- ⌚ 2 hours
- ✅ Completed

Twitter Lite... with Authentication!

At this point, you've set up an API for tweets and connected a frontend application to interact with your API. You've also added a Users model to your project to prepare for implementing user authentication.

In this project, you'll be writing Vanilla JS to render frontend elements. You have been using string template literals to render data from the database as HTML. Later on in the class, you'll use React to render components and build frontend clients. In the next portion of the project, you will set up the Users router and protect your tweet resources from unauthenticated users.

When you have completed the second part of the project, your application should have the following features:

1. Users routes
2. Protected tweet resources
3. User registration
4. User log-in
5. Authenticated tweet creation
6. A form to create tweets

Phase 1: Users router

Begin by creating a users routes module at `routes/users.js`. Require `express` to initialize a router with `express.Router()` and make sure to export the router you just initialized.
In your `app.js` file, require the `usersRouter` you have just created and have your application connect the `/users` path to the `usersRouter`.
The users router will need to use a couple of helper functions that currently exist in the tweets router: `asyncHandler` and `handleValidationErrors`. Go ahead and create a `utils.js` file within the root of your project and then move those two functions into that file.
Now add a route in the user router to handle POST requests to `localhost:8080/users`. In this route, use `express-validator` to validate the

params. Make sure to import `check` from `express-validator` as well as the `asyncHandler` and `handleValidationErrors` helper functions from the `../utils.js` file:

```
const { check } = require("express-validator");
const { asyncHandler, handleValidationErrors } =
require("../utils");
```

In your POST route, begin by wrapping the route handler with the `asyncHandler` function because there will be some asynchronous logic inside the route.
Add the following validations to validate your user authentication data so that you can render error messages upon submission of bad registration data. Go ahead and also use the `handleValidationErrors` helper middleware function.

```
const validateUsername =
  check("username")
    .exists({ checkFalsy: true })
    .withMessage("Please provide a username");

const validateEmailAndPassword = [
  check("email")
    .exists({ checkFalsy: true })
    .isEmail()
    .withMessage("Please provide a valid email."),
  check("password")
    .exists({ checkFalsy: true })
    .withMessage("Please provide a password."),
];

router.post(
  "/",
  validateUsername,
  validateEmailAndPassword,
  handleValidationErrors,
  asyncHandler(async (req, res) => {
    // TODO: User creation logic
  })
);
```

Within the asynchronous route handler function, destructure the `username`, `email`, and `password` from the `body` of your request. You'll use these properties to create a new user in your database. For now, leave a `TODO` note to create the user.
Now to actually create the user, install and require the `bcryptjs` library to use the `bcrypt.hash()` method. Invoke the `bcrypt.hash()` method with the user's `password` and a salt round of `10`. Await the generation of a `hashedPassword` before creating the user.
At this point, make sure you have required the `db` from `../db/models`. Destructure your `User` model from your `db` module and await the creation of your user (`User.create()`). In order to protect the user's credentials, create the user with the `hashedPassword` instead of the `password` from the form request body.

At this point, your user creation route should look something like this:

```
router.post(
  "/",
  validateUsername,
  validateEmailAndPassword,
  handleValidationErrors,
  asyncHandler(async (req, res) => {
    const { username, email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    const user = await User.create({ username, email,
hashedPassword });
    // TODO: Generate JSON Web Token (access token)
    // TODO: Render user in JSON
  })
);
```

Once the user has been created, you want to return an access token. You'll be using a JWT as the access token. To generate and decode the JWT, use the jsonwebtoken npm package. Begin by running `npm install jsonwebtoken`.

There are a few configuration steps to set up JWT generation. You need to set a *secret key* and a number representing how many seconds before the token expires. Add a `JWT_SECRET` and `JWT_EXPIRES_IN` variable to your `.env` file.

To generate a secret, open up your node repl in your terminal by running `node` and then use the built-in crypto module to generate your `JWT_SECRET` key:

```
require("crypto").randomBytes(32).toString("hex");
```

For the `JWT_EXPIRES_IN` set it to `604800`, which is the number of seconds for one week. Then update your `config/index.js` file to use these environment variables:

```
module.exports = {
  environment: process.env.NODE_ENV || "development",
  port: process.env.PORT || 8080,
  db: {
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE,
    host: process.env.DB_HOST,
  },
  jwtConfig: {
    secret: process.env.JWT_SECRET,
    expiresIn: process.env.JWT_EXPIRES_IN,
  },
};
```

Finally, let's put this JWT generation logic inside of an `auth.js` file. Create an `auth.js` file in the root of your project and require the `jsonwebtoken` package as well as your `jwtConfig` variables from your `./config` module.

Now define a function called `getUserToken`. This function will take a user object as a parameter and then create a payload from the user. Then, it uses the `jsonwebtoken.sign()` function to generate a token.

```
const jwt = require("jsonwebtoken");
const { jwtConfig } = require("./config");

const { secret, expiresIn } = jwtConfig;

const getUserToken = (user) => {
  // Don't store the user's hashed password
  // in the token data.
  const userDataForToken = {
    id: user.id,
    email: user.email,
  };

  // Create the token.
  const token = jwt.sign(
    { data: userDataForToken },
    secret,
    { expiresIn: parseInt(expiresIn, 10) } // 604,800 seconds = 1
week
  );

  return token;
};

module.exports = { getUserToken };
```

Notice how in the code above, we had to convert `expiresIn` from a string to an integer because environment variables declared in the `.env` file are strings, and according to the jsonwebtoken docs, anything that gets set to the `expiresIn` as an **integer** will be calculated as **seconds**, and **strings** will be calculated in **milliseconds**.

