- ⏱ 5 hours

Authenticate Me - Backend

In this multi-part project, you will learn how to put together an entire Express + React application with authentication.

Phase 0: Backend Set Up

First, you need to setup the backend of your application. This includes installing dependencies, setting up Sequelize, initializing your Express application, connecting Express security middlewares, and testing your server set up.

Backend and Frontend Separation

In this project, you will separate the backend Express code from the frontend React code.

Create a folder called `authenticate-me` and inside of it, create two folders called `backend` and `frontend`.

Your file structure should now look like this.

```
authenticate-me
├── backend
└── frontend
```

`.gitignore`

Create a `.gitignore` file at the root of the project with the following contents:

```
node_modules
.env
build
.DS_Store
```

Dependencies

In the `backend` folder, initialize the server's `package.json` by running `npm init -y`.
`npm install` the following packages as dependencies:

- `bcryptjs` - password hashing
- `cookie-parser` - parsing cookies from requests
- `cors` - CORS
- `csurf` - CSRF protection
- `dotenv` - load environment variables into Node.js from a `.env` file
- `express` - Express
- `express-async-handler` - handling `async` route handlers
- `express-validator` - validation of request bodies
- `faker` - random seeding library
- `helmet` - security middleware
- `jsonwebtoken` - JWT
- `morgan` - logging information about server requests/responses
- `per-env` - use environment variables for starting app differently
- `pg@">=8.4.1"` - PostgresQL greater or equal to version 8.4.1
- `sequelize@5` - Sequelize
- `sequelize-cli@5` - use `sequelize` in the command line

`npm install -D` the following packages as dev-dependencies:

- `dotenv-cli` - use `dotenv` in the command line
- `nodemon` - hot reload server `backend` files

Configuration

In the `backend` folder, create a `.env` file that will be used to define your environment variables.
Populate the `.env` file based on the example below:

```
PORT=5000
DB_USERNAME=auth_app
DB_PASSWORD=«auth_app user password»
DB_DATABASE=auth_db
DB_HOST=localhost
JWT_SECRET=«generate_strong_secret_here»
```

```
JWT_EXPIRES_IN=604800
```
Assign `PORT` to `5000`, add a user password and a strong JWT secret.
Recommendation to generate a strong secret: create a random string using `openssl` (a library that should be installed in your Ubuntu/MacOS shell already). Run `openssl rand -base64 10` to generate a random JWT secret.
Next, you will create a `js` configuration file that will read the environment variables loaded and export them.
Add a folder called `config` in your `backend` folder. Inside of the folder, create an `index.js` file with the following contents:
```
// backend/config/index.js
module.exports = {
  environment: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 5000,
  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,
  },
};
```

Each environment variable will be read and exported as a key from this file.

Sequelize Setup

You will setup Sequelize to look in the `backend/config/database.js` file for its database configurations. You will also setup the `backend/db` folder to contain all the files for models, seeders, and migrations.
To do this, create a `.sequelizerc` file in the `backend` folder with the following contents:
```
// backend/.sequelizerc
const path = require('path');

module.exports = {
  config: path.resolve('config', 'database.js'),
  'models-path': path.resolve('db', 'models'),
  'seeders-path': path.resolve('db', 'seeders'),
  'migrations-path': path.resolve('db', 'migrations'),
};
```
Initialize Sequelize to the `db` folder by running:
```
npx sequelize init
```

Replace the contents of the newly created `backend/config/database.js` file with the following:

```
// backend/config/database.js
const config = require('./index');

const db = config.db;
const username = db.username;
const password = db.password;
const database = db.database;
const host = db.host;

module.exports = {
  development: {
    username,
    password,
    database,
    host,
    dialect: 'postgres',
    seederStorage: 'sequelize',
  },
  production: {
    use_env_variable: 'DATABASE_URL',
    dialect: 'postgres',
    seederStorage: 'sequelize',
    dialectOptions: {
      ssl: {
        require: true,
        rejectUnauthorized: false,
      },
    },
  },
};
```

This will allow you to load the database configuration environment variables from the `.env` file into the `config/index.js`.

Notice how the `production` database configuration has different keys than the `development` configuration? When you deploy your application to production, your database will be read from a URL path instead of a username, password, and database name combination.

Next, create a user using the same credentials in the `.env` file with the ability to create databases.

```
psql -c "CREATE USER <username> PASSWORD '<password>' CREATEDB"
```

Finally, create the database using `sequelize-cli`.

```
npx dotenv sequelize db:create
```

Remember, any `sequelize db:` commands need to be prefixed with `dotenv` to load the database configuration environment variables from the `.env` file.

Express Setup

After you setup Sequelize, it's time to start working on getting your Express application set up.

`app.js`
Create a file called `app.js` in the `backend` folder. Here you will initialize your Express application.

