

* **Assessment 1**
* **Overview**
* **Week 1 - Introduction to NodeJS and MongoDB**
* **Week 2 - Exploring Node, Express and Templates**
* **Week 3 - No Sql data planning and Modelling**
* **Week 4 - Completing the MVC stack**
* **Week 5 - Getting to Grips With the Assessment**
* **Week 6 - Thinking About Updating Data**
* **Week 7 - Data Modeling and Relationships**
* **Week 8 - User Authentication**
  + [Lecture (Start Here)](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/lecture.html)
  + [**Week 8 ( User Authentication)**](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/)
    - [Storing Passwords in a Database](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#storing-passwords-in-a-database)
    - [Theoretical Task](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#theoretical-task)
    - [Representing the User In a Database](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#representing-the-user-in-a-database)
    - [Task 1 - Set up](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#task-1-set-up)
    - [Constructing Our Login Form](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#constructing-our-login-form)
    - [Task 2 - Construct A Login Form](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#task-2-construct-a-login-form)
    - [Creating User Sessions](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#creating-user-sessions)
    - [Protecting Routes](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#protecting-routes)
    - [Task 3](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#task-3)

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#week-8-user-authentication)**Week 8 ( User Authentication)**

**TIP**

**session dependencies** This week we are continuing with our wine tasting theme. Please ensure you have the latest version of the project

You can clone this version using the following command:

git clone https://github.com/joeappleton18/db-starter-project.git --single-branch --branch week-7-solutions

So far, we have been creating new tastings/tasters without attributing them to a user. In reality, a user should probably login before they can post to our database. This week we are going to address this point by answering the following question and sub questions:

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#how-can-i-allow-users-to-register-and-securely-login-to-my-wines-database)**How can I allow users to register and securely login to my wines database?**

* How can I manage users so they remain logged in between sessions?
* How can I allow users to enter a password, while keeping this password hidden from my application?

In addressing the above questions, we are going to consider simple user authentication. We will not cover ideas such as multiple user roles (e.g. super-admin, admin, users). **You should note,** I don't expect you to use multiple role for your assessment. However, you may well acknowledge that in the future multiple roles will be required. Moreover, I do not expect you to use a sophisticated and highly secure authentication process.

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#storing-passwords-in-a-database)**Storing Passwords in a Database**

Graphical user interface

Description automatically generated

The correct, but still somewhat risky, way to store a password

We should **never** store plain text passwords in a database. If this were the case, should a data breach occur, our user's passwords will be freely accessible to the public!

Most users unfortunately use same password across multiple sites and applications. If our site were to reveal an email and password pairing, an attacker could attempt to use this information to login to wider sites 😬. It is vital to avoid this house of cards scenario.

The solution to the above catastrophe is to hash our password. You can think of a hash function as a black box that takes in plain text and outputs a fixed sized unique string. It is this hashed string that we store in the database, **not** the plain text password. Crucially, when we re-run the hash function, with the same password, the same unique string is generated every time. This means to determine if a user password correct, we compare the hash password with the password stored in our database.

It is important to note that not all hash functions are created equal. With some (e.g., MD5), only being marginally better than storing plain text passwords. The problem with hashing functions like MD5 is they have been widely, and naively used, for many years. As such, large tables (rainbow tables or dictionaries) are available that associate MD5 hashes to their plain text equivalent. [As, an experiment, see if you can crack the following hash 8ff32489f92f33416694be8fdc2d4c22 (opens new window)](https://crackstation.net/). We can avoid this type of attack by salting our passwords. This involves adding a string to our password before hashing, we could also perform multiple iterations of our function. However, since MD5 is so fast, our passwords are still susceptible to brute force attacks. In avoiding the vulnerabilities of MD5, we are going to using **bcrypt** to hash our passwords.

bcrypt has the following security features:

* incorporates, by default, a salt to protect against rainbow table
* it's slow - this is good - as it is resistent to brute force attacks
* it's adaptive, allowing the developer to increase or decrease the speed of the function

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#theoretical-task)**Theoretical Task**

Let's play around with some different hashing algorithms and consider the computational expense of each. First, let's install the node libraries form two algorithms (bcrypt and md5):

npm install md5 bcrypt

1

terminal - installing bcrypt and md5 hashing libraries

Next, let's crete a file called testbed.js and we can construct a crude test:

const md5 = require('md5');

const bcrypt = require('bcrypt');

const testBed = async () => {

const testString = "advanced databases";

/\* Begin our test \*/

const start = new Date().getTime();

//const hash = md5(testString);

//const hash = await bcrypt.hashSync(testString, 10);

const end = new Date().getTime();

console.log("the total time taken is: " + ((end - start) / 1000) + 'seconds')

/\* End our test \*/

}

testBed();

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23

testbed.js - to run we can just use `node testbed.js

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#references)**References**

Ntantogian, C., Malliaros, S. and Xenakis, C. (2019). Evaluation of password hashing schemes in open source web platforms. Computers & Security, 84, pp.206–224.