Now, go back to the users routes module and import the `getUserToken` function from the `../auth` module. Then update the user creation route to call the `getUserToken` function and return a token to the user when they sign up for an account. Your user creation route should look something like this:

```
router.post(
  "/",
  validateUsername,
  validateEmailAndPassword,
  handleValidationErrors,
  asyncHandler(async (req, res) => {
    const { username, email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    const user = await User.create({ username, email,
hashedPassword });

    const token = getUserToken(user);
    res.status(201).json({
      user: { id: user.id },
      token,
    });
  })
);
```

Once you've confirmed that this is working correctly on Postman, let's put this JWT token to use by protecting the tweets resources and only allowing signed in users to access the tweets routes!

Phase 2: Protecting the tweets resources

It's time to protect the tweet resources in your API! On each request to tweet routes, do the following:

1. Parse out the JWT token from the request header.
2. Decode the JWT.
3. Find the user based on the JWT payload.

If all of the steps above succeed, then the user is considered to be signed in and can access the tweet resources.

To carry out the plan above, go to back to your `auth.js` file. Here, you'll want to use two middleware functions. The first will parse out the JWT token from the request header. The second function decodes the JWT to find the user and stores the user in the `req` object.
For the first middleware function, let's use the `express-bearer-token` middleware to do this parsing. Run `npm install express-bearer-token` to add the middleware to your application. Begin by requiring the middleware as `bearerToken` as well as requiring your `User` model from the database models.
Define a middleware function named `restoreUser` that takes in the parameters: `req`, `res`, and `next`. Begin by destructuring the JWT `token` from the `req` object. If the token is invalid, return an invocation of the `next` function.
If the token is valid, invoke the `jwt.verify()` method with four arguments:

- JWT `token` from the request body;
- JWT `secret` from your configuration file;
- `null` options; and
- An asynchronous function that takes in an error and a `jwtPayload` (more steps below).

Currently, your `restoreUser` function should look something like this. Notice if we don't get a token in the request at all we set the "WWW-Authenticate" header to "Bearer" and return a "401 Unauthorized" status code.

```
const restoreUser = (req, res, next) => {
  const { token } = req;

  if (!token) {
    return res.set("WWW-Authenticate",
"Bearer").status(401).end();
  }

  return jwt.verify(token, secret, null, async (err, jwtPayload)
=> {
    // TODO: Define asynchronous function for jwtPayload logic
  });
};
```

In the asynchronous function, check whether there is an error object. If so, set the error's status to be `401` and return an invocation of the `next` function passing in the error object:

```
if (err) {
  err.status = 401;
  return next(err);
}
```

If there is no error, extract the user's `id` from the `data` property of your `jwtPayload`:

```
const { id } = jwtPayload.data;
```

Then use a `try/catch` block to await the fetch of a user (by using `User.findByPk()`) or catch an error to return an invocation of the `next` function:

```
try {
  req.user = await User.findByPk(id);
} catch (e) {
  return next(e);
}
```

Now check to see whether the `req` object has an associated `user` property. If there isn't a valid user, return a response that sets the "WWW-Authenticate" header with a value of "Bearer" and sends a "401 Unauthorized" status code before invoking the `next` function:

```
if (!req.user) {
  return res.set("WWW-Authenticate",
"Bearer").status(401).end();
}
```

Your complete `restoreUser` function should look something like this:

```
const restoreUser = (req, res, next) => {
  const { token } = req;

  if (!token) {
    return res.set("WWW-Authenticate",
"Bearer").status(401).end();
  }

  return jwt.verify(token, secret, null, async (err, jwtPayload)
=> {
    if (err) {
      err.status = 401;
      return next(err);
    }
    const { id } = jwtPayload.data;

    try {
      req.user = await User.findByPk(id);
    } catch (e) {
      return next(e);
    }

    if (!req.user) {
      return res.set("WWW-Authenticate",
"Bearer").status(401).end();
    }

    return next();
  });
};
```

Now, take a moment to organize your `bearerToken` generator function and the `restoreUser` function into an array named `requireAuth`. Update your module exports to export the two functions as one `requireAuth` array:

```
const requireAuth = [bearerToken(), restoreUser];
module.exports = { getUserToken, requireAuth };
```