At the top of the file, import the following packages:

```
const express = require('express');
const morgan = require('morgan');
const cors = require('cors');
const csurf = require('csurf');
const helmet = require('helmet');
const cookieParser = require('cookie-parser');
```

Create a variable called `isProduction` that will be `true` if the environment is in production or not by checking the `environment` key in the configuration file (`backend/config/index.js`).

```
const { environment } = require('./config');
const isProduction = environment === 'production';
```

Initialize the Express application:

```
const app = express();
```

Connect the `morgan` middleware for logging information about requests and responses:

```
app.use(morgan('dev'));
```

Add the `cookie-parser` middleware for parsing cookies and the `express.json` middleware for parsing JSON bodies of requests with `Content-Type` of `"application/json"`.

```
app.use(cookieParser());
app.use(express.json());
```

Add several security middlewares. First, only allow CORS (Cross-Origin Resource Sharing) in development using the `cors` middleware because React frontend will be served from a different server than the Express server. CORS isn't needed in production since all of our React and Express resources will come from the same origin. Second, enable better overall security with the `helmet` middleware (for more on what `helmet` is doing, see helmet on the `npm` registry). Disable the Content Security Policy feature in `helmet`. React is generally safe at mitigating Cross-Site Scripting. Make sure to do proper research on how to protect your users against Cross-Site Scripting attacks in React when deploying a large production application. Third, add the `csurf` middleware and configure it to use cookies.

```
// Security Middleware
if (!isProduction) {
  // enable cors only in development
  app.use(cors());
}
```

```
// helmet helps set a variety of headers to better secure your
app
app.use(helmet({
  contentSecurityPolicy: false
}));

// Set the _csrf token and create req.csrfToken method
app.use(
  csurf({
    cookie: {
      secure: isProduction,
      sameSite: isProduction && "Lax",
      httpOnly: true,
    },
  })
);
```
The `csurf` middleware will add a `_csrf` cookie that is HTTP-only (can't be read by JavaScript) to any server response. It also adds a method on all requests (`req.csrfToken`) that will be set to another cookie (`XSRF-TOKEN`) later on. These two cookies work together to provide CSRF (Cross-Site Request Forgery) protection for your application. The `XSRF-TOKEN` cookie value needs to be sent in the header of any request with all HTTP verbs besides `GET`. This header will be used to validate the `_csrf` cookie to confirm that the request comes from your site and not an unauthorized site.

Now that you set up all the pre-request middleware, it's time to set up the routes for your Express application.

Routes

Create a folder called `routes` in your `backend` folder. All your routes will live in this folder.
Create an `index.js` file in the `routes` folder. In this file, create an Express router, create a test route, and export the router at the bottom of the file.
```
// backend/routes/index.js
const express = require('express');
const router = express.Router();

router.get('/hello/world', function(req, res) {
  res.cookie('XSRF-TOKEN', req.csrfToken());
  res.send('Hello World!');
});

module.exports = router;
```
In this test route, you are setting a cookie on the response with the name of `XSRF-TOKEN` to the value of the `req.csrfToken` method's return. Then, you are sending the text, `Hello World!` as the response's body.

Add the routes to the Express application by importing with the other imports in `backend/app.js` and connecting the exported router to `app` after all the middlewares.

```
// backend/app.js
const routes = require('./routes');

// ...

app.use(routes); // Connect all the routes
```

Finally, at the bottom of the `app.js` file, export `app`.

```
// backend/app.js
// ...

module.exports = app;
```

After setting up the Express application, it's time to create the server.

`bin/www`

Create a folder in `backend` called `bin`. Inside of it, add a file called `www` with the following contents:

```
#!/usr/bin/env node
// backend/bin/www
const { port } = require('../config');

const app = require('../app');
const db = require('../db/models');

// Check the database connection before starting the app
db.sequelize
  .authenticate()
  .then(() => {
    console.log('Database connection success! Sequelize is
ready to use...');

    // Start listening for connections
    app.listen(port, () => console.log(`Listening on port
${port}...`));
  })
  .catch((err) => {
    console.log('Database connection failure.');
    console.error(err);
  });
```

Here, you will be starting your Express application to listen for server requests only after authenticating your database connection.

Test the Server

At this point, your database, Express application, and server are all set up and ready to be tested!

In your `package.json`, add the following scripts:

```
  "scripts": {
    "sequelize": "sequelize",
    "sequelize-cli": "sequelize-cli",
    "start": "per-env",
    "start:development": "nodemon -r dotenv/config ./bin/www",
    "start:production": "node ./bin/www"
  }
```

`npm start` will run the `/bin/www` in `nodemon` when started in the development environment with the environment variables in the `.env` file loaded, or in `node` when started in production.

Now, it's time to finally test your entire set up!

