A Guide to Authentication in React

How I learned authentication in React

Ana
Jun 15, 2020 · 7 min read ★

Introduction

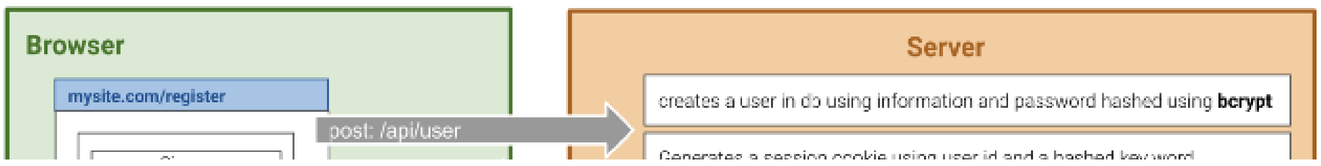
I began this post with the intent to write a definitive guide on using Passport-Local in React because the resources out there were sparse on that specific topic. React is a library that builds user interfaces while authentication happens between the browser and the server (and a database in our case). **So, lack of resources is due to the topics not being directly related.**

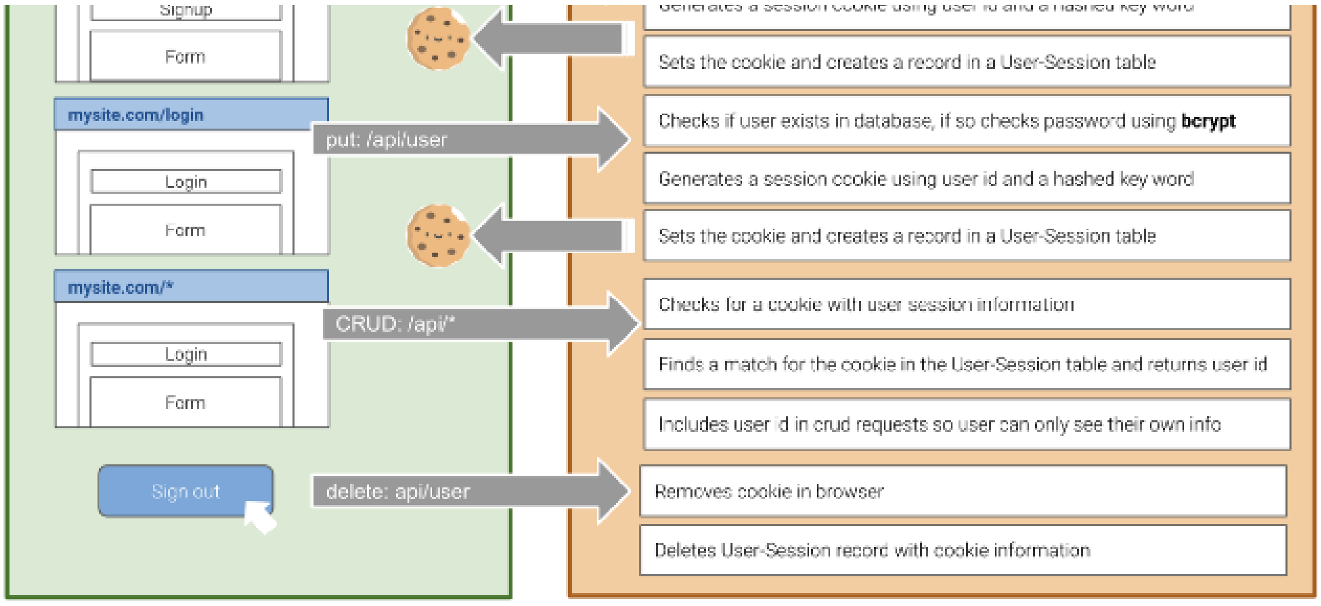
To understand the topic more deeply, I built my own Authentication method using cookies and bcrypt. I was confused on how the different elements of a MERN (MySQL, Express, React, NodeJS) application interacted with each other in the context of authentication. I addition to covering registration, login, hashing passwords, and using cookies, I review:

- How to make sure that CRUD takes into consideration the identity of the authenticated user so that they can only see, create, delete, update their own information.
- How to make sure that only authenticated users have access to certain pages.

Welcome to A Guide on Authentication!

