These is my notes and instructions forsection 21 using the newer Next-auth V4 wih the latest Next and Reacts.

- Thanks to Nicola, Andrew, and the other Tom for all of yur notes that helped me to organize this as shown here
- For anyone who has been struggling with this course section you can try these instructions with are numbered to match the new course video lectures.
- Below shows the package.json with the versions of the dependencies and the packages: { "name": "nextjs-course", "version": "0.1.0", "private": true, "scripts": { "dev": "next dev", "build": "next build", "start": "next start" }, "dependencies": { "bcryptjs": "^2.4.3", "mongodb": "^6.10.0", "next": "^13.5.3", "next-auth": "3.29.10", "react": "^18.2.0", "react-dom": "^18.2.0" } }

Here are the instructions organized by newer course section ID's

- Note i have corrupted my credentials forthe MOngoDB so do not even bother to try to use mine

(394 -A) Intro and Overview (-A denotes udated instructions to use newer versions)

- In this section we will add some user authentication to our project
- As part of this we will be implementing Signup and Login
- Then we can cotrol access to routes for authenticated users

(395-A) Our Starting Point - uding the newer versions of Next and React

1. Download the starting package and uszip it into a project folder named Ch-21-Authentication-Project
    - We will modify the package.json as shown below before running npm install: { "name": "nextjs-course", "version": "0.1.0", "private": true, "scripts": { "dev": "next dev", "build": "next build", "start": "next start" }, "dependencies": { "next": "^15.0.3", "react": "^18.3.1", "react-dom": "^18.3.1" } }
2. run npm install added 31 packages, and audited 32 packages in 26s

   6 packages are looking for funding run `npm fund` for details

   found 0 vulnerabilities

- Note the improvement in the lack of vulnerabilities, compared with the Legacy version we built in the other project folder

3. Test the project to make sure it runs: npm run dev

- we should see the home page and the Nav menu with a Login Profile and Logout Button, which does not do anyting yet
- the Login and Profile do open two other components that also are not wired up to do anything yet

- Our goals besides implementing authentication are to protectt he route for the Profile for only authenticated users and the Login route should not be accessable if we alerady are authenticated.
- Also we may decide to hide the Logout button unless we are authenticated as well.
- this should apper basically the same as the other version in step (395)

(396) Understanding how Authentication Works This is unchanged as it is just the basic explanation notes

```
- Generally we will have a cient and a  server, the client being the react
component that runs in the browser and the server being the nextJS node.js
server-side which can connect to the database where our users and
passwords will be stored.
- we want to ensure that the passwords are encrypted
- Assuming the account exists the user will open a login form and enter
their email or username and a password and click the Login button
- that acrion wll then send a rquest (API request) to the backend server
to be processed
- the server will handle the validation by comparing the credentials with
those from the database
    - this will include decrypting the encrypted passwrod on from the
database with the unencrypted password the user just entered.
    - this is done on the server side and never on the client side
    - a response will then be sent back to the client either confirming or
denying the login based on the credentials that were provided
    - the response shoud include a token of some sort to be saved in the
browser to be used for the duration of the session
    - following the authentication the client should be able to reach
protected routes and make aditional request to the server
    - additional requests could, for example, be to change the password
and those requests must be accompanied by a security token which can be
sent in with the request headers
- There are two main mechanisms to prove validation for a request:
            - server-side sessions
                - a unique session ID is stored on the server, possibly in
a database
                - then we send that same identifier to the  client who
will send it back along with every subsequent request tot he server to
maintain authentication
                - the server can extract the identifier from the incoming
request for validation
                - the connetion should be using ssl for encryption so the
identifier cannot be stolen
                - the identifier typically is stored on the client side in
a cookie which should be properly configured via JavaScript to prevent
cross-site scripting attacks

            - authentication tokens
                - in this case the server does not store any identifiers
```

```
                - instead the server creates and signs tokens amd send
them to the client
                - the client then sends the signed token back  along with
its outgoing requests to the server
                - the server know how it signed the token so it cannot be
spoofed or imitated. therfore the server can validate the token sent back
to it with every request

    - Typically with NextJS and React SPA's we will be usng the
Authentication Tokens
        - pages are served directly and populated with logic without
having to be constanty re-loaded from the server
        - Backend API's are typically stateless and will not support
sessions
        - each API request is independant of all other API requests
        - The API does not store any information about connected clients
        - The server may not be involved in every action and every request
in a React SPA application, some may be handled with front-end JavaScript
        - API's are always basically a stateless connection with our SPA
        - Because of the above we have a detached front-end / back-end
appliction
        - Therefore the use of the authentication token is the only proper
method to use for authentication in these cases
```

JSON Web Tokens JWT

```
- The token concept we use is known a JSON Web Tokens (JWT)
- we will use a third party library for generating and signing the tokens
These are created using 3 building blocks:
    - issuer data - automatically added into the token when it is
generated
    - custom data (user data)
    - secret signing key setup on the server-side. the client never gets
to see the key
- The three above pieces are combined together bythe thrid party pakage t
create the JWT
- Note that signing does not mean encryption   anyone could read the token
- The token is stored by the browser nd it will be attached to each
request
- Only the signing server can validate the token
```

(397) . Must Read Install older next-auth version

```
    This course was recorded with next-auth v3. Currently, the latest
version is v4. It generally works the same as shown in the lectures but
some imports and method names changed.
```

```
    Therefore, I recommend that you follow along with v3 for a smooth
experience (this command needs to be executed as part of the next
lecture):

    npm install --save-exact next-auth@3
    You can, optionally, also dive into the official v4 migration guide to
update your code to next-auth v4, if you want to: https://next-
auth.js.org/getting-started/upgrade-v4
```

(397-A)

```
     With React 18+, Next-Auth follow the -A steps (above and below) to
use the newer versions of dependencies and next-auth@4
     any stepswithout the -A would be unchanged and the same for both
versions of next-auth
```

(398-A) Using the Next Auth Library

1. In the newer project folder Ch-21-Authentication-Project stop the dev server if it is running
2. Install the newer version of next-auth to the newer project folder: npm install --save next-auth added 16 packages, and audited 48 packages in 9s

   10 packages are looking for funding run `npm fund` for details

   found 0 vulnerabilities
3. Check the package.json to verify what we have installed: { "name": "nextjs-course", "version": "0.1.0", "private": true, "scripts": { "dev": "next dev", "build": "next build", "start": "next start" }, "dependencies": { "next": "^15.0.3", "next-auth": "^4.24.10", "react": "^18.3.1", "react-dom": "^18.3.1" } }
4. Restart the dev server and verify that the project still runs without any errors in either client-side or server-side consoles

- Note that we have not added any new code yet but now we have the newer dependencies for our project in the newer project folder
- we will follow the video for each step then also provide the -A companion steps using the newr dependencies and the version 4 of next-auth
- Also note that with the newer dependencies and packages we do ot have any error or warning about vulnerabilities as well