Run `npm start` in the `backend` folder to start your server on the port defined in the `.env` file, which should be `5000`.
Navigate to the test route at http://localhost:5000/hello/world. There, you should see the text `Hello World!`. Take a look at your cookies. Delete all the cookies to make sure there are no lingering cookies from other projects, then refresh the page. You should still see the text `Hello World!` on the page as well as two cookies, one called `_csrf` and the other called `XSRF-TOKEN` in your dev tools.
If you don't see this, then check your backend server logs in your terminal where you ran `npm start`. Then check your routes.
If there is a database connection error, make sure you set up the correct username and password defined in the `.env` file.

When you're finished testing, commit! Now is a good time to commit because you have working code.

Phase 1: API Routes

The main purpose of this Express application is to be a REST API server. All the API routes will be served at URL's starting with `/api/`.
Get started by nesting an `api` folder in your `routes` folder. Add an `index.js` file in the `api` folder with the following contents:

```
// backend/routes/api/index.js
const router = require('express').Router();

module.exports = router;
```

Import this file into the `routes/index.js` file and connect it to the `router` there.

```
// backend/routes/index.js
// ...
const apiRouter = require('./api');

router.use('/api', apiRouter);
// ...
```

All the URLs of the routes in the `api` router will be prefixed with `/api`.

Test the API Router

Make sure to test this setup by creating the following test route in the `api` router:

```
// backend/routes/api/index.js
// ...

router.post('/test', function(req, res) {
  res.json({ requestBody: req.body });
});

// ...
```

A router is created and an API test route is added to the router. The API test route is accepting requests with the URL path of `/api/test` with the HTTP verb of `POST`. It sends a JSON response containing whatever is in the body of the request.
Test this route by navigating to the other test route, http://localhost:5000/hello/world, and creating a `fetch` request in the browser's development tools console. Make a request to `/api/test` with the `POST` method, a body of `{ hello: 'world' }`, a `"Content-Type"` header, and an `XSRF-TOKEN` header with the value of the `XSRF-TOKEN` cookie located in your dev tools.

Example fetch request:

```
fetch('/api/test', {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "XSRF-TOKEN": `<value of XSRF-TOKEN cookie>`
  },
  body: JSON.stringify({ hello: 'world' })
}).then(res => res.json()).then(data => console.log(data));
```

Replace the `<value of XSRF-TOKEN cookie>` with the value of the `XSRF-TOKEN` cookie. If you don't have the `XSRF-TOKEN` cookie anymore, access the http://localhost:5000/hello/world route to add the cookie back.

After the response returns to the browser, parse the JSON response body and print it out.

If you get an error, check your backend server logs in your terminal where you ran `npm start`. Also, check your `fetch` request syntax and your API router setup.

After you finish testing, commit your code!

Phase 2: Error Handling

The next step is to set up your server error handlers.

Connect the following error handling middlewares after your route connections in `app.js` (i.e. after `app.use(routes)`). Here is a refresher on how to create an Express error-handling middleware.

Resource Not Found Error-Handler

The first error handler is actually just a regular middleware. It will catch any requests that don't match any of the routes defined and create a server error with a status code of `404`.

```
// backend/app.js
// ...
// Catch unhandled requests and forward to error handler.
app.use((_req, _res, next) => {
  const err = new Error("The requested resource couldn't be
found.");
  err.title = "Resource Not Found";
  err.errors = ["The requested resource couldn't be found."];
  err.status = 404;
  next(err);
});
```

If this resource not found middleware is called, an error will be created with the message `"The requested resource couldn't be found."` and a status code of `404`. Afterwards, `next` will be invoked with the error. Remember, `next` invoked with nothing means that error handlers defined **after** this middleware **will not** be invoked. However, `next` invoked with an error means that error handlers defined **after** this middleware **will** be invoked.

Sequelize Error-Handler

The second error handler is for catching Sequelize errors and formatting them before sending the error response.

```
// backend/app.js
// ...
const { ValidationError } = require('sequelize');

// ...

// Process sequelize errors
app.use((err, _req, _res, next) => {
  // check if error is a Sequelize error:
  if (err instanceof ValidationError) {
    err.errors = err.errors.map((e) => e.message);
    err.title = 'Validation error';
  }
  next(err);
});
```

If the error that caused this error-handler to be called is an instance of `ValidationError` from the `sequelize` package, then the error was created from a Sequelize database validation error and the additional keys of `title` string and `errors` array will be added to the error and passed into the next error handling middleware.

Error Formatter Error-Handler

The last error handler is for formatting all the errors before returning a JSON response. It will include the error message, the errors array, and the error stack trace (if the environment is in development) with the status code of the error message.

```
// backend/app.js
// ...
// Error formatter
app.use((err, _req, res, _next) => {
  res.status(err.status || 500);
  console.error(err);
  res.json({
    title: err.title || 'Server Error',
    message: err.message,
    errors: err.errors,
    stack: isProduction ? null : err.stack,
  });
});
```

This should be the last middleware in the `app.js` file of your Express application.

