Lab 5: An actual user log in

Before you begin, you'll need:

- node and npm installed and working (and npx, although it can be installed with npm if you don't already have it)
- A means of running a database (e.g. using mongod from the command line)
- Git (optional, but committing to Git is a metric for the coursework)
- Some means of editing code (an IDE such as Atom, VS Code Notepad++ etc.)

These have all been used in previous labs, so you should be good to go.

Ideally, you'll also have done the first lab on React, too.

Overview

For user logins, a number of things are required:

- Some place to store usernames and passwords
- Some way of telling if a user is currently authenticated
- Some form of security (hashing passwords at rest and encryption in transit)
- Something on the website that is only available to logged in users (for testing)

We're going to base it on React, next.js and passport.js. Passport provides a decent amount of functionality and there is no point re-inventing the wheel.

Practical

We're going to base our work on one of the next.js/React examples given here:
https://github.com/vercel/next.js/tree/canary/examples/with-passport

```
npx create-next-app --example with-passport with-passport-app
```

it'll take a while to run, but once it has, take a look at what you got. Start the website with:

```
npm run dev
```

Have a play around with it. Follow the instructions. Sign up, login, look at profile, log out, try to look at profile. Refresh the browser. Don't just use the browser buttons, try urls as well. Look at the developer tools to figure out what is going on.
Some questions for the browser:

- Do you actually have a user "account" here? How do you know?
- Where is the authentication kept?
- What is keeping you authenticated?
- Does authentication persist through browser refresh?
- What makes it go away?
- Can you get rid of it manually?

This "website" doesn't actually have a database, so let's add one.
You'll need to start a database running so that it's ready to receive data. mongod will help.
Remember to set up a folder for the database to live inside.

```
mongod --dbpath .
```

Remember to leave your database running.

Read the ReadMe.md. It will tell you where to add the database. Open that file and prepare to edit.
From index.js in the backend of Lab4, we can see how to add a database for node. Follow the template in user.js to add the code.
Remember, you'll need

```
npm install -s mongodb
```

in order to install mongo. First add the code for creating a user in the database. The code you'll need from Lab4's index.js is:

```
const client = new MongoClient(uri);
  async function run() {
    try {
        var dbo = client.db("mydb");
        await dbo.collection("users").insertOne(user, function(err, res) {
            if (err) {
                console.log(err);
                throw err;
            }
        });
    } finally {
        // Ensures that the client will close when you finish/error
        await client.close();
    }
  }
  run().catch(console.dir);
```

And it goes in user.js, replacing the *users.push(user);* call.
Check it has worked with mongosh (from a previous lab):

```
show dbs
use <mydb>
show collections
db.<myCollection>.find()
```

In the example above, the database is called **mydb** and myCollection is **users**.
You might also want to modify the const user object so that the different fields of the document have names – it'll make it easier to read the database.

Next we want to be able to read the user table and find a user in it. In user.js, there is a function called findUser(). Take a look at that function. If it is successful, it returns the relevant user object. And in password-local.js this object is passed to the validatePassword() function so that the hashed input password can be checked against the hash stored in the database. One way of looking up the relevant document in the database is like this:

```
let user_db = await dbo.collection("users").findOne({"username": username}, function(err, res) {
            if (err) {
                throw err;
            }
        });
```

The surrounding connection to the database can be done in the same way as for the insertOne() code above.

If everything is named correctly and functions return what is expected, you should now have a basic user-login system. If you're stuck, user.js is available on Moodle. There are a few flaws that should be fixed, and a few questions to consider, though:

- What if two users sign up with the same name? Is this allowed? Should it be?
- The password is hashed. But is it secure? What could you do to improve it? Where does this code need to "live"?
- How does the "logout" work?
- What is visible to logged in users? How does the "front end" routing work?
- What more do we need to actually make it a useful login system?
- How could a multi-level authentication system work?

Two users, one username

This should obviously not be allowed, but it is something that requires a database check for signing up. You can simply re-purpose the findOne() code from the user login to check to see if a username already exists in the database prior to signing the user up. There might be a better way (https://www.mongodb.com/docs/manual/core/index-unique/).

Password constraints

https://xkcd.com/936/

signup.js already has a constraint that the password must be entered twice and the second time must be identical to the first. You can build on that. Note that the password is only checked when the submit button is clicked.

Something like: https://www.npmjs.com/package/react-password-checklist might help (although the package may cause issues with building the production-ready version). Note that the form used in signup.js comes from within the editable folder structure (it's defined in the components directory). You will have to edit that to allow onChange().

One possible solution for this is available on Moodle. The files signup.js and form2.js are provided.

Logout

This is easy – it's just a cookie and it gets deleted on logout. Use the developer tools to find it. Look through the code and see if you can find out when a login will naturally expire (without actively deleting the cookie).

When else *should* a login expire? Are there any security arguments for re-authentication for this example and other systems? Are there any times when you have to re-enter your password when you're already logged in?