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#representing-the-user-in-a-database)**Representing the User In a Database**

We need to create a user collection to store our users. Let's keep things very simple and represent our users using the following document structure:

{

"\_id" : ObjectId("5fafe79af96f1554d34320ca"),

"email" : "joeappleton18@hotmail.com",

"password" : "$2b$10$ffPZCGmD1SSfKfqlFvbuP.zzA.2UOInT",

"createdAt" : ISODate("2020-11-14T14:20:10.927Z"),

"updatedAt" : ISODate("2020-11-14T14:20:10.927Z"),

}

1  
2  
3  
4  
5  
6  
7

* users collection - user document

So far, we have a very normal looking document in terms of the values that are stored. However, we need a way to hash the password before it is stored in the database. Recall, we are going to use "bcrypt to achieve this. As such, let's install the nodeJS bcrypt library: npm install bcrypt. We can take advantage of the Schema pre hook, [provided by Mongoose (opens new window)](https://mongoosejs.com/docs/api.html#schema_Schema-pre)to hash the user's password before it hits the database. To explore how this works let's create a user model:

const mongoose = require("mongoose");

const { Schema } = mongoose;

const bcrypt = require('bcrypt');

const userSchema = new Schema(

{

email: { type: String, required: [true, 'email is required'], unique: true },

password: { type: String, required: [true, 'password is required'] }

},

{ timestamps: true }

);

userSchema.pre('save', async function (next) {

// logging

console.log(this.password);

try {

const hash = await bcrypt.hash(this.password, 10);

this.password = hash;

next();

} catch (e) {

throw Error('could not hash password');

}

})

module.exports = mongoose.model("User", userSchema);

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25

models/User.js - our user model utilising a pre hook to hash the password

The interesting part to the above code is our userSchema.pre hook. Hooks allow us to, as they sound, hook onto specific actions. Mongoose makes pre and post hooks available to us to facilitate middleware:

[Middleware (also called pre and post hooks) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins.(opens new window)](https://mongoosejs.com/docs/middleware.html)

In the code above, we are hooking onto a Mongoose save event - represented by the first string parameter we pass into the hook function. The second parameter is, the middleware, function that will be run. You must use the long form function (next) here, not (next) =>. This is a nuance of javaScript, allowing us to keep the this keyword bound to to the function itself (sorry this is a bit confusing, but it's hard to capture this idea in a sentence). Notice how we can now access the values that are to be inserted into the database by using this.<property name> format (e.g. this.email, this.password).

Next, we execute the following lines of code to hash our password:

const hash = await bcrypt.hash(this.password, 10);

this.password = hash;

next();

1  
2  
3

Above, we generate a password hash by calling bcrypt.hash(this.password, 10);. Notice, how it's asynchronous, this is because it is slow - which is good. The speed is dictated by rounds of hashing that is run. In this case we are requesting for 10 rounds - represented by the second parameter. As we've seen earlier, on the one hand, increasing this number makes our hashes less susceptible to brute-force search attacks. On the other, the time it takes to generate the hash increases exponentially. Finally, notice how we override the password with its hash value this.password = hash; and then call next();. Next, is passed into our middleware by Mongoose. We must call it, otherwise our application will hang. The result of our middleware, is that the user's password will be stored in a hashed format.

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#task-1-set-up)**Task 1 - Set up**

Use everything you have learnt so far to:

* Set up a /join route, that renders views/create-user.ejs
* Construct a models/User.js (see above), that stores users in a users collection and hashes their passwords using bcrypt.
* Create a controllers/user.js controller and implement a create function that creates a new user
* See if you can now create a new user, redirecting the user to / on a successful registration.

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#constructing-our-login-form)**Constructing Our Login Form**

Currently, our user can join our wine tasting platform and we are able to store their password in a hashed format. To allow the user to login we need to construct a login form (this is easy, it's the same as our join form), and construct the functionality to allow their plain text password to be hashed and compared with the hash stored with their email in the database. This is actually fairly straight forward:

exports.login = async (req, res) => {

try {

const user = await User.findOne({ email: req.body.email });

if (!user) {

res.render('login-user', { errors: { email: { message: 'email not found' } } })

return;

}

const match = await bcrypt.compare(req.body.password, user.password);

if (match) {

console.log('authenticated')

return

}

res.render('login-user', { errors: { password: { message: 'password does not match' } } })

} catch (e) {

return res.status(400).send({

message: JSON.parse(e),

});

}

}

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25

controllers/user.js - using bcrypt to validate a user

Above, we first use our user's email to locate them in the database. We can do this since we have configured email as unique: true incorporates the user schema. Next, we compare the users plain text password with the hash, using await bcrypt.compare(). The compare() function returns true or false, depending on whether the plane text password when hashed matches the hash stored in the database.