Testing the Error Handlers

You can't really test the Sequelize error handler now because you have no Sequelize models to test it with, but you can test the *Resource Not Found* error handler and the Error Formatter error handler.
To do this, try to access a route that hasn't been defined in your `routes` folder yet, like http://localhost:5000/not-found.
If you see the json below, you have successfully set up your *Resource Not Found* and Error Formatter error handlers!

```
{
  "title": "Resource Not Found",
  "message": "The requested resource couldn't be found.",
  "errors": [
    "The requested resource couldn't be found."
  ],
  "stack": "...stack trace..."
}
```

If you don't see the json above, check your backend server logs in your terminal where you ran `npm start`.
Make sure your other test route at http://localhost:5000/hello/world is still working. If it is not working, make sure you are defining your error handlers **after** your route connections in `app.js` (i.e. after `app.use(routes)`).

You will test the Sequelize error handler later when you populate the database with a table.

Before moving onto the next task, commit your error handling code!

Phase 3: User Authentication

Now that you have finished setting up both Sequelize and the Express application, you are ready to start implementing user authentication in the backend.

With Sequelize, you will create a `Users` table that will have the following schema:

column name	**data type**	**constraints**
`id`	integer	not null, primary key
`username`	string	not null, indexed, unique, max 30 characters
`email`	string	not null, indexed, unique, max 256 characters
`hashedPassword`	binary string	not null
`createdAt`	datetime	not null, default value of now()

column name	data type	constraints
updatedAt	datetime	not null, default value of now()

Users Table Migration

The first thing to do is to generate a migration and model file. Navigate into the `backend` folder in your terminal and run the following command:

```
npx sequelize model:generate --name User --attributes
username:string,email:string,hashedPassword:string
```

This will create a file in your `backend/db/migrations` folder and a file called `user.js` in your `backend/db/models` folder.

In the migration file, apply the constraints in the schema. If completed correctly, your migration file should look something like this:

```
'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Users', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      username: {
        type: Sequelize.STRING(30),
        allowNull: false,
        unique: true,
      },
      email: {
        type: Sequelize.STRING(256),
        allowNull: false,
        unique: true,
      },
      hashedPassword: {
        type: Sequelize.STRING.BINARY,
        allowNull: false,
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE,
        defaultValue: Sequelize.fn('now'),
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE,
        defaultValue: Sequelize.fn('now'),
      }
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Users');
  }
};
```
Migrate the `Users` table by running the following command:
```
npx dotenv sequelize db:migrate
```

If there is an error when migrating, check your migration file and make changes.

If there is no error when migrating, but you want to change the migration file afterwards, undo the migration first, change the file, then migrate again.

Command to undo the migration:

```
npx dotenv sequelize db:migrate:undo
```
You can check out the `Users` table schema created in your PostgresQL database by running the following command in your terminal:
```
psql <database name> -c '\d "Users"'
```

User Model

After you migrate the `Users` table with the database-level constraints, you need to add Sequelize model-level constraints. In your `User` model file, `backend/db/models/user.js`, add the following constraints:

column name	**data type**	**constraints**
`username`	string	not null, unique, min 4 characters, max 30 characters, isNotEmail
`email`	string	not null, unique, min 3 characters, max 256 characters, isEmail
`hashedPassword`	binary string	not null, min and max 60 characters

See the Sequelize docs on [model-level validations]() for a reminder on how to apply these constraints.
Your `user.js` file should look like this with the applied constraints:

```
'use strict';
const { Validator } = require('sequelize');

module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define('User', {
    username: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        len: [4, 30],
        isNotEmail(value) {
          if (Validator.isEmail(value)) {
            throw new Error('Cannot be an email.');
          }
        },
      },
    },
    email: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        len: [3, 256]
      },
    },
    hashedPassword: {
      type: DataTypes.STRING.BINARY,
      allowNull: false,
      validate: {
        len: [60, 60]
      },
    },
  }, {});
  User.associate = function(models) {
    // associations can be defined here
  };
  return User;
};
```

A custom validator needs to be created for the `isNotEmail` constraint. See here for a refresher on custom Sequelize validators. You can use the imported `isEmail` validation from the `sequelize` package's `Validator` to check if the `username` is an email. If it is, throw an error with a message.

Users Seeds

Generate a user's seeder file for the demo user with the following command:

```
npx sequelize seed:generate --name demo-user
```

In the seeder file, create a demo user with `email`, `username`, and `hashedPassword` fields. For the `down` function, delete the user with the `username` or `email` of the demo user. If you'd like, you can also add other users and populate the fields with random fake data from the `faker` package. To generate the `hashedPassword` you should use the `bcryptjs` package's `hashSync` method.