Below is a summary of my authentication approach:


Browser
mysite.com/register
post: /api/user
Server
creates a user in db using information and password hashed using **bcrypt**
Generates a session cookie using user id and a hashed key word


Signup
Form
Sets the cookie and creates a record in a User-Session table
mysite.com/login
put: /api/user
Checks if user exists in database, if so checks password using **bcrypt**
Login
Generates a session cookie using user id and a hashed key word
Form
Sets the cookie and creates a record in a User-Session table
mysite.com/*
CRUD: /api/*
Checks for a cookie with user session information
Login
Finds a match for the cookie in the User-Session table and returns user id
Form
Includes user id in crud requests so user can only see their own info
Sign out
delete: api/user
Removes cookie in browser
Deletes User-Session record with cookie information

Summary of my authentication approach

In React, information can be shared across components using Global State and Context. Either can be a valuable tool for determining what the user sees; however, we will depend on good ole fashioned cookies for authenticating users.

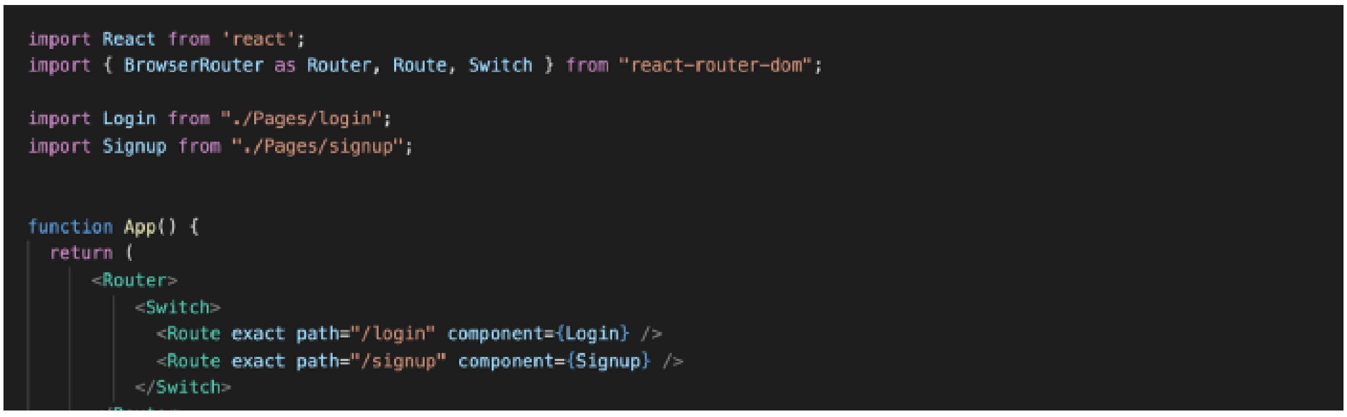
Step by Step on Authenticating Users in MERN app

Before we dive in, let's clarify some definitions! Registration is essentially a request to create a new user record. Login is a request to be given access. Access is verified by checking if the user exists in the database and if the password provided matches the password in the record.

Think of registration as being added to the list to an exclusive club and login as showing up and having to show your id to prove that you are you.

First, we need to build a few paths

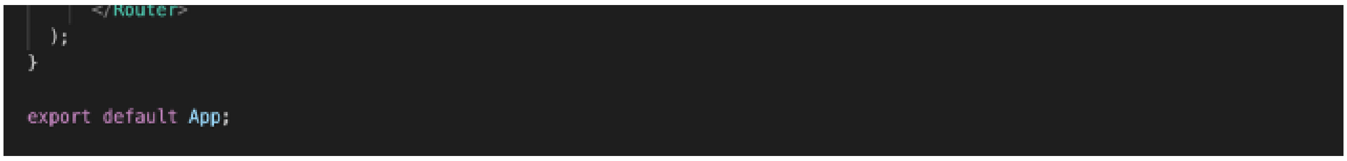
Start by creating the routes that the user will interact with directly — I chose to create two pages, one for login and one for registration or signup. In the App.js folder, I created the routes and referenced the pages I wanted to render.



```
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

import Login from "./Pages/login";
import Signup from "./Pages/signup";


function App() {
  return (
      <Router>
          <Switch>
            <Route exact path="/login" component={Login} />
            <Route exact path="/signup" component={Signup} />
          </Switch>
```

