves it locally as a cookie.
8. **Client** displays the "welcome" page that the cookie came with.

9. **User** clicks a link on the welcome page, navigating to his "notifications" page.
10. **Client** retrieves the session cookie from storage.
11. **Client** requests the notifications page, sending along the session cookie (containing the session ID).
12. **Server** receives the request.
13. **Server** looks at the session cookie, and extract the session ID.
14. **Server** retrieves the session data from the database, for the session ID that it received.
15. **Server** associates the session data (containing the user ID) with the request, and passes it on to something that handles the request.
16. **Server request handler** receives the request (containing the session data including user ID), and sends a personalized notifications page for the user with that ID.
17. **Client** receives the personalized notifications page, and displays it.
18. **User** clicks another link, and we go back to step 10.

Configuring sessions

Thankfully, you won't have to implement all this yourself - most of it is done for you by existing session implementations. If you're using Express, that implementation would be express-session.

The `express-session` module doesn't implement the actual session storage itself, it only handles the Express-related bits - for example, it ensures that `req.session` is automatically loaded from and saved to.

For the storage of session data, you need to specify a "session store" that's specific to the database you want to use for your session data - and when using Knex, `connect-session-knex` is the best option for that.

While full documentation is available in the `express-session` repository, this is what your `express-session` initialization might look like when you're using a relational database like PostgreSQL (through Knex):

```
const express = require("express");
const knex = require("knex");
const expressSession = require("express-session");
const KnexSessionStore = require("connect-session-knex")(expressSession);

const config = require("./config.json");

/* ... other code ... */

/* You will probably already have a line that looks something like the below.
 * You won't have to create a new Knex instance for dealing with sessions - you
```

```
 * can just use the one you already have, and the Knex initialization here is
 * purely for illustrative purposes. */
let db = knex(require("./knexfile"));

let app = express();

/* ... other app initialization code ... */

app.use(expressSession({
    secret: config.sessions.secret,
    resave: false,
    saveUninitialized: false,
    store: new KnexSessionStore({
        knex: db
    })
}));

/* ... rest of the application goes here ... */
```

The configuration example in more detail

```
require("connect-session-knex")(expressSession)
```

The `connect-session-knex` module needs access to the `express-session` library, so instead of exporting the session store constructor directly, it exports a *wrapper function*. We call that wrapper function immediately after requiring the module, passing in the `express-session` module, and we get back a session store constructor.

```
app.use(expressSession({
    secret: config.sessions.secret,
    resave: false,
    saveUninitialized: false,
    store: new KnexSessionStore({
        knex: db
    })
}));
```

This is where we 1) create a new `express-session` middleware, and 2) `app.use` it, so that it processes every request, attaching session data where needed.

```
secret: config.sessions.secret,
```

Every application should have a "secret" for sessions - essentially a secret key that will be used to cryptographically sign the session cookie, so that the user can't tamper with it. This should be a *random* value, and it should be stored in a configuration file. You should *not* store this value (or any other secret values) in the source code directly.

On Linux and OS X, a quick way to generate a securely random key is the following command: `cat /dev/urandom | env LC_CTYPE=C tr -dc _A-Za-z0-9 | head -c${1:-64}`

```
  resave: false,
```

When `resave` is set to `true`, `express-session` will *always* save the session data after every request, regardless of whether the session data was modified. This can cause race conditions, and therefore you usually don't want to do this, but with some session stores it's necessary as they don't let you reset the "expiry timer" without saving all the session data again.

`connect-session-knex` doesn't have this problem, and so you should set it to `false`, which is the safer option. If you intend to use a different session store, you should consult the `express-session` documentation for more details about this option.

```
  saveUninitialized: false,
```

If the user doesn't have a session yet, a brand new `req.session` object is created for them on their first request. This setting determines whether that session should be saved to the database, *even* if no session data was stored into it. Setting it to `false` makes it so that the session is only saved if it's actually *used* for something, and that's the setting you want here.

```
  store: new KnexSessionStore({
    knex: db
  })
```

This tells `express-session` where to store the actual session data. In the case of `connect-session-knex` (which is where `KnexSessionStore` comes from), we need to pass in an existing Knex instance, which it will then use for interacting with the `sessions` table. Other options can be found in the `connect-session-knex` documentation.

Using sessions

The usage of sessions is quite simple - you simply set properties on `req.session`, and you can then access those properties from other requests within the same session. For example, this is what a login route might look like (assuming you're using Knex, `scrypt-for-humans`, and a custom `AuthenticationError` created with `create-error`):

```
router.post("/login", (req, res) => {
    return Promise.try(() => {
        return db("users").where({
            username: req.body.username
        });
    }).then((users) => {
        if (users.length === 0) {
            throw new AuthenticationError("No such username exists");
        } else {
            let user = users[0];

            return Promise.try(() => {
                return scryptForHumans.verifyHash(req.body.password, user.hash);
            }).then(() => {
                /* Password was correct */
                req.session.userId = user.id;
                res.redirect("/dashboard");
            }).catch(scryptForHumans.PasswordError, (err) => {
                throw new AuthenticationError("Invalid password");
            });
        }
    });
});
```

And your `/dashboard` route might look like this:

```
router.get("/dashboard", (req, res) => {
    return Promise.try(() => {
        if (req.session.userId == null) {
            /* User is not logged in */
            res.redirect("/login");
        } else {
            return Promise.try(() => {
                return db("users").where({
                    id: req.session.userId
                });
            }).then((users) => {
                if (users.length === 0) {
                    /* User no longer exists */
                    req.session.destroy();
```

```
                res.redirect("/login");
            } else {
                res.render("dashboard", {
                    user: users[0];
                });
            }
        });
    }
  });
});
```

In this example, `req.session.destroy()` will - like the name suggests - destroy the session, essentially returning the user to a session-less state. In practice, this means they get "logged out".

Now, if you had to do all that logic for *every* route that requires the user to be logged in, it would get rather unwieldy. So let's move it out into some middleware:

```
function requireLogin(req, res, next) {
    return Promise.try(() => {
        if (req.session.userId == null) {
            /* User is not logged in */
            res.redirect("/login");
        } else {
            return Promise.try(() => {
                return db("users").where({
                    id: req.session.userId
                });
            }).then((users) => {
                if (users.length === 0) {
                    /* User no longer exists */
                    req.session.destroy();
                    res.redirect("/login");
                } else {
                    req.user = users[0];
                    next();
                }
            });
        }
    });
}
```

```
router.get("/dashboard", requireLogin, (req, res) => {
    res.render("dashboard", {
        user: req.user
    });
});
```

Note the following:

- We now have a separate `requireLogin` function that verifies whether the user is logged in.
- That same function also sets `req.user` if they *are* logged in, with their user data, before calling `next()` (which passes control to the next middleware/route).
- Instead of only specifying a path and a route in the `router.get` call, we now specify our `requireLogin` middleware as well. It will get called before the route, and the route is *only* ever called if the `requireLogin` middleware calls `next()` (which it only does for logged-in users).

Revision #1
Created 11 December 2024 01:22:06 by joepie91
Updated 11 December 2024 18:44:19 by joepie91