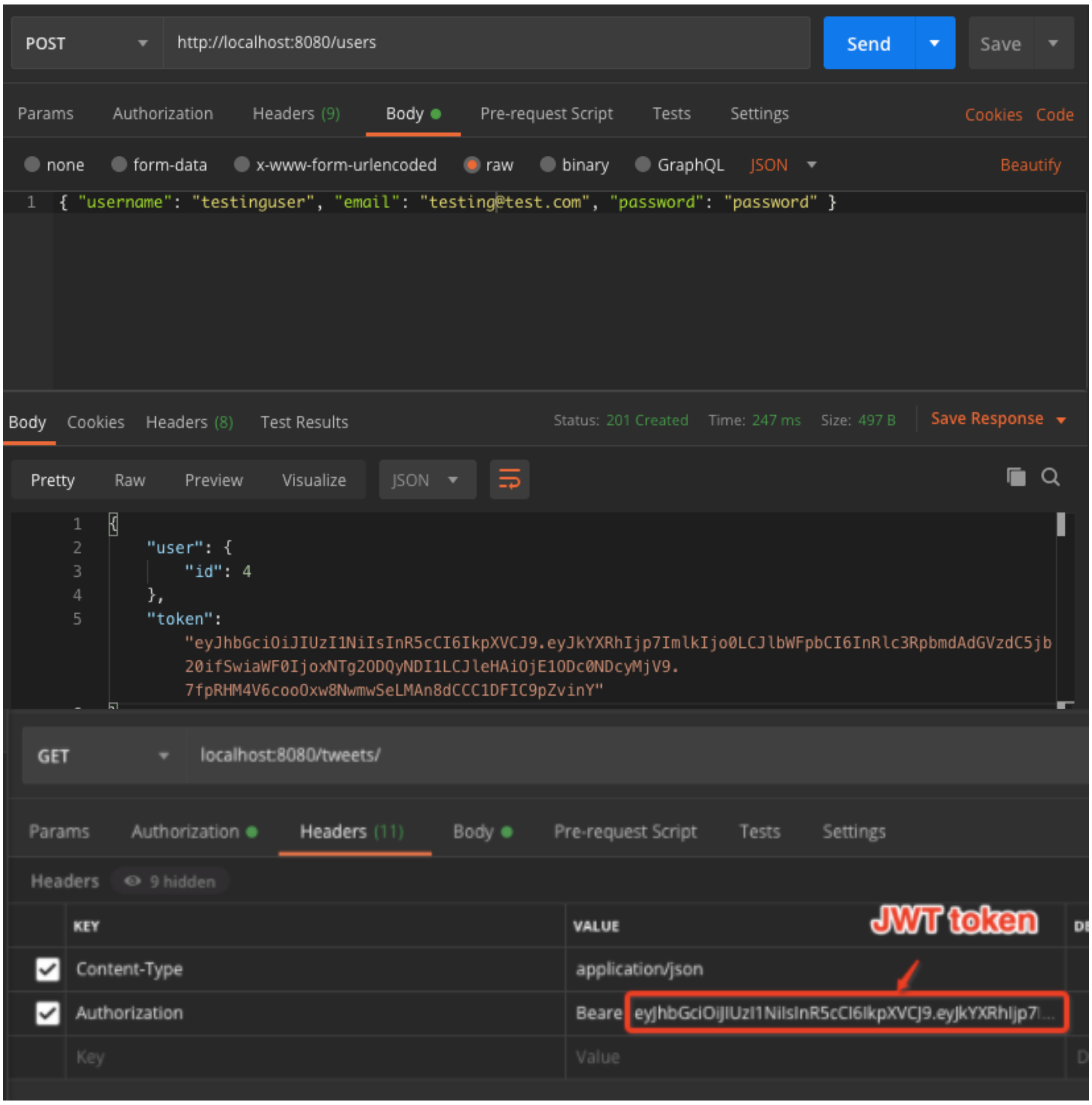
Now it's time to add the `requireAuth` middleware to the tweets router! Add the following code to your `routes/tweets.js` file to use the user authentication functions you just created:

```
const { requireAuth } = require("../auth");

// REST OF FILE NOT SHOWN

router.use(requireAuth);
```

Verify that your tweet protection works by trying to make a GET request to `localhost:8080/tweets` from Postman. You should see a `401` response. To fix this, create another user by posting to `localhost:8080/users`. This time, when the token comes back in the response, copy that response and paste it in to the *Authorization header*:



http://localhost:8080/users

```
{ "username": "testinguser", "email": "testing@test.com", "password": "password" }
```

Status: 201 Created Time: 247 ms Size: 497 B

```
{
    "user": {
        "id": 4
    },
    "token":
        "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7ImlkIjo0LCJlbWFpbCI6InRlc3RpbmdAdGVzdC5jb
        20ifSwiaWF0IjoxNTg2ODQyNDI1LCJleHAiOjE1ODc0NDcyMjV9.
        7fpRHM4V6cooOxw8NwmwSeLMAn8dCCC1DFIC9pZvinY"
```

localhost:8080/tweets/

	KEY	VALUE
✓	Content-Type	application/json
✓	Authorization	Beare eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7I...
	Key	Value

Then try to get all of the tweets again. This time, your request should pass through the `requireAuth` middleware functions successfully. Now that your tweets are protected, let's add a view for users to sign up from the client application!

Phase 3: Sign up for user account from the client

Begin by adding a GET route in the `express-apis-frontend/index.js` file to render a `sign-up` template:

```
app.get("/sign-up", (req, res) => {
  res.render("sign-up");
});
```

Use the snippet below to create a registration form for the `sign-up.pug` template.

```
extends layout.pug

block content
  div(class="sign-up-container")
    div(class="errors-container")
    h2 Sign Up
    form(class="sign-up-form")
      .form-group
        label(for='username') Username
        input#username.form-control(type='text' name='username'
placeholder="Username")
      .form-group
        label(for='email') Email address
        input#email.form-control(type='email', name="email",
placeholder='Enter email')
      .form-group
        label(for='password') Password
        input#password.form-control(type='password',
name="password", placeholder='Password')
      button.btn.btn-primary(type='submit') Sign Up
    a(href='/log-in') Already have an account? Log in here.
  script(src="js/sign-up.js")
```

In the code above, there's also an anchor element to navigate to the `/log-in` route that we'll implement next. Since the user will use the email and password input fields for the later log in form, let's move those to an *includes* file to DRY up our code.