(399-A) Adding a User Signup API route to our Project (newer dependencies and packages)

1. stop the dev server, install the mongodb package with the command: npm install mongodb
2. also we want to install an encryption package with the command npm install bcryptjs
3. then restart the dev srver.
4. create an api folder in the pages folder
5. We will be needing to interface with the MongoDB database from more than one portion of our project so to be forward thinkng in our code refactoring we will also create a lib folder in the project root.
6. In that lib folder create a db.js as shown below:

```
import {MongoClient} from 'mongodb';

const CONNECT_URL =
`mongodb+srv://bananna:coconut@cluster0.wbbvg.mongodb.net/your_own_db_name
?retryWrites=true&w=majority`;

export async function connectToDatabase() {

const client =  await   MongoClient.connect(CONNECT_URL)

return client;

}
```

7. In the lib folder create a new code file named auth.js as shown below:

```
    import { hash } from  'bcryptjs';


        export async function hashPassword(password) {

        const hashedPassword =   await hash(password, 12);
        return hashedPassword;
        }
```

8. In the api folder create an auth subfolder
9. in the auth subfolder create a new signup.js code file as shown below:

```
    import { connectToDatabase } from '../../../lib/db';
    import { hashPassword } from '../../../lib/auth';

    async function handler(req, res) {
         if (req.method === 'POST'){
            const data = req.body;

            const { email, password } = data

            if (!email || !email.includes('@') || !password ||
```

```
password.trim().length < 7 ) {
                res.status(422).json({message: 'invalid credentials!'});
                return;
            }

        const client =  await connectToDatabase();

        const hashedPassword = await hashPassword(password);

        const db = client.db();

        const result = await  db.collection('users').insertOne({
            email: email,
            password: hashedPassword
        })
            res.status(201).json({message: 'Created user!'});
        }
    }
    // we should implement error handling later in production version
    export default handler;
```

- In the next section we will be sure the submission form for creating a new user is 'wired up' to the api route properly

(400-A) Sending Signup Request From the Frontend (new dependency versions and packages)

- In this section we will 'wire up' the components/auth/auth-form.js to the API route

1. Edit the components/auth/auth-form.js AuthForm component as shown below:
    - update this code:

```
    import { useState } from 'react';
import classes from './auth-form.module.css';

function AuthForm() {
  const [isLogin, setIsLogin] = useState(true);

  function switchAuthModeHandler() {
    setIsLogin((prevState) => !prevState);
  }

  return (
    <section className={classes.auth}>
      <h1>{isLogin ? 'Login' : 'Sign Up'}</h1>
      <form>
        <div className={classes.control}>
          <label htmlFor='email'>Your Email</label>
          <input type='email' id='email' required />
        </div>
```

```
          <div className={classes.control}>
            <label htmlFor='password'>Your Password</label>
            <input type='password' id='password' required />
          </div>
          <div className={classes.actions}>
            <button>{isLogin ? 'Login' : 'Create Account'}</button>
            <button
              type='button'
              className={classes.toggle}
              onClick={switchAuthModeHandler}
            >
              {isLogin ? 'Create new account' : 'Login with existing
account'}
            </button>
          </div>
        </form>
      </section>
    );
}

export default AuthForm;
```

- to this:

```
        import { useState, useRef } from 'react';
    import classes from './auth-form.module.css';

    async function createUser(email, password) {
    const response = await  fetch('/api/auth/signup', {
        method: 'POST',
        body: JSON.stringify({email, password }),
        headers: {
            'Content-Type': 'application/json'
        }

    });
    const data = await response.json();
    if (!response.ok){
        throw new Error(data.message || 'Something went wrong!')
    }
    return data;
    }

    function AuthForm() {
        const emailInputRef = useRef();
        const passwordInputRef = useRef();
        const [isLogin, setIsLogin] = useState(true);
```

```
    function switchAuthModeHandler() {
        setIsLogin((prevState) => !prevState);
    }

    async function submitHandler(event) {
        event.preventDefault();
        const enteredEmail = emailInputRef.current.value;
        const enteredPassword = passwordInputRef.current.value;

        // add optional validation later

        if (isLogin) {
            //add login logic later
        } else {
            try {
                const result =  await  createUser(enteredEmail,
enteredPassword);
                console.log(result);
                // optional add user result feedback later
            } catch (error) {
                console.log(error);
                // optional add user error feedback later
            }

        }
    }

    return (
        <section className={classes.auth}>
        <h1>{isLogin ? 'Login' : 'Sign Up'}</h1>
        <form onSubmit={submitHandler}>
            <div className={classes.control}>
            <label htmlFor='email'>Your Email</label>
            <input type='email' id='email' required  ref={emailInputRef}/>
            </div>
            <div className={classes.control}>
            <label htmlFor='password'>Your Password</label>
            <input type='password' id='password' required ref=
{passwordInputRef}/>
            </div>
            <div className={classes.actions}>
            <button>{isLogin ? 'Login' : 'Create Account'}</button>
            <button
                type='button'
                className={classes.toggle}
                onClick={switchAuthModeHandler}
            >
                {isLogin ? 'Create new account' : 'Login with existing
account'}
            </button>
            </div>
        </form>
```

```
            </section>
        );
        }

        export default AuthForm;
```

ing specific to next-auth versions or use in this section - Note there is noth

2. Save all relaod and test
    - from the Login component select Create new account and enter a valid email and a password
    - the form will not clear or provide any UI indication at this point
3. Check the MongoDb Atlas and verify that we have a new user added to the DB and collection we specified: {"_id": {"$oid":"673f4e88e3a434d35d53b2cc"},"email":"test@vtestme.org","password":"$2acbhxsoYb/y7YzlguS.Gxh2Herujwb7vv3obJoEW0zg7HagelQq4Qwk8UwHW"}

(401-A) Improving Signup with Unique Email Addresses (newer dependencies and packages)

- Our new code for creatong a user works however we have no ristrictions in place to avoid creating users with the same email address. We rally want each user to be uniques so wew ill need to implement some code logic to prevent thec reation of a user with an email that already exist in our database.
- therefore we will add some code logic to check if given user email already exist in the database and if so not allow ths to be duplicated.