Your seeder file should look something like this:

```
'use strict';
const faker = require('faker');
const bcrypt = require('bcryptjs');

module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.bulkInsert('Users', [
      {
        email: 'demo@user.io',
        username: 'Demo-lition',
        hashedPassword: bcrypt.hashSync('password'),
      },
      {
        email: faker.internet.email(),
        username: 'FakeUser1',
        hashedPassword:
bcrypt.hashSync(faker.internet.password()),
      },
      {
        email: faker.internet.email(),
        username: 'FakeUser2',
        hashedPassword:
bcrypt.hashSync(faker.internet.password()),
      },
    ], {});
  },

  down: (queryInterface, Sequelize) => {
    const Op = Sequelize.Op;
    return queryInterface.bulkDelete('Users', {
      username: { [Op.in]: ['Demo-lition', 'FakeUser1',
'FakeUser2'] }
    }, {});
  }
};
```

Notice how you do not need to add the `createdAt` and `updatedAt` fields for the users. This is a result of the default value that you defined in the Sequelize migration file for those fields.
Make sure to import `faker` and `bcryptjs` at the top of the file.

After you finish creating your demo user seed file, migrate the seed file by running the following command:

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

If there is an error with seeding, check your seed file and make changes.

If there is no error in seeding but you want to change the seed file, remember to undo the seed first, change the file, then seed again.

Command to undo the migration for the most recent seed file:

```
npx dotenv sequelize db:seed:undo
```

Command to undo the migrations for all the seed files:

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

Check your database to see if the users have been successfully created by running:

```
psql <database name> -c 'SELECT * FROM "Users"'
```

Model Scopes - Protecting your Users' Information

To ensure that a user's information like their `hashedPassword` doesn't get sent to the frontend, you should define `User` model scopes. Check out the official documentation on model scoping to look up how to define a model scope to prevent certain fields from being sent in a query.
For the default query when searching for `Users`, the `hashedPassword` and `updatedAt` and, depending on your application, `email` and `createdAt` fields should not be returned. To do this, set a `defaultScope` on the `User` model to exclude the desired fields from the default query. For example, when you run `User.findAll()` all fields besides `hashedPasswords`, `updatedAt`, `email`, and `createdAt` will be populated in the return of that query.
Next, define a `User` model scope for `currentUser` that will exclude only the `hashedPassword` field. Finally, define another scope for including all the fields, which should only be used when checking the login credentials of a user. These scopes need to be explicitly used when querying. For example, `User.scope('currentUser').findByPk(id)` will find a `User` by the specified `id` and return only the `User` fields that the `currentUser` model scope allows.
Your `user.js` model file should now look like this:

```
'use strict';
const { Validator } = require('sequelize');

module.exports = (sequelize, DataTypes) => {
  const User = sequelize.define('User', {
    username: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        len: [3, 30],
        isNotEmail(value) {
          if (Validator.isEmail(value)) {
            throw new Error('Cannot be an email.');
          }
        },
      },
    },
    email: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        len: [3, 256]
      },
    },
    hashedPassword: {
      type: DataTypes.STRING.BINARY,
      allowNull: false,
      validate: {
        len: [60, 60]
      },
    },
  },
  {
    defaultScope: {
      attributes: {
        exclude: ['hashedPassword', 'email', 'createdAt',
'updatedAt'],
      },
    },
    scopes: {
      currentUser: {
        attributes: { exclude: ['hashedPassword'] },
      },
      loginUser: {
        attributes: {},
      },
    },
  });
  User.associate = function(models) {
    // associations can be defined here
  };
  return User;
};
```

These scopes help protect sensitive user information that should not be exposed to other users. You will be using these scopes in the later sections.

Authentication Flow

The backend login flow in this project will be based off of the following plan:

1. The API login route will be hit with a request body holding a valid credential (either username or email) and password combination
2. The API login handler will look for a `User` with the input credential in either the `username` or `email` columns
3. Then the `hashedPassword` for that found `User` will be compared with the input `password` for a match
4. If there is a match, the API login route should send back a JWT in an HTTP-only cookie and a response body. The JWT and the body will hold the user's `id`, `username` and `email`

The backend signup flow in this project will be based off of the following plan:

1. The API signup route will be hit with a request body holding a `username`, `email`, and `password`
2. The API signup handler will create a `User` with the `username`, an `email`, and a `hashedPassword` created from the input `password`
3. If the creation is successful, the API signup route should send back a JWT in an HTTP-only cookie and a response body. The JWT and the body will hold the user's `id`, `username` and `email`

The backend logout flow will be based off of the following plan:

1. The API logout route will be hit with a request
2. The API logout handler will remove the JWT cookie set by the login or signup API routes and return a JSON success message

User Model Methods

After creating the model scopes, you should create methods that the API routes for authentication will use to interact with the `Users` table. The planned methods are based off of the authentication flow plans outlined above.