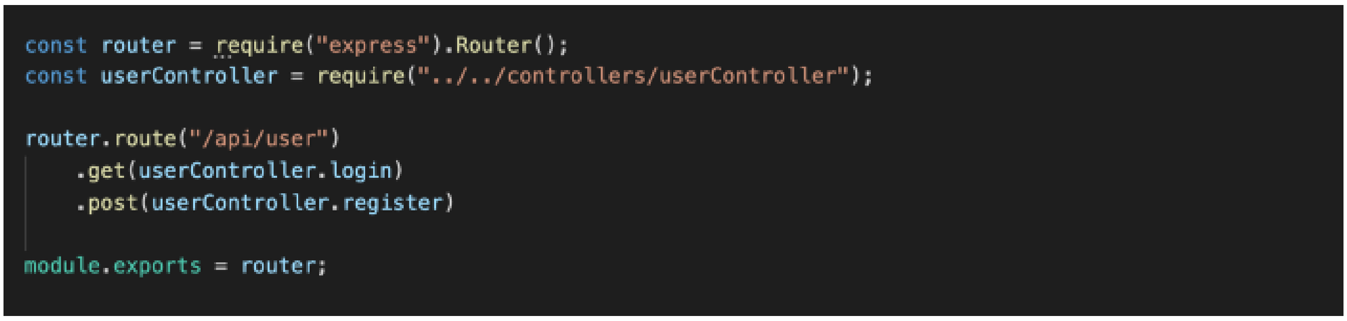


```
    </Router>
  );
}

export default App;
```

Creating login and register routes in React

On the server side, I created one route **/api/user** to handle registration and login.



```
const router = require("express").Router();
const userController = require("../../controllers/userController");

router.route("/api/user")
    .get(userController.login)
    .post(userController.register)

module.exports = router;
```

API calls on Server side to actually login and register user

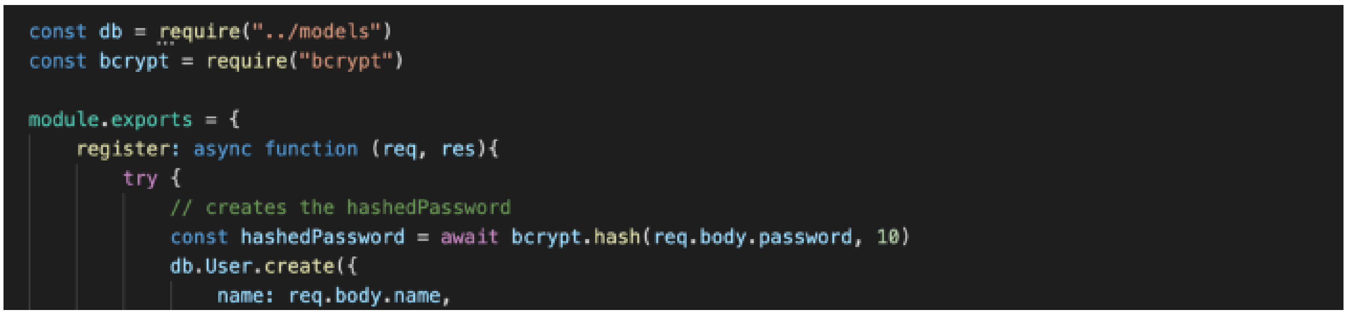
The two pieces above can all be created on a single server file or split out into multiple files as I have done so. The concepts are still the same.

Registering A New User

When a user registers, they fill out a form with information such as name, username/email and password. For our application, we will use email as the primary identifier for our users.

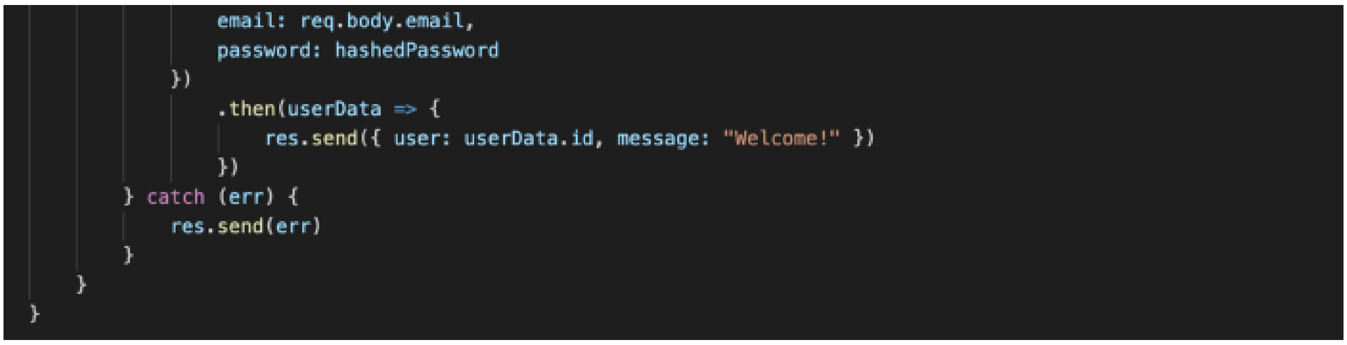
…but should the password be stored in some special way?