Move the following `.form-group` elements from your `sign-up.pug` file to a new `views/includes/auth-form-fields.pug` file:

```
.form-group
  label(for='email') Email address
  input#email.form-control(type='email', name="email",
placeholder='Enter email')
.form-group
  label(for='password') Password
  input#password.form-control(type='password',
name="password", placeholder='Password')
```

Now take a moment to refactor your `sign-up.pug` template to use your new *includes* file.

At the bottom of the `sign-up.pug` template, notice that a `js/sign-up.js` script has already been imported. Set up this `sign-up.js` script inside your `express-apis-frontend/public/js` directory. In the script, you want to add a *submit* event listener to the sign up form (hint: search for the element with a class of `sign-up-form`) and an asynchronous callback function to handle the sign up request. Remember that `submit` events automatically prompt a GET request to re-render the page. Make sure to prevent the form from re-rendering by using `e.preventDefault()` at the beginning of your event listener:

```
const signUpForm = document.querySelector(".sign-up-form");

signUpForm.addEventListener("submit", async (e) => {
  e.preventDefault();
  // Sign up logic here
});
```
Generate a new `FormData` object with your `signUpForm`:
```
const formData = new FormData(signUpForm);
```
Now use the formData.get() method to retrieve the `username`, `email`, and `password` from the form. You can use the form values to declare a `body` variable:
```
const email = // TODO: Get email
const password = // TODO: Get password
const username = // TODO: Get username

const body = { email, password, username };
```
When a submit event happens, parse out the data from the form and then use a `fetch` call to make a POST request to `localhost:8080/users` to create the user in a `try` block. Set the `fetch` call's `method` to be "POST" and the `body` to be a JSON string of the form fields (hint: use `JSON.stringify({ email, password, username })`). Lastly, set a `Content-Type` header of "application/json".

Your fetch call should look something like this:

```
try {
  const res = await fetch("http://localhost:8080/users", {
    method: "POST",
    body: JSON.stringify(body),
    headers: {
      "Content-Type": "application/json",
    },
  });
}
```
Fetch calls don't throw errors on error status code responses. Note that the Fetch API only rejects network failure errors. This means you need to manually handle response errors by checking the response's `ok` property. Add the code snippet below in your `try` block to manually throw an error if the fetch request was rejected:
```
if (!res.ok) {
  throw res;
}
```
When the response body returns, store the JWT token and the user id in `localStorage` and redirect the user to the home page:
```
const {
  token,
  user: { id },
} = await res.json();

localStorage.setItem("TWITTER_LITE_ACCESS_TOKEN", token);
localStorage.setItem("TWITTER_LITE_CURRENT_USER_ID", id);
```

If an error happens, await the parsing of the error object as JSON and query for your element with the class of `errors-container`. Your `try/catch` block should currently look something like this:

```
try {
  // CODE OMITTED FOR BREVITY

  if (!res.ok) {
    throw res;
  }
  const {
    token,
    user: { id },
  } = await res.json();

  // CODE OMITTED FOR BREVITY
} catch (err) {
  if (err.status >= 400 && err.status < 600) {
    const errorJSON = await err.json();
    const errorsContainer = document.querySelector(".errors-
container");
    // TODO: Generate and render errors
  } else {
    // TODO: Alert user about bad internet connection
  }
}
```

Like how you rendered your list of tweets when you first connected the frontend to your application, you'll set the `innerHTML` of your `errorsContainer` to a stringified HTML block. Sometimes there might not be an errors array returned, and in those cases, go ahead and declare the `errorsHtml` array to display a generic "Something went wrong" message.

```
// errorsHtml block to alert user of error:
let errorsHtml = [
  `
    <div class="alert alert-danger">
        Something went wrong. Please try again.
    </div>
  `,
];
```

Now take a look at the global error handling function in your `app.js` and notice how you are rendering an `errors` property in your JSON response. This `errors` property is the array of error messages from your validations. Iterate through the `errors` array and display them:

```
if (errors && Array.isArray(errors)) {
  errorsHtml = errors.map(
    (message) => `
    <div class="alert alert-danger">
        ${message}
    </div>
  `
  );
  // TODO: Join errorsHtml and set the
errorsContainer.innerHTML
  // TODO: Error rendering
}
```

Since the Fetch API only throws errors on network errors (like if your internet cut out), so then we would also need to account for those types of errors. To account for these cases, go ahead and use the JavaScript `alert()` function to let the user know to check their internet connection:

```
alert(
  "Something went wrong. Please check your internet connection
and try again!"
);
```

If the create request succeeds, redirect the user back to the home page right after you set your `localStorage` items. Use the `window.location.href` property so that the user can see all existing tweets:

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

At this point, your `sign-up.js` file should look something like this:

```
const signUpForm = document.querySelector(".sign-up-form");

signUpForm.addEventListener("submit", async (e) => {
  e.preventDefault();
  const formData = new FormData(signUpForm);

  const username = formData.get("username");
  const email = formData.get("email");
  const password = formData.get("password");

  const body = { email, password, username };
  try {
    const res = await fetch("http://localhost:8080/users", {
      method: "POST",
      body: JSON.stringify(body),
      headers: {
        "Content-Type": "application/json",
      },
    });

    if (!res.ok) {
      throw res;
    }

    const {
      token,
      user: { id },
    } = await res.json();
    // storage access_token in localStorage:
    localStorage.setItem("TWITTER_LITE_ACCESS_TOKEN", token);
    localStorage.setItem("TWITTER_LITE_CURRENT_USER_ID", id);
    // redirect to home page to see all tweets:
    window.location.href = "/";
  } catch (err) {
    if (err.status >= 400 && err.status < 600) {
      const errorJSON = await err.json();
      const errorsContainer = document.querySelector(".errors-
container");
      let errorsHtml = [
        `
        <div class="alert alert-danger">
            Something went wrong. Please try again.
        </div>
      `,
      ];
      const { errors } = errorJSON;
      if (errors && Array.isArray(errors)) {
        errorsHtml = errors.map(
          (message) => `
          <div class="alert alert-danger">
              ${message}
          </div>
        `
        );
      }
      errorsContainer.innerHTML = errorsHtml.join("");
    } else {
      alert(
        "Something went wrong. Please check your internet
connection and try again!"
      );
    }
  }
});
```

When a user signs up and is redirected to the home page, the request for all tweets should be failing because that request doesn't currently have the correct authentication headers. Let's fix this.

In the `public/js/index.js` file, add the `Authorization` header to your fetch call.

```
const res = await fetch("http://localhost:8080/tweets", {
  headers: {
    Authorization: `Bearer ${localStorage.getItem(
      "TWITTER_LITE_ACCESS_TOKEN"
    )}`,
  },
});
```

Then, add the following code into your "DOMContentLoaded" event listener to redirect users to the log-in route if a user is not logged in. The next step is to implement the log-in route!

```
if (res.status === 401) {
  window.location.href = "/log-in";
  return;
}
```

Now that you've implemented a sign-up flow, let's next implement a log-in flow!

Phase 4: Setting up the log-in flow

Go back to the API and add a route to the users router that allows users to fetch a new token, effectively logging them in. Do this by creating POST route to the `/token` path and using the `validateEmailAndPassword` middleware. Wrap your asynchronous route handler function with your `asyncHandler` helper method and destructure the `email` and `password` from your request body. Begin by using the `email` to find your user instance:

```
router.post(
  "/token",
  validateEmailAndPassword,
  asyncHandler(async (req, res, next) => {
    const { email, password } = req.body;
    const user = await User.findOne({
      where: {
        email,
      },
    });

    // TODO: Password validation and error handling
    // TODO: Token generation
  })
);
```

Now we need a way to validate whether or not the provided password is correct. Let's implement a password validation function in the User model as an instance method! Add the instance method below to your `user.js` model file:

```
User.prototype.validatePassword = function (password) {
  // Note that since this function is a model instance method,
  // `this` is the user instance here:
  return bcrypt.compareSync(password,
this.hashedPassword.toString());
};
```

Note how the `hashedPassword` has to be converted back from its *binary* format to a *string* format before it can be compared with the provided `password`.
Back in the `/token` route handler, verify whether the a valid `user` has been found and use the `validatePassword` instance method to verify whether or not the user provided the correct credentials. If a valid user was found with a valid password, return a `token` and the user's `id` in the JSON response. If a valid user was not found or the password was incorrect, generate an error and return an invocation of the `next` function passing in the error:

```
if (!user || !user.validatePassword(password)) {
  const err = new Error("Login failed");
  err.status = 401;
  err.title = "Login failed";
  err.errors = ["The provided credentials were invalid."];
  return next(err);
}
```

Make sure to generate a token by invoking your `getUserToken()` function with the `user` fetched from your database. Lastly, render a JSON response with the `token` and the user's `id`:

```
const token = getUserToken(user);
res.json({ token, user: { id: user.id } });
```

Next, go back to the frontend and set up a `localhost:4000/log-in` route. The logic is fairly similar to the sign-up route. First, set up a GET route in your `express-apis-frontend/index.js` file to render a `log-in` template:

```
app.get("/log-in", (req, res) => {
  res.render("log-in");
});
```

Then create a `views/log-in.pug` template with the snippet below:

```
extends layout.pug

block content
  div(class="log-in-container")
    div(class="errors-container")
    h2 Log In
    form(class="log-in-form")
      include includes/auth-form-fields
      button.btn.btn-primary(type='submit') Log In
    a(href='/sign-up') Don't have an account? Sign up here.
  script(src="js/log-in.js")
```

Now create a `public/js/log-in.js` file and add a *submit* event listener to do fairly similar logic to the event listener in your `public/js/sign-up.js` file:

```
const logInForm = document.querySelector(".log-in-form");

logInForm.addEventListener("submit", async (e) => {
  e.preventDefault();
  const formData = new FormData(logInForm);
  const email = formData.get("email");
  const password = formData.get("password");
  const body = { email, password };
  try {
    const res = await
fetch("http://localhost:8080/users/token", {
      method: "POST",
      body: JSON.stringify(body),
      headers: {
        "Content-Type": "application/json",
      },
    });
    if (!res.ok) {
      throw res;
    }
    const {
      token,
      user: { id },
    } = await res.json();
    // storage access_token in localStorage:
    localStorage.setItem("TWITTER_LITE_ACCESS_TOKEN", token);
    localStorage.setItem("TWITTER_LITE_CURRENT_USER_ID", id);
    // redirect to home page to see all tweets:
    window.location.href = "/";
  } catch (err) {
    if (err.status >= 400 && err.status < 600) {
      const errorJSON = await err.json();
      const errorsContainer = document.querySelector(".errors-
container");
      let errorsHtml = [
        `
        <div class="alert alert-danger">
            Something went wrong. Please try again.
        </div>
       `,
      ];
      const { errors } = errorJSON;
      if (errors && Array.isArray(errors)) {
        errorsHtml = errors.map(
          (message) => `
          <div class="alert alert-danger">
              ${message}
          </div>
        `
        );
      }
      errorsContainer.innerHTML = errorsHtml.join("");
    } else {
      alert(
        "Something went wrong. Please check your internet
connection and try again!"
      );
    }
  }
});
```