1. To illustrate we will edit the api/auth/signup.js as shown below:

```
    import { connectToDatabase } from '../../../lib/db';
import { hashPassword } from '../../../lib/auth';

async function handler(req, res) {

    if (req.method === 'POST'){

        const data = req.body;

        const { email, password } = data

        if (!email || !email.includes('@') || !password ||
password.trim().length < 7 ) {
            res.status(422).json({message: 'invalid credentials!'});
            return;
        }

    const client =  await connectToDatabase();
```

```
    const db = client.db();

    const existingUser = await db.collection('users').findOne({email:
email});
    if(existingUser){
        res.status(422).json({message: 'User with this email exists'})
        client.close();
        return;
    }


    const hashedPassword = await hashPassword(password);

    const result = await  db.collection('users').insertOne({
        email: email,
        password: hashedPassword
    })
        res.status(201).json({message: 'Created user!'});
        client.close();
  }
}
// we should implement error handling later in production version
export default handler;
```

2. Save all reload and test
    - If we attempt to enter a user with the same email in the dev-tools console we should see: Error: User with this email exists along with the 422 error code
    - we have not yet done anything to have this display for the user in the UI

(402-A) Adding the "Credentials Auth Provider" & User Login Logic (newer versions and packages)

- Now that we have managed to be able to properly create users we will begin to work on the authentication
- Here is where we will be using the next-auth package
- In this section we will be implementing the V4 with the Newer versions of Next and React
- We will be usng next-auth for two basic tasks: - to authenticate users - to check if a user is authenticated

1. we will be needing to validaate the saved hashed password in our database so first we will add that functionality to our lib/auth.js code file as shown below:

```
        import { hash, compare } from  'bcryptjs';

        export async function hashPassword(password) {

        const hashedPassword =   await hash(password, 12);
        return hashedPassword;
```

```
}

export async function verifyPassword(password, hashedPassword){
    const isValid = await compare(password, hashedPassword);
    return isValid;
}
```

- Notes: - we added the compare function to the imports: import { hash, compare } from 'bcryptjs';

```
- we added the new exported async function to compare the password:
```

export async function verifyPassword(password, hashedPassword){ const isValid = await compare(password, hashedPassword); return isValid; }

2. In the api/auth folder we will create a special dynamic catch-all api route file name of [...nextauth].js as shown below which is sing the newer versions of next-auth so this is where the instructions vs the instrctions-A will begin th be different:

```
import NextAuth from "next-auth";
import CredentialsProvider from "next-auth/providers/credentials";
import { connectToDatabase } from "../../../lib/db";
import { verifyPassword } from "../../../lib/auth";

export const authOptions = {
    session: {
        jwt: true
    },
providers: [
    CredentialsProvider({
    async authorize(credentials) {
        const client = await connectToDatabase();

                    const usersCollection =
client.db().collection('users');
                    //previously we set the credentials properties in
a configuration
                    const user = await usersCollection.findOne({email:
credentials.email})
                    if(!user){
                        client.close();
                        throw new Error('user not found!')
                    }
                    // if we get here it means we found the email
adderess match and now we need to validate the password
                    // keep in mind that the stored password has been
hashed

                    const isValid = await
verifyPassword(credentials.password, user.password);
```

```
                        if (!isValid) {
                            client.close();
                            throw new Error('Login Failed!')
                        }
                            // if we get this far we know we have an
existing  email and a validated password
                            // therefore we consider the user to be logged
in so we will return an object:

                            client.close();
                            return {email: user.email};
            },
            }),
        ],
        };

        export default NextAuth(authOptions);
```

- Notes:
    - the big difference here seems to be the import and how it is called
    - V4 is mported: import CredentialsProvider from "next-auth/providers/credentials";
        - then used in the code: providers: [CredentialsProvider({ async authorize(credentials) {...} ...}) ...]
    - V3 is imported: import Providers from 'next-auth/providers';
        - then used in the code: providers: [Providers.Credentials({ async authorize(credentials) {...} ...}) ...]
    - otherwise the rest of the logic is basically the same except the export syntax differs
        - in V3 we export NextAuth on the first line right after the package imports : export default NextAuth({...})
        - in V4 we use different syntax we firs export everything as a const: export const authOptions = {...}
            - then at the end we default export NextAuth with the const as an argument: export default NextAuth(authOptions);

(403-A) Sending a Signin Request from the FrontEnd (new packages and next-auth v4)

- Previously we implemented the catch-all api route to use for authentication of users.
- In this section we used the newer version of next-auth V4
- Part of this process included creating the JWT token for the user to validate their authentication with repeated requests to the site
- we will also be using the authentiction state to enable or disble certain routes in some subsequent coure sectons and also t control some logic in the api routes as well to limit or allw what a user can

do.

- This previous implementation next-auth package has been put to use for us to help us in authenticating users as well as determining of a current user is authenticated or not.
- This was done largely through managing the creation and storage of the JWT token.
- Now we will be 'wiring up' this api route to our signin form in this section.
- At the moment we only have the api route logic and we need to add code logic to our fron-end components/auth/auth-form.js
- In that form previously we left a reminder for ourselves to return here and use this later:

```
...
// add optional validation later

if (isLogin) {
    //add login logic later
} else {...}
```

1. We will now edit this auth-form.js code file as shown below

```
import { useState, useRef } from 'react';
import classes from './auth-form.module.css';
import { signIn } from "next-auth/react";

async function createUser(email, password) {
const response = await  fetch('/api/auth/signup', {
    method: 'POST',
    body: JSON.stringify({email, password }),
    headers: {
        'Content-Type': 'application/json'
    }

});
const data = await response.json();
if (!response.ok){
    throw new Error(data.message || 'Something went wrong!')
}
return data;
}

function AuthForm() {
    const emailInputRef = useRef();
    const passwordInputRef = useRef();
    const [isLogin, setIsLogin] = useState(true);


function switchAuthModeHandler() {
    setIsLogin((prevState) => !prevState);
}

async function submitHandler(event) {
```

```
            event.preventDefault();
            const enteredEmail = emailInputRef.current.value;
            const enteredPassword = passwordInputRef.current.value;

            // add optional validation later

            if (isLogin) {
                //add login logic later:
                const result = await signIn('credentials', {
                redirect: false,
                email: enteredEmail,
                password: enteredPassword,
                });
                console.log(result);
            } else {
                try {
                    const result =  await  createUser(enteredEmail,
enteredPassword);
                    console.log(result);
                    // optional add user result feedback later
                } catch (error) {
                    console.log(error);
                    // optional add user error feedback later
                }

            }
        }

        return (
            <section className={classes.auth}>
            <h1>{isLogin ? 'Login' : 'Sign Up'}</h1>
            <form onSubmit={submitHandler}>
                <div className={classes.control}>
                <label htmlFor='email'>Your Email</label>
                <input type='email' id='email' required  ref=
{emailInputRef}/>
                </div>
                <div className={classes.control}>
                <label htmlFor='password'>Your Password</label>
                <input type='password' id='password' required ref=
{passwordInputRef}/>
                </div>
                <div className={classes.actions}>
                <button>{isLogin ? 'Login' : 'Create Account'}</button>
                <button
                    type='button'
                    className={classes.toggle}
                    onClick={switchAuthModeHandler}
                >
                    {isLogin ? 'Create new account' : 'Login with existing
account'}
                </button>
                </div>
            </form>
```

```
            </section>
        );
        }

        export default AuthForm;
```