I'm glad you thought of that! Yes. One way to do that is to use **bcrypt**, a library that hashes passwords. In plain speak, bcrypt maps the password into something unrecognizable. In fact, I created over 20 users with the password "password" and none of the outputted text was similar in any recognizable way. So, server side, before actually creating the User record in your database, hash the password. That way, if anyone hacks your database, they won't be able to get any actual password information.



```
const db = require("../models")
const bcrypt = require("bcrypt")

module.exports = {
    register: async function (req, res){
        try {
            // creates the hashedPassword
            const hashedPassword = await bcrypt.hash(req.body.password, 10)
            db.User.create({
                name: req.body.name,
```



```
                email: req.body.email,
                password: hashedPassword
            })
                .then(userData => {
                    res.send({ user: userData.id, message: "Welcome!" })
                })
        } catch (err) {
            res.send(err)
        }
    }
}
```

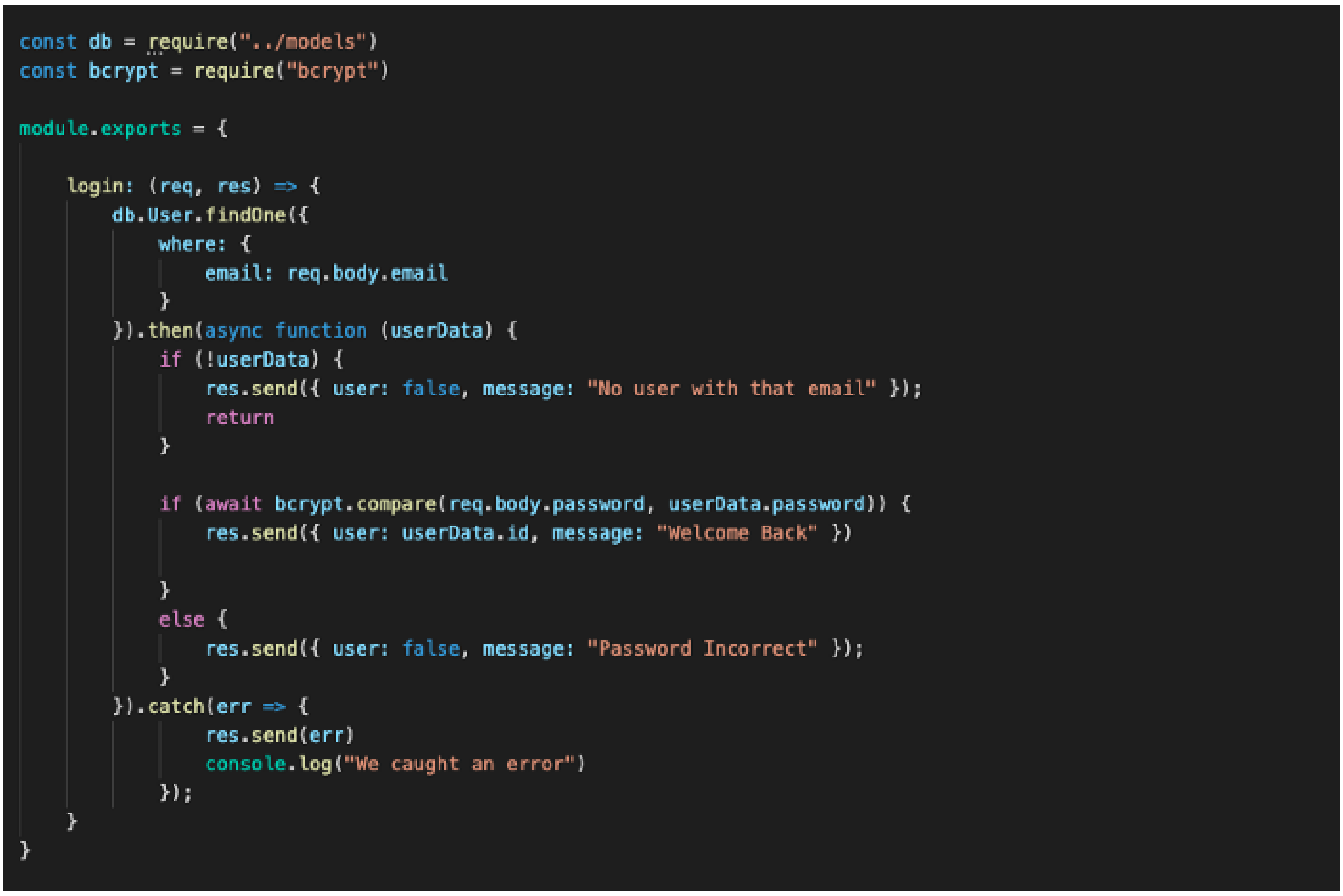
User Registration shown at the database level

An async function is used with bcrypt to ensure that the password is hashed before the User record is created. After creation, the User Id will be tied to a session record. Since the method is similar for registering and logging in, I'll explain later.