At this point, you might be thinking to yourself that this all seems super repetitive. In server-side Node.js code, you've been using CommonJS modules to DRY up your code. Unfortunately, you haven't learned about client-side modules yet, but when you get to React, you'll learn more about using a frontend module system in order to clean up your code!

Now create a new user, and then go to the log in form to verify that the log in process works when you log in with the created user credentials.

Phase 5: Creating tweets from the frontend application

Let's first reorganize the API so that tweets belong to a specific user. First, rollback all of your migrations by running the command below:

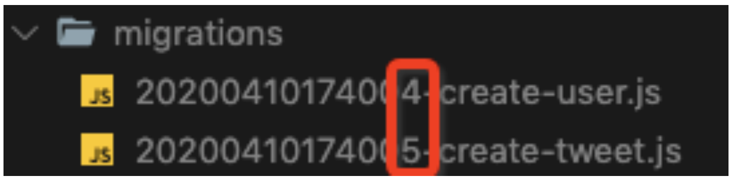
```
npx dotenv sequelize db:migrate:undo:all
```

Then since we need to associate tweets with a user, add a foreign key of `userId` so that tweets can belong to users. Here's what your updated tweets migration file should look like:

```
"use strict";
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable("Tweets", {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER,
      },
      message: {
        type: Sequelize.STRING(280),
        allowNull: false,
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE,
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE,
      },
      userId: {
        allowNull: false,
        type: Sequelize.INTEGER,
        references: {
          model: "Users",
          key: "id",
        },
      },
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable("Tweets");
  },
};
```

However, there's a problem right now. The `create-tweet` migration file was created first, and because of the timestamp in the first part of the file name, that specific migration will run first. The problem is that when we try to reference the Users table in the `references` definition part of the migration, the Users table has not actually been created yet. We're going to take a *hacky* approach to resolve this issue and make the `create-user` migration run first.

To be clear, this is absolutely not how you would want to handle migrations in production. This whole "undo all migrates and then switch up the order of migrations" is purely for learning purposes and to help you get a better of how migrations work. So, go ahead and rename the `create-tweet` migration file so that the timestamp comes later than the `create-user` migration file:



Now, that you've updated the migrations to properly configure the tables in your database, update the models to also reflect this relationship. This is what your tweet model should look like:

```
"use strict";
module.exports = (sequelize, DataTypes) => {
  const Tweet = sequelize.define(
    "Tweet",
    {
      message: {
        type: DataTypes.STRING(280),
        allowNull: false,
      },
    },
    {}
  );
  Tweet.associate = function (models) {
    Tweet.belongsTo(models.User, {
      as: "user",
      foreignKey: "userId",
    });
  };
  return Tweet;
};
```

This is what your user model should look like after you add the associations:

```
"use strict";
const bcrypt = require("bcryptjs");

module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define(
    "User",
    {
      email: { type: DataTypes.STRING, allowNull: false, unique:
true },
      username: { type: DataTypes.STRING, allowNull: false,
unique: true },
      hashedPassword: {
        type: DataTypes.STRING.BINARY,
        allowNull: false,
      },
    },
    {}
  );
  User.associate = function (models) {
    User.hasMany(models.Tweet, {
      as: "tweets",
      foreignKey: "userId",
    });
  };

  User.prototype.validatePassword = function (password) {
    // because this is a model instance method, `this` is the
user instance here:
    return bcrypt.compareSync(password,
this.hashedPassword.toString());
  };

  return User;
};
```

Finally, let's also update the seeders to reflect this new relationship between tweets and users. We'll also use the faker library to generate fake content, so run `npm install faker`, and then update the seeders to:

```
"use strict";
```

```javascript
const faker = require("faker");
const bcrypt = require("bcryptjs");

module.exports = {
  up: async (queryInterface, Sequelize) => {
    const users = await queryInterface.bulkInsert(
      "Users",
      [
        {
          username: faker.internet.userName(),
          email: faker.internet.email(),
          hashedPassword:
bcrypt.hashSync(faker.internet.password()),
          createdAt: new Date(),
          updatedAt: new Date(),
        },
        {
          username: faker.internet.userName(),
          email: faker.internet.email(),
          hashedPassword:
bcrypt.hashSync(faker.internet.password()),
          createdAt: new Date(),
          updatedAt: new Date(),
        },
      ],
      { returning: true }
    );

    return queryInterface.bulkInsert(
      "Tweets",
      [
        {
          message: faker.company.catchPhrase(),
          createdAt: new Date(),
          updatedAt: new Date(),
          userId: users[0].id,
        },
        {
          message: faker.company.catchPhrase(),
          createdAt: new Date(),
          updatedAt: new Date(),
          userId: users[0].id,
        },
        {
          message: faker.company.catchPhrase(),
          createdAt: new Date(),
          updatedAt: new Date(),
          userId: users[1].id,
        },
      ],
      {}
    );
  },

  down: async (queryInterface, Sequelize) => {
    await queryInterface.bulkDelete("Tweets", null, {});
    return queryInterface.bulkDelete("Users", null, {});
  },
};
```