- for now, above, to test we log the result to see if our next-auth is working

2. Save all relod and test - try a login - check the dev-tools console log if we have entered an invalid password with a valid user we should see:

```
{
    "error": "Login Failed!",
    "status": 200,
    "ok": true,
    "url": null
}
```

```
        - if we enter an invalid user email (one that does not exist in the db) we should see:
```

```
{
    "error": "user not found!",
    "status": 200,
    "ok": true,
    "url": null
}
```

```
        - if we enter a valid email and password we get a response that may at first look a bit surprizing:
```

```
{
    "error": null,
    "status": 200,
    "ok": true,
    "url": "http://localhost:3000/auth"
}
```

```
            - what we get here is actually a good response because the "error: property is null, the "status":200 and "ok": true
```

```
            - The above indicates a successful login
```

3. Now that we know this is working
4. Also see the code for [...nextauth].js below with all the commented out comment lines which can be uncommented to explore how this is workng along with the signIn function provied by NextAuth. We wreally do not have access to the signIn() code because that is part of the next-auth package

```
import NextAuth from "next-auth";
import CredentialsProvider from "next-auth/providers/credentials";
import { connectToDatabase } from "../../../lib/db";
import { verifyPassword } from "../../../lib/auth";

export const authOptions = {
    session: {
        jwt: true
    },
  providers: [
    CredentialsProvider({
      async authorize(credentials) {
        //  console.log('authorize has been called');
        const client = await connectToDatabase();

                const usersCollection = client.db().collection('users');
                //previously we set the credentials properties in a
configuration
                //  console.log('made db connection')
                //  console.log('credentials.email: ', credentials.email)
                const user = await usersCollection.findOne({email:
credentials.email})
                //  console.log('found: ', user);
                if(!user){
                  // console.log('no user found')
                    client.close();
                    throw new Error('user not found!')
                }
                // if we get here it means we found the email adderess
match and now we need to validate the password
                // keep in mind that the stored password has been hashed
                //  console.log('credentials.password: ',
credentials.password)
                const isValid = await verifyPassword(credentials.password,
user.password);
                if (!isValid) {
                  // console.log('invalid password', isValid)
                    client.close();
                    throw new Error('Login Failed!')
                }
                    // if we get this far we know we have an existing
email and a validated password
                    // therefore we consider the user to be logged in so
we will return an object:
```

```
                    // console.log('password valid: ', isValid);
                    client.close();
                    return {email: user.email};
      },
    }),
  ],
};

export default NextAuth(authOptions);
```

- So lets think about doing something now more than just logging the result as shown below in the code snippet from that section:

```
  if (isLogin) {
        // login logic below:
        const result = await signIn('credentials', {
          redirect: false,
          email: enteredEmail,
          password: enteredPassword,
        });
        // console.log(result);
        if (!result.error) {
          // set some auth state by saving the token
        }
```

- we might think abut using some global state to do so but that would be lost with a restart of the page so we want to do somethinbg that would involve usng the JWT and saving that somewhere for as long as the user is authorized ad that is what we wll work on next...

(404-A) Managing Active Sessions (on the Frontend) Using the V4 of next-auth

1. In order to use useSession, next-auth v4 needs SessionProvider (Doc: "Version 4 makes using the SessionProvider mandatory") So, we first edit _app.js as shown below:

```
        import Layout from '../components/layout/layout';
        import { SessionProvider } from "next-auth/react";
        import '../styles/globals.css';

        function MyApp({ Component, pageProps }) {
        return (

        <SessionProvider session={pageProps.session}>
            <Layout>
            <Component {...pageProps} />
            </Layout>
        </SessionProvider>
```

```
        );
        }

        export default MyApp;
```

- Notes:
    - above we imported the SessionProvider:

```
            import { SessionProvider } from "next-auth/react";
```

```
- then we use it as a wrapper as shown above
```

2. Then we can edit the components/layout/main-navigation.js as shown below:

```
    import Link from 'next/link';
    import { useSession, signOut } from "next-auth/react";

    import classes from './main-navigation.module.css';

    function MainNavigation() {
    const { data: session, status } = useSession();

    function logoutHandler() {
    console.log('log me out!')
    // more code will be added later
    }

    return (
        <header className={classes.header}>
        <Link href="/">
            <div className={classes.logo}>Next Auth</div>
        </Link>
        <nav>
            <ul>
            {!session && status !== "loading" && (
                <li>
                <Link href="/auth">Login</Link>
                </li>
            )}
            {session && (
                <li>
                <Link href="/profile">Profile</Link>
                </li>
            )}
            {session && (
                <li>
                <button onClick={logoutHandler}>Logout</button>
```

```
                </li>
            )}
            </ul>
        </nav>
        </header>
    );
    }

    export default MainNavigation;
```

- Notes: - we import the useSessions an also the signput which we will use later:

```
                            import { useSession, signOut } from "next-
auth/react";
```

```
    - then we implement the useSession with someslightly different syntax
than we used with the older version of next-auth:
```

```
                            const { data: session, status } =
useSession();
```

```
    - then we use some similar logic for showing or hiding elements in the
Nav menu:
```

```
            <ul>
            {!session && status !== "loading" && (
                <li>
                <Link href="/auth">Login</Link>
                </li>
            )}
            {session && (
                <li>
                <Link href="/profile">Profile</Link>
                </li>
            )}
            {session && (
                <li>
                <button onClick={logoutHandler}>Logout</button>
                </li>
            )}
            </ul>
```

```
- we will come back later and finish the logoutHandler()
```

(405-A) Adding User Logout (with new dependencies and next-auth v4)

1. Adding logout is easy as shown below because it is also a function provided to us by next-auth so we can edit the main-navigation.js as shown below:

```
    import Link from 'next/link';
    import { useSession, signOut } from "next-auth/react";

    import classes from './main-navigation.module.css';

    function MainNavigation() {
    const { data: session, status } = useSession();

    function logoutHandler() {
        signOut();
    }
    // more code will be added later

    return (
        <header className={classes.header}>
        <Link href="/">
            <div className={classes.logo}>Next Auth</div>
        </Link>
        <nav>
            <ul>
            {!session && status !== "loading" && (
                <li>
                <Link href="/auth">Login</Link>
                </li>
            )}
            {session && (
                <li>
                <Link href="/profile">Profile</Link>
                </li>
            )}
            {session && (
                <li>
                <button onClick={logoutHandler}>Logout</button>
                </li>
            )}
            </ul>
        </nav>
        </header>
    );
    }

    export default MainNavigation;
```

- Notes: - we added signOut to our imports:

```
import { useSession, signOut } from 'next-auth/react';
```

```
- then we added an onClick event to our Logout button:
```

```
<button onClick={logoutHandler}>Logout</button>
```

- and we add the logoutHandler() function:

```
function logoutHandler() {
    signOut();
}
```

- the only real differnce in this code vs the legacy version 3 is the import line which the newer version 4 imports from: import { useSession, signOut } from 'next-auth/react
- while the older V3 uses this import line: import { useSession, signOut } from 'next-auth/client';
- Note that when we logout the session token cookie will be removed but the other two cookies remain.