Define an instance method, User.prototype.toSafeObject, in the user.js model file. This method will return an object with only the User instance information that is safe to save to a JWT.

```
User.prototype.toSafeObject = function() { // remember, this
cannot be an arrow function
  const { id, username, email } = this; // context will be the
User instance
  return { id, username, email };
};
```

Define an instance method, User.prototype.validatePassword in the user.js model file. It will accept a password string and return true if there is a match with the User instance's hashedPassword, otherwise it will return false.

```
User.prototype.validatePassword = function (password) {
 return bcrypt.compareSync(password,
this.hashedPassword.toString());
};
```

You are using the bcryptjs package to compare the password and the hashedPassword, so make sure to import the package at the top of the user.js file.

```
const bcrypt = require('bcryptjs');
```

Define a static method, User.getCurrentUserById in the user.js model file that will accept an id. It should use the currentUser scope to return a User with that id.

```
User.getCurrentUserById = async function (id) {
 return await User.scope('currentUser').findByPk(id);
};
```

Define a static method, User.login in the user.js model file. It will accept an object with a credential and password key. The method should search for one User with the specified credential, (a username or an email). If a user is found, then validate the password by passing it into the instance's .validatePassword method. If the password is valid, then return the user by using the currentUser scope.

```
User.login = async function ({ credential, password }) {
  const { Op } = require('sequelize');
  const user = await User.scope('loginUser').findOne({
    where: {
      [Op.or]: {
        username: credential,
        email: credential,
      },
    },
  });
  if (user && user.validatePassword(password)) {
    return await User.scope('currentUser').findByPk(user.id);
  }
};
```

Define a static method, User.signup in the user.js model file that will accept an object with a username, email and password key. Hash the password using bcryptjs package's hashSync method. Create a User with the username, email, and hashedPassword. Return the created user using the currentUser scope.

```
User.signup = async function ({ username, email, password }) {
  const hashedPassword = bcrypt.hashSync(password);
  const user = await User.create({
    username,
    email,
    hashedPassword,
  });
  return await User.scope('currentUser').findByPk(user.id);
};
```

User Auth Middlewares

There are three functions in this section that will aid you in authentication.

Create a folder called `utils` in your `backend` folder. Inside that folder, add a file named `auth.js` to store the auth helper functions.

At the top of the file, add the following imports:

```
// backend/utils/auth.js
const jwt = require('jsonwebtoken');
const { jwtConfig } = require('../config');
const { User } = require('../db/models');

const { secret, expiresIn } = jwtConfig;
```

`setTokenCookie`

This first function is setting the JWT cookie after a user is logged in or signed up. It takes in the response and the session user and generates a JWT using the imported secret. It is set to expire in however many seconds you set on the `JWT_EXPIRES_IN` key in the `.env` file. The payload of the JWT will be the return of the instance method `.toSafeObject` that you added previously to the `User` model. After the JWT is created, it's set to an HTTP-only cookie on the response as a `token` cookie.

```
// backend/utils/auth.js
// ...

// Sends a JWT Cookie
const setTokenCookie = (res, user) => {
  // Create the token.
  const token = jwt.sign(
    { data: user.toSafeObject() },
    secret,
    { expiresIn: parseInt(expiresIn) }, // 604,800 seconds = 1
week
  );

  const isProduction = process.env.NODE_ENV === "production";
  // Set the token cookie
  res.cookie('token', token, {
    maxAge: expiresIn * 1000, // maxAge in milliseconds
    httpOnly: true,
    secure: isProduction,
    sameSite: isProduction && "Lax",
  });

  return token;
};
```

This function will be used in the login and signup routes later.

`restoreUser`

Certain authenticated routes will require the identity of the current session user. You will create and utilize a middleware function called restoreUser that will restore the session user based on the contents of the JWT cookie. Create a middleware function that will verify and parse the JWT's payload and search the database for a `User` with the id in the payload (this query should use the `currentUser` scope since the `hashedPassword` is not needed for this operation. If there is a `User` found, then save the user to a key of `user` onto the request. If there is an error verifying the JWT or a `User` cannot be found with the `id`, then clear the `token` cookie from the response.

```
// backend/utils/auth.js
// ...

const restoreUser = (req, res, next) => {
  // token parsed from cookies
  const { token } = req.cookies;

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

    try {
      const { id } = jwtPayload.data;
      req.user = await User.scope('currentUser').findByPk(id);
    } catch (e) {
      res.clearCookie('token');
      return next();
    }

    if (!req.user) res.clearCookie('token');

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

This will be added as a pre-middleware for route handlers and for the following authentication middleware.

`requireAuth`

The last authentication middleware to add is for requiring a session user to be authenticated before accessing a route.

Create an Express middleware called `requireAuth`. Define this middleware as an array with the `restoreUser` middleware function you just created as the first element in the array. This will ensure that if a valid JWT cookie exists, the session user will be loaded into the `req.user` attribute. The second middleware will check `req.user` and will go to the next middleware if there is a session user present there. If there is no session user, then an error will be created and passed along to the error-handling middlewares.