Logging in An Existing User

When a user signs in, they fill out a form with their username/email and password. The values are bundled in the API request.

Server side, we are first checking whether the user exists and then checking if the password is valid. Note that the comparison isn't a triple equal — The truth is that *we* can't match them directly but the bcrypt compare function can.



```
const db = require("../models")
const bcrypt = require("bcrypt")

module.exports = {

    login: (req, res) => {
        db.User.findOne({
            where: {
                email: req.body.email
            }
        }).then(async function (userData) {
            if (!userData) {
                res.send({ user: false, message: "No user with that email" });
                return
            }

            if (await bcrypt.compare(req.body.password, userData.password)) {
                res.send({ user: userData.id, message: "Welcome Back" })

            }
            else {
                res.send({ user: false, message: "Password Incorrect" });
            }
        }).catch(err => {
                res.send(err)
                console.log("We caught an error")
            });
    }
}
```

User login shows at the database level

Great, we've technically authenticated a user! Now, we need to make it *mean* something:

Using Cookies to Restrict Access

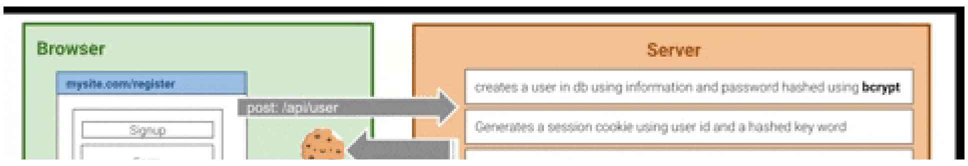
Restricting Access based on authenticated User information

Let's visit the database to understand how information interacts: User data (name, email, hashed password) is stored in a User Table. A second table stores User-Sessions by tracking Users and Active Session-Cookies relations. Every time we give a User a cookie, we store the cookie value and the User's id in this User-Session table. Users have a one-to-many relationship with User-Sessions meaning that a single User could technically be authenticated on multiple browsers at the same time — think ipad, iphone and laptop.

How We Will Use Cookies

Now it's cookie time! Cookies are a piece of data stored by a browser and sent along with every request.

When the user has been authenticated, a cookie will be created. Since cookies are included in every request, the server will be able to use Session Cookies to confirm the User's identity. With every CRUD request, we will first check the Session Cookies, and use the User-Session table to match the cookie back to a User and returning the User's identity if the relationship exists. Once the User signs out, the cookie is deleted and the User-Session record is deleted.


Browser
mysite.com/register
Signup
post: /api/user
Server
creates a user in db using information and password hashed using bcrypt
Generates a session cookie using user id and a hashed key word

Form
Sets the cookie and creates a record in a User-Session table

Illustration of how Cookies are set

Piraveena beautifully outlined relationship the between browser and server in this article. Check it out for your reference.

Creating a Cookie is the Easiest Part

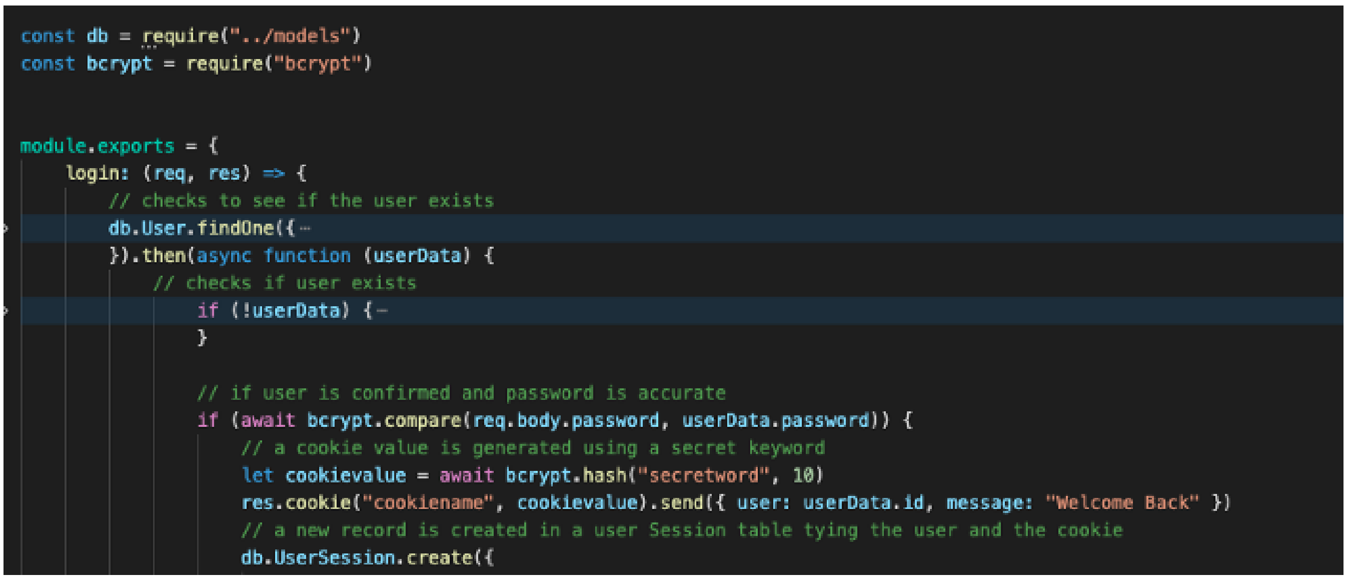
It's simple to create a cookie but it's difficult to make it unique enough that it cannot be hacked. Welcome back bcrypt!

```
let cookievalue = bcrypt.hash("secretword",10)
```

