User Authentication Part 3

Long Term Session Storage

Sessions So Far

- Our sessions have been holding data that's unique per user
- The session gets loaded when the user presents us the session cookie
- However, our sessions disappear on server restarts
- And our complex user objects don't fully functional, they turn in to simple objects on the next request

Session Storage

- The reason our sessions disappear on restart is they're stored in memory
- This is the same way global variables are stored, they stick around for as long as the program's running, but get cleaned out when the program is done
- In order to have a session stay around, we want to store them in something more long term
- There are tons of options for where to store it (file storage, long term memory, database, cloud server) but we'll stick with our database for now

Session Storage - Which Module

- Most session storage modules are based off of a `connect` module
  - Think of it as a precursor to `express`, a simple HTTP server
- Because of that, the modules we use to save our session are usually prepended with `connect-` instead of `express-`
- This doesn't mean we need to use `connect` though, `express-session` will handle the expressification for us
- The module we'll be using today is `connect-session-sequelize`

```
npm install --save connect-session-sequelize
```

Session Storage - `connect-session-sequelize`

- There's a bit of whacky configuration below but what's happening is:
  - The module provides us a function to generate a SessionStore class
  - That SessionStore class creates an instance, given a config object
  - We pass an object with our Sequelize at the `db` key

```
/* ...Require express and other middlewares... */
const session = require("express-session");
const connectSessionSequelize = require("connect-session-sequelize");
const sql = require("./util/sql");

/* ...Create app, get cookie secret, apply cookie middleware... */
const SessionStore = connectSessionSequelize(session.Store);
app.use(session({
    secret: secret,
    store: new SessionStore({ db: sql }),
}));
```

Session Storage - The Result

- That's all you need to do!
- Now you have a new `Sessions` table in your database that it pulls sessions in from
- Try restarting your server, refreshing, and seeing that your session persists
- But we'll find that if we save a user to the session, it still gets simplified
- And if we changed data in the `users` table, it wouldn't be reflected in our session
- This is because none of the storage options are meant to hold full Javascript class objects, only simple data
- So how do we end up with an up-to-date user object?

Session Storage - (De)Serializing

- When we need to store large objects, we do something known as "serializing"
- This is converting something complicated and stateful into something that can more easily be sent to other services
- It can later be "deserialized" to turn it into its former complex object
- This process is inherently lossy, so rather than relying on the default serialization, we can implement our own using middleware

Session Storage - Saving the User

- Instead of storing the whole user, just store the userid for reference later

```
function login(req, user, password) {
    return user.comparePassword(password).then(function(valid) {
        if (valid) {
            // Save the userid instead of the whole user
            // req.session.user = user;
            req.session.userid = user.get("id");
        }

        return valid;
    });
}
```

Session Storage - Deserialize the User

- Next we'll make a new middleware that deserializes the user and attaches it to req, if they have a userid in session

```
const User = require("../models/user");

function deserializeUserMW(req, res, next) {
    if (req.session.userid) {
        User.findById(req.session.userid).then(function(user) {
            if (user) {
                // Attach directly to req, NOT session
                req.user = user;
            }
            else {
                // If it was a bad userid, remove it from session
                req.session.userid = null;
            }
            next();
        });
    }
    else {
        next();
    }
}
```

Session Storage - Using the User

- Now throughout our app, if they have logged in, you'll have `req.user` available
- This also means you can use the `.get()`, `.update()`, and custom instance methods for the user
- Make sure you update any code that *was* looking for `req.session.user` to check `req.user`
- You'll want to make sure that the `deserializeUserMW` is added *before* any middleware that relies on `req.user`
- Ideally `app.use()` it *after* cookie and session, but *before* your other middleware

Session Storage - Using the User (Example)

```
// POST request for updating your profile
app.post("/profile/:userid", function(req, res) {
    if (!req.user || !req.user.get("id") === req.params.userid) {
        // Return 403 status, error message
    }

    req.user.update({
        username: req.body.username,
        firstName: req.body.firstName,
        firstName: req.body.lastName,
        email: req.body.email,
    })
    .then(function() {
        // Redirect to home, save success message
    })
    .catch(function() {
        // Return 500 status, error message
    });
})
```

Additional Reading

- `connect-session-sequelize` docs - Documents configuration, if you want to customize your session storage
- List of session stores - Not needed for class, but if you're curious of other ways to store the session