```
// backend/utils/auth.js
// ...

// If there is no current user, return an error
const requireAuth = [
  restoreUser,
  function (req, res, next) {
    if (req.user) return next();

    const err = new Error('Unauthorized');
    err.title = 'Unauthorized';
    err.errors = ['Unauthorized'];
    err.status = 401;
    return next(err);
  },
];
```

Both restoreUser and requireAuth will be applied as a pre-middleware to route handlers where needed.

Finally, export all the functions at the bottom of the file.

```
// backend/utils/auth.js
// ...

module.exports = { setTokenCookie, restoreUser, requireAuth };
```

Test User Auth Middlewares

Let's do some testing! It's always good to test your code anytime you have an opportunity to do it. Testing at the very end is not a good idea because it will be hard to pinpoint the location of the error in your code.

Add a test route in your `backend/routes/api/index.js` file that will test the `setTokenCookie` function by getting the demo user and calling `setTokenCookie`.

```
// backend/routes/api/index.js
// ...

// GET /api/set-token-cookie
const asyncHandler = require('express-async-handler');
const { setTokenCookie } = require('../../utils/auth.js');
const { User } = require('../../db/models');
router.get('/set-token-cookie', asyncHandler(async (req, res)
=> {
  const user = await User.findOne({
      where: {
        username: 'Demo-lition'
      },
    })
  setTokenCookie(res, user);
  return res.json({ user });
}));

// ...
```

Go to http://localhost:5000/api/set-token-cookie and see if there is a `token` cookie set in your browser's dev tools. If there isn't, then check your backend server logs in your terminal where you ran `npm start`. Also, check the syntax of your `setTokenCookie` function as well as the test route.
Next, add a test route in your `backend/routes/api/index.js` file that will test the `restoreUser` middleware by connecting the middleware and checking whether or not the `req.user` key has been populated by the middleware properly.

```
// backend/routes/api/index.js
// ...

// GET /api/restore-user
const { restoreUser } = require('../../utils/auth.js');
router.get(
  '/restore-user',
  restoreUser,
  (req, res) => {
    return res.json(req.user);
  }
);

// ...
```

Go to http://localhost:5000/api/restore-user and see if the response has the demo user information returned as JSON. Then, remove the `token` cookie manually in your browser's dev tools and refresh. The JSON response should be empty.
If this isn't the behavior, then check your backend server logs in your terminal where you ran `npm start` as well as the syntax of your `restoreUser` middleware and test route.
To set the `token` cookie back, just go to the `GET /api/set-token-cookie` route again, http://localhost:5000/api/set-token-cookie.

Lastly, test your `requireAuth` middleware by adding a test route in your `backend/routes/api/index.js` file. If there is no session user, the route will return an error. Otherwise it will return the session user's information.

```
// backend/routes/api/index.js
// ...

// GET /api/require-auth
const { requireAuth } = require('../../utils/auth.js');
router.get(
  '/require-auth',
  requireAuth,
  (req, res) => {
    return res.json(req.user);
  }
);

// ...
```

Set the `token` cookie back by accessing the `GET /api/set-token-cookie` route again, http://localhost:5000/api/set-token-cookie.
Go to http://localhost:5000/api/require-auth and see if the response has the demo user's information returned as JSON. Then, remove the `token` cookie manually in your browser's dev tools and refresh. The JSON response should now be an `"Unauthorized"` error.
If this isn't the behavior, then check your backend server logs in your terminal where you ran `npm start` as well as the syntax of your `requireAuth` middleware and test route.
To set the `token` cookie back, just go to the `GET /api/set-token-cookie` route again, http://localhost:5000/api/set-token-cookie.
Once you are satisfied with the test results, you can remove all code for the testing the user auth middlewares routes.

Phase 4: User Auth Routes

It's finally time to create the authentication API routes!

In this section, you will add the following routes to your Express application:

- Login: `POST /api/session`
- Logout: `DELETE /api/session`
- Signup: `POST /api/users`
- Get session user: `GET /api/session`

First, create a file called `session.js` in the `backend/routes/api` folder. This file will hold the resources for the route paths beginning with `/api/session`. Create and export an Express router from this file.

```
// backend/routes/api/session.js
```

```
const express = require('express')
const router = express.Router();

module.exports = router;
```

Next create a file called `users.js` in the `backend/routes/api` folder. This file will hold the resources for the route paths beginning with `/api/users`. Create and export an Express router from this file.