To set the cookie, we just need to tie it to a response

```
res.cookie("cookiename", cookievalue)
```

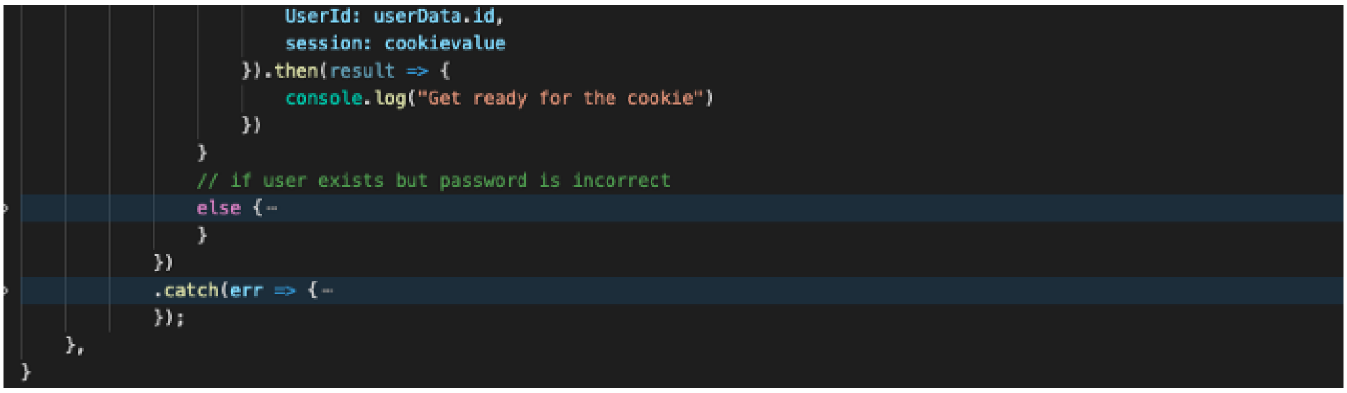
In your app, you can check the application tab under Inspect to see if this worked correctly. Here is how you can tie session cookie creation to the login process:



```
const db = require("../models")
const bcrypt = require("bcrypt")


module.exports = {
    login: (req, res) => {
        // checks to see if the user exists
        db.User.findOne({…
        }).then(async function (userData) {
            // checks if user exists
            if (!userData) {…
            }

            // if user is confirmed and password is accurate
            if (await bcrypt.compare(req.body.password, userData.password)) {
                // a cookie value is generated using a secret keyword
                let cookievalue = await bcrypt.hash("secretword", 10)
                res.cookie("cookiename", cookievalue).send({ user: userData.id, message: "Welcome Back" })
                // a new record is created in a user Session table tying the user and the cookie
                db.UserSession.create({
```

