User Authentication Part 2

Secure Passwords and Extending Sequelize

User Authentication So Far

- We've set up a user table with some sort of login ID (username, email) and a password
- But our passwords are sitting right there, in plain text, ready to be hacked
- We've seen hashing, but how can we implement that to keep our passwords secure?

Password Encryption via `bcrypt`

- When it comes to storing passwords, it's not important that we know exactly what the password is, just that we can confirm later if something is or is not equal to it
- An extremely thorough and useful module for doing so is `bcrypt`
- It provides us with a way of obscuring passwords for saving, and later comparing a user-submitted password to it later
- The methodology it uses is quite complex, so don't worry if you're not sure how it works, just know that it does

Password Encryption via `bcrypt` (Code)

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

// Take a plain text password, and return a promise
// that resolves with its hashed version.
function hashPassword(password) {
    return bcrypt.genSalt().then(function(salt) {
        return bcrypt.hash(password, salt);
    });
}

// Take a plain text password and a hashed password,
// and return a promise that resolves with a boolean.
function comparePassword(comparePw, hashedPw) {
    return bcrypt.compare(comparePw, hashedPw);
}
```

Saving the Hashed Password

- So now that we have the power to hash and compare, we can do that before saving, and during login with our Users
- But this is prone to mistake; what if we forget to hash it before saving? What if we accidentally double-hash it?
- Instead of manually calling `hashPassword()` and `comparePassword()` around in our code, let's enhance our `User` model to do some of this for us!

Sequelize Hooks

- Our Sequelize models can do more than just define what data is held, we can also manipulate that data on its way to or from the database using **hooks**
- Hooks are kind of like events, for your database
- They let you alter data or trigger actions when certain operations happen in your database
- To handle encrypting our password, we'll manipulate the password during the `beforeCreate` and `beforeUpdate` hooks

Sequelize Hooks (Code)

```
// Hooks know how to deal with waiting for promises, so be sure
// to RETURN a promise for hashing, otherwise the user will be
// inserted in to the database before the hash finishes!
function hashUserPassword(user, options) {
    return hashPassword(user.password).then(function(hashedPw) {
        // Set the user's password to the newly hashed password
        user.password = hashedPw;
    });
}

// Specify hooks in the THIRD argument, model options
const User = sql.define("user", { /* ... */ }, {
    hooks: {
        beforeCreate: hashUserPassword,
        beforeUpdate: hashUserPassword,
    },
});
```

Extending Sequelize Models

- So now that our passwords are *saving* the right way, we need an easy way to compare them
- Rather than having to import our `comparePassword` function everywhere, and grab the password from model instances, what if we could add our own methods?
  - The `Users` model could carry around the comparePassword function for us
  - Better yet, each instance of the model could have a `comparePassword` function that handles getting its own hashed password!

Extending Sequelize - Singleton Methods

- The `Users` model is really just a glorified singleton, so we can attach our own methods to it if we want!

```
// models/user.js
const User = sql.define("user", /* columns and options */);

User.comparePassword = function(comparePw, hashedPw) {
    return bcrypt.compare(comparePw, hashedPw);
};


// Some route elsewhere
const usernameWhere = { where: { username: req.body.username } };

User.findAll(usernameWhere).then(function(user) {
    User.comparePassword(req.body.password, user.get("password")).then(function(valid) {
        // If valid then login, else reject login
    });
});
```

Extending Sequelize - Instance Methods

- Better yet, we can define methods that will be available on each User instance
- They'll be able to use the `this` keyword that refers to the instance itself

```
// models/user.js
const User = sql.define("user", /* columns and options */);

User.prototype.comparePassword = function(pw) {
    return bcrypt.compare(pw, this.get("password"));
}

// Some route elsewhere
const usernameWhere = { where: { username: req.body.username } };

User.findAll(usernameWhere).then(function(user) {
    user.comparePassword(req.body.password).then(function(valid) {
        // If valid then login, else reject login
    });
});
```

Extending Sequelize - Best Practices

- Think of all of the different ways you could extend your models and their instances
  - A model method that handles searching for instance(s)
  - An instance method that get an instance's relationships (Photos, settings, etc.)
  - An instance method to send an email to the user
- Try to keep complex interactions reusable by extending, rather than copy / pasting or defining util functions
  - Use model methods when it's functionality that deals with creating new instances, or managing multiple or unknown instances
  - Use instance methods if it's functionality that relates to one instance of a model

Additional Reading

- Sequelize Hooks
- Extending Sequelize Models
- Bcrypt's documentation
- Bcrypt & Password Security - An Introduction - If you want a better overview of bcrypt and password security, this is a good video