Now that you've updated the migrations, models, and seeders to reflect the tweets *belongsTo* users and user *hasMany* tweets relationships, let's run all of the migrations and seed scripts:

```
npx dotenv sequelize db:migrate
npx dotenv sequelize db:seed:all
```

Check Postbird to verify all the data has been properly created. Go to `routes/tweets.js` and update the post route so that when a tweet is created, the `id` of the `req.user` is stored in the `userId` column of the tweet:

```
const tweet = await Tweet.create({ message, userId: req.user.id
});
```

Finally, update the GET / tweets route so that it also includes the author of the tweet. You previously learned about eager loading associations. When designing APIs, it's critical that you don't return any more info than is necessary.

So for example, in this case, when you eager load the user, if you didn't filter anything out, then it would include extra info like user email address or the hashedPassword, when really all the client needs to know from this route is the username of the author of the specific tweet and perhaps the user id of the author.

Fortunately, we can use sequelize `attributes` to filter out the fields that get returned from the sequelize query:

```
const tweets = await Tweet.findAll({
  include: [{ model: User, as: "user", attributes: ["username"]
}],
  order: [["createdAt", "DESC"]],
  attributes: ["message"],
});
```

Let's now go back to the client to set up a form for an authenticated user to create tweets!

Phase 6: Seeing tweets with authors and creating tweets from the client

First, let's update the `public/js/index.js` script to render the username on top of the message (hint: destructure the user from each tweet and then destructure the username `{ message, user: { username } }`).
Update the `tweetsHtml` to render a `username` as content within a `div.card-header` element. Make the `div.card-header` element a child of the `div.card` element. Feel free to make use fo the HTML snippet below:

```
<div class="card-header">
  ${username}
</div>
```

Next, let's set up a `create` view for the user to create a tweet. First add the route in your `express-apis-frontend/index.js` file:

```
app.get("/create", (req, res) => {
  res.render("create");
});
```

Then set up the associated `create.pug` template with the snippet below:

```
extends layout.pug

block content
  .errors-container
  form(class="create-form")
    .form-group
      label(for='message') Message
      input#message.form-control(type='message',
name="message", placeholder='What do you want to tweet?')
    button.btn.btn-primary(type='submit') Tweet

  script(src="js/create.js")
```

Before we implement the `public/js/create.js` script file, let's update the `layout.pug` template to now also render a nav bar at the top of the application to allow users to easily navigate around:

```
doctype html
html
  head
    title Twitter Lite
    link(rel='stylesheet'
href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/b
ootstrap.min.css' integrity='sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9If
jh' crossorigin='anonymous')
    link(rel='stylesheet' type='text/css' href='/style.css')
  body
    nav.navbar.navbar-expand-lg.navbar-light.bg-light
      a.navbar-brand(href='/') Twitter Lite
      .navbar-nav
        a.nav-item.nav-link(href='/create') Create Tweet
        a.nav-item.nav-link(href='/profile') My Tweets
    block content
```

Notice how there's a link to a `/profile` route, which we will implement in the next phase. Now create the `public/js/create.js` file. In this file, begin by adding an event listener to the element with a class name of `create-form`. Remember to use `e.preventDefault()` within the event listener to prevent the form from sending a GET request upon submission. Then parse the form data:

```
const form = document.querySelector(".create-form");

form.addEventListener("submit", async (e) => {
  e.preventDefault();
  const formData = new FormData(form);
  const message = formData.get("message");
  const body = { message };

  // TODO: Fetch tweets
  // TODO: Redirect users to login page upon a 401 error
  // TODO: Handle errors
});
```

Within a `try` block, use the Fetch API to make a POST request to `http://localhost:8080/tweets` with the necessary `body` (formatted as a JSON string) and headers (`Content-Type` and `Authorization`). Upon a successful fetch request, remember to redirect the user to the home page by using the `window.location.href` property.

Now handle the following response and errors caught. If you receive a response with a status of `401`, redirect users to log-in page. Throw the response (`res`) for any other response statuses. If you catch an error with a status greater than or equal to `400` and less than `600`, parse your caught error as JSON and find your `errorsContainer` (element with a class of `errors-container`). Declare an array of error elements by using the `errorsHtml` snippet below:

```
let errorsHtml = [
  `
  <div class="alert alert-danger">
      Something went wrong. Please try again.
  </div>
`
  ,
];
```

Now destructure your `errors` array from your error parsed as JSON and verify if `errors` is a valid, truthy array. If `errors` is a valid array, map through each `message` in the array to generate an array of stringified error elements:

```
const { errors } = errorJSON;
if (errors && Array.isArray(errors)) {
  errorsHtml = errors.map(
    (message) => `
    <div class="alert alert-danger">
        ${message}
    </div>
  `
  );
}
```

Now join the elements within your `errorsHtml` array and set the joined array as the `innerHTML` of the `errorsContainer`. Lastly, use the `alert()` method to notify the end user with a generic error message.