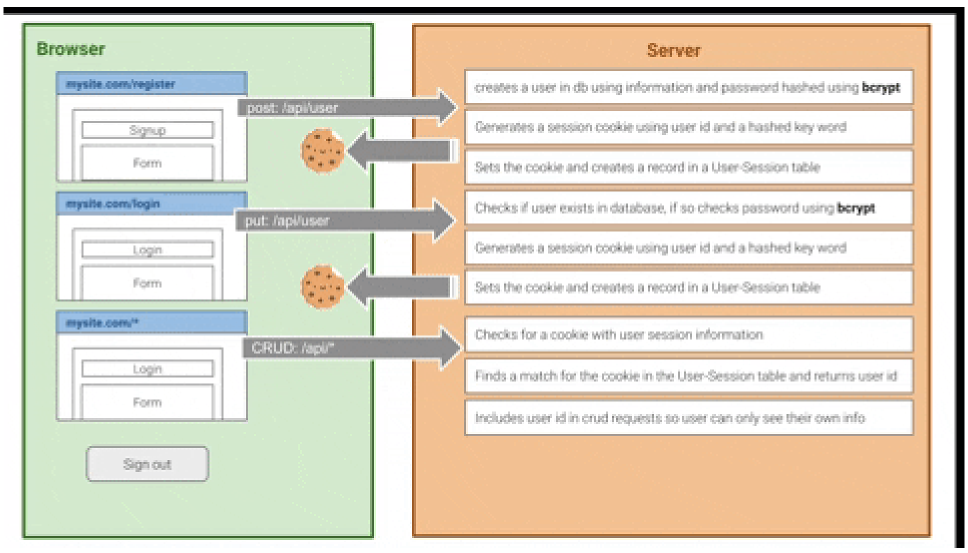


```
                        UserId: userData.id,
                        session: cookievalue
                    }).then(result => {
                        console.log("Get ready for the cookie")
                    })
                }
                // if user exists but password is incorrect
                else {…
                }
            })
            .catch(err => {…
            });
        },
}
```

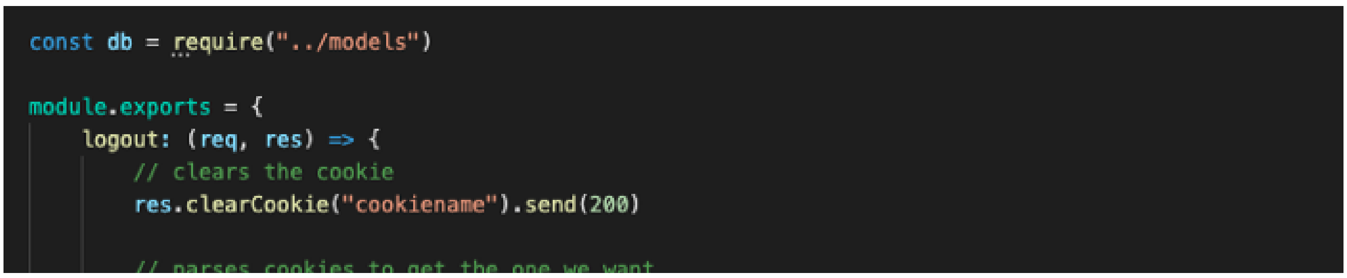
You can take a similar approach for registration but rather than waiting to confirm that the password exists, you need to wait for the user to be created in the database. Grab the User Id from the response to create a User-Session record. Client side won't receive a response until the User Session has been created.

Deleting Cookies is just as simple

When a user logs out, we remove the cookies in the browser and delete the User-Session relationship from our database.


Browser
mysite.com/register
Signup
Form
post: /api/user
mysite.com/login
Login
Form
put: /api/user
mysite.com/*
Login
Form
CRUD: /api/*
Sign out
Server
creates a user in db using information and password hashed using bcrypt
Generates a session cookie using user id and a hashed key word
Sets the cookie and creates a record in a User-Session table
Checks if user exists in database, if so checks password using bcrypt
Generates a session cookie using user id and a hashed key word
Sets the cookie and creates a record in a User-Session table
Checks for a cookie with user session information
Finds a match for the cookie in the User-Session table and returns user id
Includes user id in crud requests so user can only see their own info