Visible to logged in users

Look at profile.js. Notice where the redirect is. If you're feeling adventurous, take a closer look at how the redirect works (https://www.npmjs.com/package/swr to retrieve data). To practice your JSX skills, try editing the rendered code so that it displays something a little more well formatted.

For the next task, you'll want to add an input box and submit button to this page, too. You might want to do it as a form and useState:
https://www.w3schools.com/react/react_forms.asp

Adding a profile description and expanding the API

In the default example, each "user" (i.e. document in the database) is simply a username, password hash and salt, as well as a "createdAt" and some unique ids. You can learn a lot by trying to add a "profile", that is, a couple of sentences that a user can define. There are a number of places you need to edit to do this. Use the "createUser" and "findUser" functionality as a basis, and use mongo's findOneAndUpdate() function to ultimately add details to the database.
Places to edit:

- The front end – add a form to the profile page (profile.js). You might even want to create a component.
- The API – add a route to the API (the API is in the pages directory in the project). Mine was called addprofile and the code is on Moodle. Remember that this route should be used to call the api function. Look at the existing code – the /api/signup route already exists. Base your edits on that.
- The database functionality. Put it into user.js. Your new function (maybe called updateUser) will look a lot like findUser but with findOneAndUpdate.

Moodle supplies the 3 files that were edited to give a possible solution.

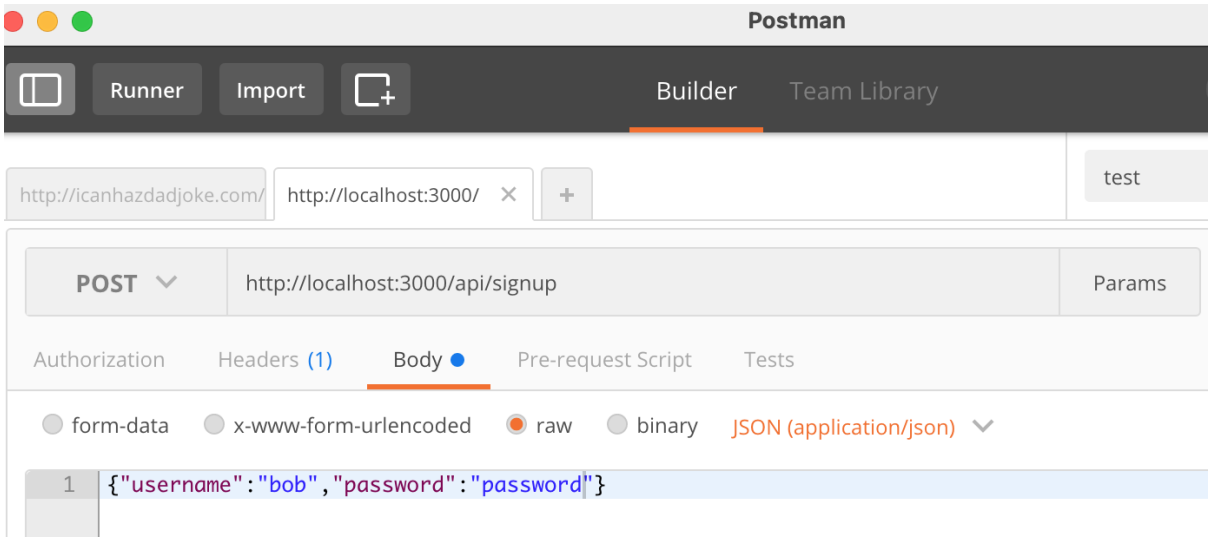
Playing with Postman (or curl)

As you've noticed, this React sample code comes with an API. It's an API that lives on the same server as the code that is destined for browser-rendering. In some cases, that's ok. But it means that we can also access the API independently. Postman will be able to do this if you can formulate the correct HTTP commands. cURL is another possibility if you don't mind using a terminal (or PowerShell).

To generate the correct curl commands, you need to know the API url and the parameters, but other than that, curl should work just the same as your web page but you don't have to fill in all the parameters every time. It is a way to script the testing of your API, or a way you can quickly populate a fresh database with some default users.

Postman used to be a free, stand-alone application. It now wants you to sign up to an account so that you have the "convenience" (and they have your data) of stored activity. You don't need to sign up if you don't want to, and if they start charging, learn to use cURL or look up the python "request" import.

You can use Postman to sign up a user as below. Put a "raw" but JSON formatted body, remember quotes around the strings. In this way, you can test the backend API without the front end, and this might help when debugging. It is also easier to click "Send" on Postman than it is to fill out all your browser interface boxes repeatedly when debugging.


Postman
Runner
Import
Builder
Team Library
http://icanhazdadjoke.com/
http://localhost:3000/
test
POST
http://localhost:3000/api/signup
Params
Authorization
Headers (1)
Body
Pre-request Script
Tests
form-data
x-www-form-urlencoded
raw
binary
JSON (application/json)
1
{"username":"bob","password":"password"}