2. Save relaod and test - Once we are logged in click the Logout element in the Nav menu - After clicking the Logout button note that the items displayed in the Nav menu change and we only see the Login menu item. - The signOut() function will return a promise which tells us when it has completed, but here we do nt even care about that because since we are using the useSession() hook the component will be updatd automatically, as soon as the active session changes, and it will chnage when we sign out. - NextJS will clear the cookie that is holding the JWT thereby clearing the information that the active user is logged in. - Note that when we logout the session token cookie will be removed but the other two cookies remain.

(406-A) Adding client side guard rails (route protection) Using newer depenc=dencies and V4 of next-auth

1. edit the components/profile/user-profile.js as shown below:

```
import ProfileForm from './profile-form';
import { useSession } from "next-auth/react";
import classes from './user-profile.module.css';

function UserProfile() {
// Redirect away if NOT auth
const { data: session, status } = useSession();
```

```
    if (status === "loading") {
    return <p className={classes.profile}>Loading...</p>;
    }
    if (status === "unauthenticated") {
    // For the sake of using the "reacty way" and consistency, using
router would be better.
    window.location.href = "/";
    }

    return (
        <section className={classes.profile}>
        <h1>Your User Profile</h1>
        <ProfileForm />
        </section>
    );
    }

    export default UserProfile;
```

- Notes:
- the SessionProvider we added in _app.js fixes the issue where the loading status never changes. So, you won't need Max' workaround. instead we can simply use the useSession() hook as shown below we import the V4 differently than the V3:

```
import { useSession } from "next-auth/react";
```

- we impleent useSession, not the difference in the syntax vs V3:

```
  const { data: session, status } = useSession();
```

- we add the logic to check the status of both loading and session

```
if (status === "loading") {
  return <p className={classes.profile}>Loading...</p>;
}
if (status === "unauthenticated") {
  // For the sake of using the "reacty way" and consistency, using router
would be better.
  window.location.href = "/";
}
```

- there is good reason to use newer packages as is evidenced by this difference in the coomplexity of code required for the older version vs the newer V4 along with the newer Next and React.

2. Save reload and test

- Now with ourselves being logged in with the Profile page open click the logout button
- the Nav menu should only display the Login option
- we should be redirected to the home page
- also if we try to manualy visit the profile route when logged out with the url http://loclhoat:3000/profile we shoumd be redircted away as well
- as an option we could have the redirect go directly to the auth route if desired:

```
window.location.href = "/auth";
```

2. Save relaod and test

- Now with ourselves being logged in with the Profile page open click the logout button
- the Nav menu should only display the Login option
- we should be redirected to the home page
- also if we try to manualy visit the profile route when logged out with the url http://loclhoat:3000/profile we shoumd be redircted away as well
- with this newer version when we try to visit the route anually, without logging on we may briefly see the Loading message then be redirected.

(407-A) Adding Server-Side Page Guards (route guards) And when to do so... (New versions Next and React and V4 of next-auth)

1. We will edit the pages/profile.js page as shown below:

```
        import { getSession } from 'next-auth/react';
        import UserProfile from '../components/profile/user-profile';


        function ProfilePage() {
        return <UserProfile />;
        }

        export async function getServerSideProps(context) {
        const session = await   getSession({req: context.req})

        if (!session) {
            return {
            redirect: {
                destination: '/',
                permanent: false
            }
            };
        }

        return {
```

```
            props: { session },
        }

        }

        export default ProfilePage;
```

- Note the only difference here between this and the V3 version with the legacy Next and React is the import line:

```
import { getSession } from 'next-auth/react';
```

- instead of:

```
import { getSession } from 'next-auth/client';
```

2. then we can remove/remark-out the code from the components/profile/user-profile.js as shiwn below:

```
    import ProfileForm from './profile-form';
    // import { useSession } from "next-auth/react";
    import classes from './user-profile.module.css';

    function UserProfile() {
    // Redirect away if NOT auth
    //   const { data: session, status } = useSession();
    // if (status === "loading") {
    //   return <p className={classes.profile}>Loading...</p>;
    // }
    // if (status === "unauthenticated") {
    //   // For the sake of using the "reacty way" and consistency, using
router would be better.
    //   window.location.href = "/";
    // }

    return (
        <section className={classes.profile}>
        <h1>Your User Profile</h1>
        <ProfileForm />
        </section>
    );
    }

    export default UserProfile;
```

3. Save all relaod and test

- login then open the profile page
- logout and note that we are redirected to the home oage and the only page we can get to will be the login poage
- try to manually go to the profile page, we should be redirected back to the home page, without even seeing the Loading... message or even a flash of the profile page.
- We will return to work on the error shiwn below for now we can ignore it

```
    [next-auth] [warn] [nextauth_url]
    https://next-auth.js.org/warn/warnings#nextauth_url NEXTAUTH_URL
environment variable not set
```

- I have not seen that in either of the legacy orthe newer versiond of naedt-auth in my examples

(408-A) Protecting the Auth Page (Newer dependencies and next-auth v4)

1. Generally we handle this the same way and by redirecting from the auth-form.js page as shown below:

```
import { useState, useRef } from 'react';
        import classes from './auth-form.module.css';
        import { signIn } from "next-auth/react";
        import { useRouter } from 'next/router';

        async function createUser(email, password) {
        const response = await  fetch('/api/auth/signup', {
            method: 'POST',
            body: JSON.stringify({email, password }),
            headers: {
                'Content-Type': 'application/json'
            }

        });
        const data = await response.json();
        if (!response.ok){
            throw new Error(data.message || 'Something went wrong!')
        }
        return data;
        }

        function AuthForm() {
            const emailInputRef = useRef();
            const passwordInputRef = useRef();
            const [isLogin, setIsLogin] = useState(true);
            const router = useRouter();

        function switchAuthModeHandler() {
            setIsLogin((prevState) => !prevState);
        }
```

```
        async function submitHandler(event) {
            event.preventDefault();
            const enteredEmail = emailInputRef.current.value;
            const enteredPassword = passwordInputRef.current.value;

            // add optional validation later

            if (isLogin) {
                //add login logic later:
                const result = await signIn('credentials', {
                redirect: false,
                email: enteredEmail,
                password: enteredPassword,
                });
                //console.log(result);
                if (!result.error) {
                    // set some auth state by saving the token
                    //redirect thw uer
                    router.replace('/profile');
                  }
            } else {
                try {
                    const result =  await  createUser(enteredEmail,
enteredPassword);
                    console.log(result);
                    // optional add user result feedback later
                } catch (error) {
                    console.log(error);
                    // optional add user error feedback later
                }

            }
        }

        return (
            <section className={classes.auth}>
            <h1>{isLogin ? 'Login' : 'Sign Up'}</h1>
            <form onSubmit={submitHandler}>
                <div className={classes.control}>
                <label htmlFor='email'>Your Email</label>
                <input type='email' id='email' required  ref=
{emailInputRef}/>
                </div>
                <div className={classes.control}>
                <label htmlFor='password'>Your Password</label>
                <input type='password' id='password' required ref=
{passwordInputRef}/>
                </div>
                <div className={classes.actions}>
                <button>{isLogin ? 'Login' : 'Create Account'}</button>
                <button
                    type='button'
                    className={classes.toggle}
                    onClick={switchAuthModeHandler}
```

```
                >
                    {isLogin ? 'Create new account' : 'Login with existing
account'}
                </button>
                </div>
            </form>
            </section>
        );
        }

        export default AuthForm;
```