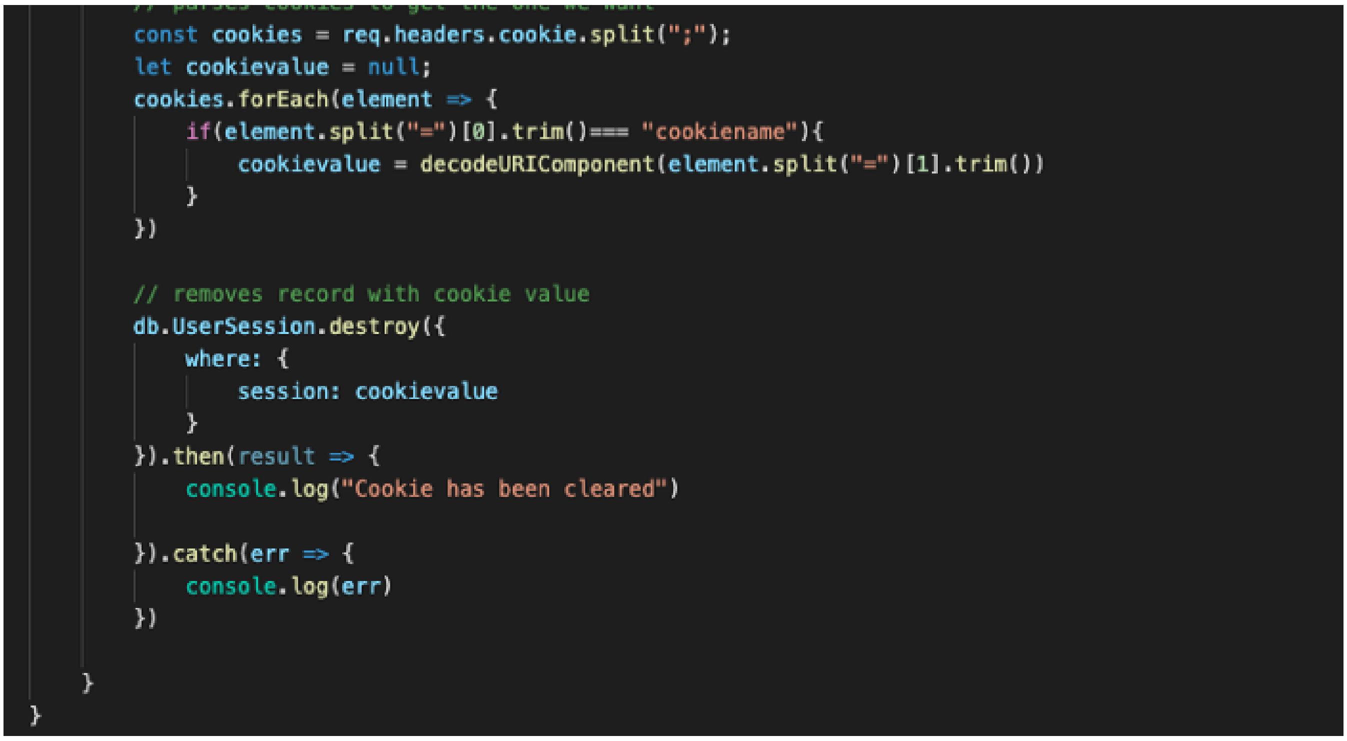
Cookies are sent with every request. To access the cookie that you specifically set you can use a parser. I chose to parse the cookie myself to better understand how a parse may work. Note the user of decodeURIComponent to undo the URL encoding that happened when the cookie was set.



```
const db = require("../models")

module.exports = {
    logout: (req, res) => {
        // clears the cookie
        res.clearCookie("cookiename").send(200)

        // parses cookies to get the one we want
```



```
        // parses cookies to get the one we want
        const cookies = req.headers.cookie.split(";");
        let cookievalue = null;
        cookies.forEach(element => {
            if(element.split("=")[0].trim()=== "cookiename"){
                cookievalue = decodeURIComponent(element.split("=")[1].trim())
            }
        })

        // removes record with cookie value
        db.UserSession.destroy({
            where: {
                session: cookievalue
            }
        }).then(result => {
            console.log("Cookie has been cleared")

        }).catch(err => {
            console.log(err)
        })

    }
}
```

How to delete a user and parse out a cookie

CRUD is the tricky part

The most important thing we've learned about cookies is that they are appended to all requests. That means that on the client side, there aren't additional pieces of information we need to append or provide in order to make API request.

The flow is as follows:

- parse the cookies to get your specific cookie
- use the value of the cookie to make a request to the User-Session table
- if the result is null, or no relationship exists, return nothing
- if there is a match, return the User Id and then use that id to fulfill the request that the user originally made

Every time the user makes a request, we are checking their credentials but doing it in a way so subtle that the client doesn't know.

This method is effective because if the user logs out their authentication — the record on the user session table — will be wiped and they will be asked to log in again.

User Session Needs to Exist For User To Have Access

What About Context?

Since we are building a MERN app, Context is a tool available to share information across React components. Even if you chose to use User Context for tracking whether there is an authenticated user logged in, you still need to validate the user with cookies. Context will still be valuable to restrict what pages are viewable to the user at a given time. For example, set context that checks for cookies every time the page loads and if there is a cookie, you could show the sign out button because it means a user is logged in.

Since Context is on the Client side, we don't want to depend on it as a method for confirming the user's identity.

What's Next?

Now that you have learned how authentication works, you can explore additional resources that teach you how to use Passport, indexedDB, cookie-parse, firebase-authentication, etc. The world is your oyster.

Thanks to Jonathon Mah.

Authentication Passport Mern Reactjs Beginners Guide