Phase 7: Set up the profile page

In the profile page, a user should be able to see their own tweets. Up to this point we haven't dealt with nested resources yet, but let's go ahead and build out a route where tweets are nested under a user.

In the users router, add an endpoint for `localhost:8080/users/:id/tweets`. You'll need to also import the Tweet model into this router. This route should await the database fetch of all tweets that belong to a specific user based on the `id` in the request `params`.
Make sure to import and your `requireAuth` middleware from the `../auth.js` file to keep your tweets *protected*. Also be sure to wrap the route handling function with your `asyncHandler` to be able to `await` the database fetch. Lastly, render the tweets in a JSON response.
Go to the client now and set up the profile page. Add the following route in the `express-apis-frontend/index.js` file to serve a `profile.pug` template:

```
app.get("/profile", (req, res) => {
  res.render("profile");
});
```

Then set up the `profile.pug` template:

```
extends layout.pug
block content
  h2 Your Tweets
  .tweets-container
  script(src="js/profile.js")
```

Finally, set up a script in the `public/js/profile.js` file to fetch all of the tweets that belong to the logged in user. Begin by adding a `DOMContentLoaded` event listener. Now remember that after a user logs in or signs up, the server gives the client the user's id, which is stored in localStorage. Use that id to `try` to make a fetch request to the correct route (hint: interpolate the user's id into the fetch path). Make sure to include an `Authorization` header that gets the `TWITTER_LITE_ACCESS_TOKEN` from localStorage.
Upon a response status of `401`, re-direct the user to the `/log-in` page by using the `window.location.href` property and `return` out of the event listener function.
Upon a successful fetch, parse the response as JSON and destructure the response `tweets`. Search for the `tweetsContainer` (element with a class name of `tweets-container`) and declare a `tweetsHtml` variable. Now map over the `tweets` from your fetch response to generate an array of stringified tweet elements and set the `tweetsHtml` variable:

```
const tweetsHtml = tweets.map(
  ({ message, id }) => `
  <div class="card" id="tweet-${id}">
    <div class="card-body">
      <p class="card-text">${message}</p>
    </div>
  </div>
`
);
```

Make sure to join your stringified tweet elements to set the `innerHTML` of the `tweetsContainer`:

```
tweetsContainer.innerHTML = tweetsHtml.join("");
```

Lastly, use `console.error()` to log any caught errors.

Nice work making it to this part of the project! In the next bonus phases, let's add a little more functionality, improve the UX of the app, and then also clean up the code!

Bonus Phase: Add ability to delete your own tweet

Let's add the ability to delete your own tweets, while not allowing anyone else to delete them!

Update your `profile.js` script so that after you fetch your own tweets, each message can show a delete button. Then, after the HTML has been added to the DOM, select all of the delete buttons to add a click handler to each button. The click handler's callback function should make a DELETE request to `http://localhost:8080/tweets/:id` in order to delete a tweet.
To add click handlers for each delete button, here's what you might want to write at the bottom of your "DOMContentLoaded" event listener in the `profile.js` file:

```
const deleteButtons = document.querySelectorAll(".delete-
button");
if (deleteButtons) {
  deleteButtons.forEach((button) => {
    button.addEventListener("click", handleDelete(button.id));
  });
}
```

In the code snippet above, there's a `handleDelete` callback function that's not shown. Go ahead and implement the logic for that function!

Next, let's add some logic so that only the author of a tweet can actually delete the tweet. Go to the endpoint for deleting tweets. How can you check whether or not the current user who's making this request is actually authorized to delete this tweet?

Once you've tested that a tweet can only be deleted by its author in Postman, move on to the next bonus phase!

Bonus Phase: Combine create and index views

Let's make it easier for users to create a tweet in the app. Right now, the whole process to create a tweet is a little cumbersome. Users have to navigate to a separate page for creating the tweet, and after it's created, they're then redirected back to another page to see the newly created.

Go ahead and move the create tweet form from `create.pug` to above the tweets container in `index.pug`. This will give the users the ability to create a tweet at the top of their timeline, which is an experience that's more in line with what the real Twitter does.

In this view, when users create a tweet, go ahead and just re-fetch all of the tweets again. Be sure to keep your code DRY here by extracting the logic to fetch all tweets into its own function.

Also, since you are no longer redirecting users to a different page after a tweet is created make sure you are clearing out the create tweet form inputs upon a successful tweet creation.

As a reminder, since you no longer need a create tweet page, go ahead and delete those template and JavaScript files, and then remove the navigation route from `layout.pug`.

When you are done with this phase, you should have an index page that allows users to create a tweet and then see that newly created tweet without ever requiring a page reload. This type of experience is more in line with what users expect from modern web apps. You'll dive deeper into how to build these types of modern frontends as you progress into the React curriculum soon!

Bonus Phase: ES Modules

Finally, let's clean up some of the JavaScript code in the client app! Right now a lot of the error handling logic is being duplicated between the different files. DRY up your code with JavaScript modules.

You'll need to declare `type="module"` in any script tags that load JavaScript files that use modules. Also, unlike the CommonJS module system (`require` and `module.exports`) that you've been using in your server-side JavaScript, you'll need to use the ES module system (`import` and `export`) in client-side JavaScript.

In the JavaScript modules documentation, there's a section that talks about how ES modules are not supported by all browsers. In the React curriculum, you'll learn more about how to effectively handle client-side modules using build tools like webpack.