- Notes:
  - previously we used window.locaction href='/some url' but that is better used for an initial page load and not in the middle of an ongoing app, because it bacicallyresets the entire application and all the existing states would be cleared.
  - because of the above, instead we will import and use the next/router:

```
                    import { useRouter } from 'next/router';
```

```
- then we will imlement it in our component:
```

```
                const router = useRouter();
```

```
- then we will use it in our code to replace (redirect) the url
```

```
        ...
          if (!result.error) {
         // set some auth state by saving the token
         //redirect thw uer
         router.replace('/profile');
        }
        ...
```

2. Then to protect the route the code is identical except for the line wherree we import next-auth. as shown below:

```
import { useState, useRef } from 'react';
import classes from './auth-form.module.css';
import { signIn } from "next-auth/react";
import { useRouter } from 'next/router';

async function createUser(email, password) {
const response = await  fetch('/api/auth/signup', {
    method: 'POST',
    body: JSON.stringify({email, password }),
    headers: {
        'Content-Type': 'application/json'
    }

});
const data = await response.json();
if (!response.ok){
    throw new Error(data.message || 'Something went wrong!')
}
return data;
}

function AuthForm() {
    const emailInputRef = useRef();
    const passwordInputRef = useRef();
    const [isLogin, setIsLogin] = useState(true);
    const router = useRouter();

function switchAuthModeHandler() {
    setIsLogin((prevState) => !prevState);
}

async function submitHandler(event) {
    event.preventDefault();
    const enteredEmail = emailInputRef.current.value;
    const enteredPassword = passwordInputRef.current.value;

    // add optional validation later

    if (isLogin) {
        //add login logic later:
        const result = await signIn('credentials', {
        redirect: false,
        email: enteredEmail,
        password: enteredPassword,
        });
        //console.log(result);
        if (!result.error) {
            // set some auth state by saving the token
            //redirect thw uer
            router.replace('/profile');
          }
    } else {
```

```
                try {
                    const result =  await  createUser(enteredEmail,
enteredPassword);
                    console.log(result);
                    // optional add user result feedback later
                } catch (error) {
                    console.log(error);
                    // optional add user error feedback later
                }

            }
        }

        return (
            <section className={classes.auth}>
            <h1>{isLogin ? 'Login' : 'Sign Up'}</h1>
            <form onSubmit={submitHandler}>
                <div className={classes.control}>
                <label htmlFor='email'>Your Email</label>
                <input type='email' id='email' required  ref=
{emailInputRef}/>
                </div>
                <div className={classes.control}>
                <label htmlFor='password'>Your Password</label>
                <input type='password' id='password' required ref=
{passwordInputRef}/>
                </div>
                <div className={classes.actions}>
                <button>{isLogin ? 'Login' : 'Create Account'}</button>
                <button
                    type='button'
                    className={classes.toggle}
                    onClick={switchAuthModeHandler}
                >
                    {isLogin ? 'Create new account' : 'Login with existing
account'}
                </button>
                </div>
            </form>
            </section>
        );
        }

        export default AuthForm;
```

- Notes:
    - previously we used window.locaction href='/some url' but that is better used for an initial page load and not in the middle of an ongoing app, because it bacicallyresets the entire application and all the existing states would be cleared.
    - because of the above, instead we will import and use the next/router:

```
import { useRouter } from 'next/router';
```

- then we will imlement it in our component:

```
const router = useRouter();
```

- then we will use it in our code to replace (redirect) the url

```
...
  if (!result.error) {
 // set some auth state by saving the token
 //redirect thw uer
 router.replace('/profile');
}
...
```

3. Save all reload and test - try a fresh login with valid credentials - when we click the Login button we will be redirected to the profile component page form
4. This redirects us but it does not protect the route because if we try to manualy goto the url localhost:3000/auth we will be taken to the login page and form So we do need to also implement some route gurards.

```
import { getSession } from 'next-auth/react';
import { useEffect, useState} from 'react';
import { useRouter } from 'next/router';
import AuthForm from '../components/auth/auth-form';

function AuthPage() {
const [isLoading, setIsloading] = useState(true);
// client-side page guards
// Redirect away if auth
    const router = useRouter();
    useEffect(() => {
        getSession()
        .then(session => {
        if(session){
            router.replace('/');
        }else {
            setIsloading(false);
```

```
                };
                })
            }, [router]);


            if(isLoading) {
            return <p>Loading...</p>;
            }


        return <AuthForm />;
        }

        export default AuthPage;
```

- Note this is exactly the same as the old verison except for the import line:

```
            import { getSession } from 'next-auth/react';
```

- This is somewhat similar to the front-end logic we originally used for the user-profile.js except we are using the router.relace() method instead of the window.location.href="/" to redirect.

5. Save relaod and test - try to manually go to te http://localhost/auth route after a successful login and note that we are redircted

6. Now with the newer version of next-auth V4 we can actually handle this in a much simopler way without havig to bother with useEffect as shiwn below. Flor the sake of clarity we an save the code file pages/auth.js as pages/old-auth.js and creare a new auth.js as shiwn below which takes advantage of using the newer auth-next v4 and its syntax. I suspect this simplicity may also have somethinbg to do with the mandatory use of the wrapper with the V4 auth-next as well. So the newer version of the pages/auth.js can be coded as shown below:

```
        // import { getSession } from 'next-auth/react';
        import { useSession } from 'next-auth/react';
        // import { useEffect, useState} from 'react';
        import { useRouter } from 'next/router';
        import AuthForm from '../components/auth/auth-form';

        function AuthPage() {
        // const [isLoading, setIsloading] = useState(true);
        const { data: session, status } = useSession();
        const router = useRouter();
        // client-side page guards
        // Redirect away if auth old code adapted to newer versions but
replacement below is better
            // useEffect(() => {
            //     getSession()
            //     .then(session => {
```

```
            //       if(session){
            //         router.replace('/');
            //       }else {
            //         setIsloading(false);
            //       };
            //     })
            // }, [router]);
            // if(isLoading) {
            //   return <p>Loading...</p>;
            // }
        // newer V4 suggested code syntax using useSession in place of
getSession inside of useEffect and also with useState
        // below is much simpler implementation using newer version V4
from  'next-auth/react'

        if (status === "authenticated") {
        router.replace("/");
        }

        if (status === "loading") {
        return <p className="center">Loading...</p>;
        }

        return <AuthForm />;
        }

        export default AuthPage;
```

- Notes:
    - all the workaround semantics using getSession wrapped in a useEffect and creating our own useState() is unnecessary with the newer improved version of next-auth