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#task-2-construct-a-login-form)**Task 2 - Construct A Login Form**

Use the notes above to create a login form. As per above, on successful login, console.log('authenticated') so you know your functionality works.

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#creating-user-sessions)**Creating User Sessions**

As it stands, a user can sign-up and log-in to our wine tasting application. However, currently, we have no way of persisting the fact that a user has actually logged in. Since HTTP is stateless, this means we can't use the protocol to keep track of users between page requests. As such, to achieve this, we need to use what's know as a user session.

User sessions are implemented using HTTP cookie. You'll be aware that web applications use cookies to store information. Cookies are stored by web browsers and sent to the originating server with each request to this server. We can use this functionality to store a user id in a cookie that will allow us to authenticate the user between requests. Let's consider how this can be done.

We can utilise a npm package called express-session to maintain our user sessions. To install it, run npm install express-session. Next, within app.js we can require and initialise our session:

const expressSession = require("express-session");

app.use(expressSession({ secret: 'foo barr', cookie: { expires: new Date(253402300000000) } }))

1  
2  
3

app.js setting up a session, the secret is an identifier for our application. I've set the cookie expiry data to a distant point in the future

Now we've initiated our cookie, we need to set its value when the user logs in. Let's update our controllers/user.js login method so that it stores the logging in user id to the cookie. Currently, we are just logging to the console, 'authenticated' if the user authenticates, it is this section of code that we need to update:

exports.login = async (req, res) => {

const user = await User.findOne({ email: req.body.email });

...

const match = await bcrypt.compare(req.body.password, user.password);

if (match) {

req.session.userID = user.\_id;

console.log(req.session.userID);

res.redirect('/');

return

}

...

}

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18

controllers/user.js - Assigning the user session to the user\_id. This will allow us to identify the user for future requests

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#storing-our-logged-in-user-details)**Storing our Logged in User Details**

Above, we've explored how to set up session containing the logged in user id. We now need to this information to manage our logged in user within our application. To do this, we are going to define two express middlewear functions in app.js.

global.user = false;

app.use("\*", async (req, res, next) => {

if (req.session.userID && !global.user) {

const user = await User.findById(req.session.userID);

global.user = user;

}

next();

})

1  
2  
3  
4  
5  
6  
7  
8  
9

app.js - a custom express middleware function

Above, the our custom middleware will fire on every request, hence the wild card symbol "\*". Notice, how the session is injected into our request for us, and we can access it like this: req.session.userID. Next we use a little trick, we set what's know as a global variable global.user = false;. Notice how we assign our logged in user to this variable, global.user = user;. Since user is now global we can access it everywhere, including in our EJS templates! If all has worked correctly, placing <%= user %> in one of your templates should output the current user. Given that we now have this global, user, variable available to us in our views, we can conditionally show content based on the user being logged-in. Try amending views/common/header.ejs to conditionally show some links, as follows:

...

<% if(user) { %>

...

<a class="nav-link" href="logout"> <%= user.email %> (Logout) </a> <%

}%>

<% if(!user) { %>

<a class="nav-link" href="login"> Login </a>

<a class="nav-link" href="join"> Register </a>

<% } %>

...

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11

views/common/header.ejs - conditionally rendering our menu based on the user being logged in

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#protecting-routes)**Protecting Routes**

We're very nearly there with our authentication journey. You may have noticed that, as it stands, we have not locked down any routes. For instance, I could still visit create-taster whether I am logged in or not. Let's write one final pice of middleware to help us lock down routes:

const authMiddleware = async (req, res, next) => {

const user = await User.findById(req.session.userID);

if (!user) {

return res.redirect('/');

}

next()

}

....

app.get("/create-taster", authMiddleware, (req, res) => {

res.render("create-taster", { errors: {} });

});

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13

app.js - the middleware above redirects un-authenticated users back to the / home page. Notice how we are selective where we use it, locking down our create-taster route - in this instance

Finally, let's consider how a user can log our. All we need to do is set up a route, that when visited destroys our session and sets the global user variable to false

app.get("/logout", async (req, res) => {

req.session.destroy();

global.user = false;

res.redirect('/');

})

1  
2  
3  
4  
5

app.js - a logout route. We now just need to set up a link in our header pointing to /logout and the user will be able to logout (I'll let you work out how to do this)

[#](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/#task-3)**Task 3**

Use the notes above to finish off our authentication system. On successful completion:

* The header links should adapt based on a users being logged in
* Your 'create' routes should all be locked down, unless a user is logged in
* A user should be able to logout

← [Lecture (Start Here)](https://joeappleton18.github.io/advanced-database-systems-2022/sessions/week_8/lecture.html)