(409) Using the next-auth Session Provider component

- (this only applies to the legacy version as we have already inplemented this wrapper previously, althogu the nme is slightly different, for the newer versions V4 of next-auth as it is mandatory for that version) hense there is no section (409-A)

(410) Analyzing Further Authentication Requirements Explanation only hense no section (410-A) is required

- So far, in our example project we have achieved the following:
    - users can login and logout
    - users cancteate a new account for themselves
    - we also protected some specific routes
    - we also control the choices in the NAv menu based on the state of authentication
    - most of this was made possibe through the use of the next-auth package and how it helps us to manage sessions via the cookies it managaes in the browser side from the server-side of our Next app

    - The session we refer to here is basically the JWT Jason WEB Token that is saved in the cookie
    - Next Auth checks the validity of that cookie to determine if we hve an authorized session or not
    - we initiate this with either getSession() or the useSession() hooks
- Currently however we have one crucial feature missing form our application
    - having all of this page and route protection is good but what really matters is what the users can do
    - in other situtions, such as for an online shop we might only want to allow certain users to create and delete and manage products
    - so we may be in need of a more granular control over what specific users are allowed to do.
    - generally control of these types of features would be handled by specific api routes which would handle those specifi types of operations
    - therefore we want to insure that API route requests such as are only allowed from authenticated users
    - What we are getting to here is to point ut the need for API route protection as well as the ther routes we have protected as a security measure. Remember an API request can be sent manually and from other sources such as Postman for example
    - So here we see a need to have a method to enforce authentication on the server-side such as fro an API request
- In the next course section we will begin to look into how we can secure our API routes with authentication.

(411-A) . Protecting API Routes (version 4 of next-auth and newer REact / Next)

- In ths section through HOL we will learn how we can protect API routes with next-auth
- We would not want to protect the existing route we have because if we did then nonoe would be able to sign up but we should protect the route that we will use to allow the user to change theor password.
- we will wind up wth a new api route od /api/user/change-password by following the steps below:

1. Create new folder in the api folder named user
2. In that folder we will create a code file named change-password.js as shown below:

```
import { getSession } from 'next-auth/react';

async function handler(req,res) {
if(req.method !== 'PATCH') {
    return;
}

const session = await getSession({req: req});
if (!session) {
    res.status(401).json({message: 'Not authenticated!'});
    return;
}
// code blow is logic to change the password
```

```
}

export default handler;
```

- Our Logic here for this api will include the following goals and requirements:
    - we want to extract both the old and the new password from the form
    - we want to verify that the request is coming from an authenticated user and deny the api action if it is not authenticated
    - we want to also get the email address of this authenticated user
        - hint remember in our [...nextauth].js at the end e=we returned the user email to be used as part of the JWT
    - then we want to use that to query the databsse and see if we find that user there
    - then if we find that user we want to verify the match of the old password with the current one in the DB
    - Finally, if the old password tht the user entered matches the current one in the DB then we want to hash the new password and replace the hashed password stored in the DB with the new hashed passwod.
- First we verify that the correct method is being deployed (we could use POST, PUT, or PATCH). We will require the PATCH method:

```
if(req.method !== 'PATCH') {
    return;
}
```

- Next we want to check if the request is coming from an authenticated user or not and we will import:

```
import { getSession } from 'next-auth/react';
```

- note that other than the import line the remainder of the code in this section is identical to the (411) instructions.
- and use the getSession() hook as a serverside function call:

```
const session = getSession({req: req});
```

- we define a const by calling getSession and passing a configuration object that will invoke the req key of the getSession object
- getSession() will then look into the incoming request and verify if the JWT toekn hs been passed properly with that request. If the JWT is there it will validate and extract the data from the cookie and it wll then give is the session objecrt as the (internal) return.

- This is a promise so we convert out function to use async / await.
- Then we check to see f we have a session object or not and if we do not we return an error and return from the API route handler() function:

```
        if (!session) {
            res.status(401).json({message: 'Not authenticated!'});
            return;
        }
```

- At this point we have created all the code we need to protect this API route so that only authenticated users will be able to use it
- In the next section we will add the code with the logic for changing the password.
- we will actually revisit this and revise it in the next course section as this will not quite work correctly

(412-A) Adding the 'Change PAssword Logic to the new API route /api/user/change-password

- This has some major differences to work properly between V4 nd V4 of next auth.

1. For one thing with V4 next auth we need to ceate an environment variable with a secret key in the next.confog.js code file in order for this to work because we need to key for decryption of the JWT with V4:

```
module.exports = {
    env: {
        NEXTAUTH_SECRET: "onebanannatwobananna3bananna4",
    }
}
```

2. Then we need to add this key in the authOptions configuration object in the [...nextauth].js code file

```
import NextAuth from "next-auth";
import CredentialsProvider from "next-auth/providers/credentials";
import { connectToDatabase } from "../../../lib/db";
import { verifyPassword } from "../../../lib/auth";

export const authOptions = {
  secret: process.env.NEXTAUTH_SECRET,
    session: {
        jwt: true
    },
  providers: [
    CredentialsProvider({
      async authorize(credentials) {
        //  console.log('authorize has been called');
        const client = await connectToDatabase();
```

```
                const usersCollection = client.db().collection('users');
                //previously we set the credentials properties in a
configuration
                //  console.log('made db connection')
                //  console.log('credentials.email: ', credentials.email)
                const user = await usersCollection.findOne({email:
credentials.email})
                //  console.log('found: ', user);
                if(!user){
                  // console.log('no user found')
                    client.close();
                    throw new Error('user not found!')
                }
                // if we get here it means we found the email adderess
match and now we need to validate the password
                // keep in mind that the stored password has been hashed
                //  console.log('credentials.password: ',
credentials.password)
                const isValid = await verifyPassword(credentials.password,
user.password);
                if (!isValid) {
                  // console.log('invalid password', isValid)
                    client.close();
                    throw new Error('Login Failed!')
                }
                    // if we get this far we know we have an existing
email and a validated password
                    // therefore we consider the user to be logged in so
we will return an object:
                    // console.log('password valid: ', isValid);
                    client.close();
                    return {email: user.email};
      },
    }),
  ],
};

export default NextAuth(authOptions);
```

3. Then we can create the api/user/change-password.js as shown below:

```
// note this is a different import than V3 uses
import { getServerSession } from "next-auth";
import { connectToDatabase } from '../../../lib/db';
import { verifyPassword, hashPassword } from '../../../lib/auth';
import { authOptions } from "../auth/[...nextauth]";

async function handler(req,res) {
if(req.method !== 'PATCH') {
    return;
}
```

```
 //const session = await getSession({req: req});
 // also different syntax to get the session which requires the 3rd
argument
const session = await getServerSession(req, res, authOptions); // Use
instead of getSession
console.log('session is:', session);
if (!session) {
    res.status(401).json({message: 'Not authenticated!'});
    return;
}
// code blow is logic to change the password
// because the email a dress is encoded in the JWT token we can access the
email from the session object.

const userEmail = session.user.email;
console.log('user email is: ', userEmail);
const oldPassword = req.body.oldPassword;
const newPassword = req.body.newPassword;

//connect to db
const client = await connectToDatabase();
const usersCollection = client.db().collection('users');
const user = await usersCollection.findOne({email: userEmail})
// check to be sure we found a user
if(!user){
    res.status(404).json({message: 'User not found in DB!'})
    client.close();
    return;
}
// note below is the hashed password
const currentPassword = user.password
//verify the password
const passwordsAreEqual =  await verifyPassword(oldPassword,
currentPassword);
if(!passwordsAreEqual){
    res.status(403).json({message: 'Invalid Old Password'});
    client.close();
    return;
}
// hash the new password before we save it to the DB
const hashedPassword = await hashPassword(newPassword);

// now we can alow the user to change the password i we got this far
// note we will return later and add a try catch for error handling soon
const result = await usersCollection.updateOne(
    {email: userEmail},
    {$set: {password: hashedPassword}});

 client.close();
 res.status(200).json({message: 'Password Successfully Updated!'});


}
```

```
export default handler;
```

- see comments in code above for explantion of difference between this V4 and the legacy V3 use of next-auth
- Instead of getSession we revise this to import and implement getServerSession()
- we also had to create an environment variable and use it as s cypher key in the authOptions config object.

(413-A) Semding the 'Change Password" Request from the Form Component FrontEnd (no changes between versions in this section)

- The code for the new V4 next-auth does not come into play with the profile-form or the user-profile pages as they are, so the code we added will be the same in those two code files, profile-form.js and user-profile.js for both the old and the new versions of Next, React and next-auth.
- In this section we will 'wire-up' our ProfileForm component, components/profile/profile-form.js to make the call to the new API route we just created.
- Our goals will include:
    - handle the form submission, by extracting the entered values in the form fields
    - send the http request to out new api route, /api/user/change-password
    - or we could send the extracted data to the parent component, the pages/profile, and actually send the http request from there.
    - We will start.below, with getting the entered data from the form

1. Edit the profile-form.js code file as shown below:

```
import classes from './profile-form.module.css';
import { useRef } from 'react'


function ProfileForm(props) {

  const oldPasswordRef = useRef();
  const newPasswordRef = useRef();

function submitHandler(event) {
  event.preventDefault();
  const enteredOldPassword = oldPasswordRef.current.value;
  const enteredNewPassword = newPasswordRef.current.value;
// optional we chould add some client side validation here later but we
already do have it on the server side
// call the function passed by props fromthe parent
props.onChangePassword({
  oldPassword: enteredOldPassword,
  newPassword: enteredNewPassword
})

}
  return (
```

```
      <form className={classes.form} onSubmit={submitHandler}>
        <div className={classes.control}>
          <label htmlFor='new-password'>New Password</label>
          <input type='password' id='new-password' ref={newPasswordRef}/>
        </div>
        <div className={classes.control}>
          <label htmlFor='old-password'>Old Password</label>
          <input type='password' id='old-password' ref={oldPasswordRef}/>
        </div>
        <div className={classes.action}>
          <button>Change Password</button>
        </div>
      </form>
    );
}

export default ProfileForm;
```

- Notes:
    - we create a new submitHandler() function within our component function and we be sure to use the event to prevent the default form submit action because we want to handle that ourselves with JavaScript:

```
            ...
            function ProfileForm() {

            function submitHandler(event) {
            event.preventDefault();

            }
            ...
```

- then we set this as the form action for onSubmit:

```
            ...
            return (
                <form className={classes.form} onSubmit={submitHandler}>
            ...
```

- we will import useRef:

```
    import { useRef } from 'react'
```

- and define the two refs to get our form field values:

```
  const oldPasswordRef = useRef();
  const newPasswordRef = useRef();
```

- then implement them in the form fields:

```
<div className={classes.control}>
        <label htmlFor='new-password'>New Password</label>
        <input type='password' id='new-password' ref={newPasswordRef}/>
      </div>
      <div className={classes.control}>
        <label htmlFor='old-password'>Old Password</label>
        <input type='password' id='old-password' ref={oldPasswordRef}/>
      </div>
```

- we can get these values in our handler function:

```
const enteredOldPassword = oldPasswordRef.current.value;
  const enteredNewPassword = newPasswordRef.current.value;
}
```

- we decided rather than make the api request here we would instead pass the data up to the parent component and make the request there so we added the props aregument to this compoinent function:

```
function ProfileForm(props) {...}
```

- then we call the function that we will be oassing from the parent component:

```
props.onChangePassword({
  oldPassword: enteredOldPassword,
  newPassword: enteredNewPassword
})
```

2. Now we will setup the function to performthe submission to the api route in the parent component UserProfile, components/profile/user-profile as shown below:

```
import ProfileForm from './profile-form';
import classes from './user-profile.module.css';

function UserProfile() {
```

```
  async function changePasswordHandler(passwordData){
    const response = await fetch('/api/user/change-password', {
      method: 'PATCH',
      body: JSON.stringify(passwordData),
      headers: {
        'Content-Type': 'application/json'
      }
    });
    const data = await response.json();
    console.log(data);
  }

  return (
    <section className={classes.profile}>
      <h1>Your User Profile</h1>
      <ProfileForm onChangePassword={changePasswordHandler} />
    </section>
  );
}

export default UserProfile;
```

- Notes:
  - we reated a new function in the parent and passed that as a prop to the child function:

```
 return (
    <section className={classes.profile}>
      <h1>Your User Profile</h1>
      <ProfileForm onChangePassword={changePasswordHandler} />
    </section>
  );
```

- Our function then expects the argument passed as an object and will make the http request to the api route:

```
  async function changePasswordHandler(passwordData){
    const response = await fetch('/api/user/change-password', {
      method: 'PATCH',
      body: JSON.stringify(passwordData),
      headers: {
        'Content-Type': 'application/json'
      }
    });
    const data = await response.json();
    console.log(data);
  }
```

- Note abve we are currently only logging the respose to the console for now, typically we woild want to provide better feedback vi the Ul forthe user. but for now, as a developer we will easily be able to check whether this worked via the console.log()

3. Save all relaod and test.
    - look at the mongodb users collection at the document fr this user and note the password string
    - open the dev-tools console so we can see the console.log
    - now we should be able to change the password and see the confirmation in the console.log:

```
{
    "message": "Password Successfully Updated!"
}
```

```
- look again at the mongodb password again and note that it has changed,
albeit it a hashed password it should be different than it was previously.
```

4. Log out then try to log back in using the new password
    - our login should be successful using the new credentials
5. Now try to change the password but enter an incorrect old password and note what we see in the console: PATCH http://localhost:3000/api/user/change-password 403 (Forbidden) { "message": "Invalid Old Password" }
    - above we see it is forbidden with a 403 error which is corrct as it should be forbidden because we entered the incorrect old password