```
// backend/routes/api/users.js
const express = require('express')
const router = express.Router();

module.exports = router;
```

Connect all the routes exported from these two files in the `index.js` file nested in the `backend/routes/api` folder.
Your `backend/routes/api/index.js` file should now look like this:

```
// backend/routes/api/index.js
const router = require('express').Router();
const sessionRouter = require('./session.js');
const usersRouter = require('./users.js');

router.use('/session', sessionRouter);

router.use('/users', usersRouter);

router.post('/test', (req, res) => {
  res.json({ requestBody: req.body });
});

module.exports = router;
```

User Login API Route

In the `backend/routes/api/session.js` file, import the following code at the top of the file and create an Express router:

```
// backend/routes/api/session.js
const express = require('express');
const asyncHandler = require('express-async-handler');

const { setTokenCookie, restoreUser } =
require('../../utils/auth');
const { User } = require('../../db/models');

const router = express.Router();
```

The `asyncHandler` function from `express-async-handler` will wrap asynchronous route handlers and custom middlewares.

Next, add the `POST /api/session` route to the router using an asynchronous route handler. In the route handler, call the `login` static method from the `User` model. If there is a user returned from the `login` static method, then call `setTokenCookie` and return a JSON response with the user information. If there is no user returned from the `login` static method, then create a `"Login failed"` error and invoke the next error-handling middleware with it.

```
// backend/routes/api/session.js
// ...

// Log in
router.post(
  '/',
  asyncHandler(async (req, res, next) => {
    const { credential, password } = req.body;

    const user = await User.login({ credential, password });

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

    await setTokenCookie(res, user);

    return res.json({
      user,
    });
  }),
);
```

Make sure to export the `router` at the bottom of the file.

```
// backend/routes/api/session.js
// ...

module.exports = router;
```

Test the Login Route

Test the login route by navigating to the http://localhost:5000/hello/world test route and making a fetch request from the browser's dev tools console. Remember, you need to pass in the value of the `XSRF-TOKEN` cookie as a header in the fetch request because the login route has a `POST` HTTP verb.

If at any point you don't see the expected behavior while testing, then check your backend server logs in your terminal where you ran `npm start`. Also, check the syntax in the `session.js` as well as the `login` method in the `user.js` model file.

Try to login the demo user with the username first.

```
fetch('/api/session', {
  method: 'POST',
  headers: {
    "Content-Type": "application/json",
    "XSRF-TOKEN": `<value of XSRF-TOKEN cookie>`
  },
  body: JSON.stringify({ credential: 'Demo-lition', password:
'password' })
}).then(res => res.json()).then(data => console.log(data));
```

Remember to replace the `<value of XSRF-TOKEN cookie>` with the value of the `XSRF-TOKEN` cookie found in your browser's dev tools. If you don't have the `XSRF-TOKEN` cookie anymore, access the http://localhost:5000/hello/world route to add the cookie back.

Then try to login the demo user with the email next.

```
fetch('/api/session', {
  method: 'POST',
  headers: {
    "Content-Type": "application/json",
    "XSRF-TOKEN": `<value of XSRF-TOKEN cookie>`
  },
  body: JSON.stringify({ credential: 'demo@user.io', password:
'password' })
}).then(res => res.json()).then(data => console.log(data));
```

Now test an invalid user `credential` and `password` combination.

```
fetch('/api/session', {
  method: 'POST',
  headers: {
    "Content-Type": "application/json",
    "XSRF-TOKEN": `<value of XSRF-TOKEN cookie>`
  },
  body: JSON.stringify({ credential: 'Demo-lition', password:
'Hello World!' })
}).then(res => res.json()).then(data => console.log(data));
```

You should get a `Login failed` error back with an invalid `password` for the user with that `credential`.

Commit your code for the login route once you are done testing!

User Logout API Route

The `DELETE /api/session` logout route will remove the `token` cookie from the response and return a JSON success message.

```
// backend/routes/api/session.js
// ...

// Log out
router.delete(
  '/',
  (_req, res) => {
    res.clearCookie('token');
    return res.json({ message: 'success' });
  }
);

// ...
```

Notice how `asyncHandler` wasn't used to wrap the route handler. This is because the route handler is not `async`.

Test the Logout Route

Start by navigating to the http://localhost:5000/hello/world test route and making a fetch request from the browser's dev tools console to test the logout route. Check that you are logged in by confirming that a `token` cookie is in your list of cookies in the browser's dev tools. Remember, you need to pass in the value of the `XSRF-TOKEN` cookie as a header in the fetch request because the logout route has a `DELETE` HTTP verb.

Try to logout the session user.

```
fetch('/api/session', {
  method: 'DELETE',
  headers: {
    "Content-Type": "application/json",
    "XSRF-TOKEN": `<value of XSRF-TOKEN cookie>`
  }
}).then(res => res.json()).then(data => console.log(data));
```

You should see the `token` cookie disappear from the list of cookies in your browser's dev tools. If you don't have the `XSRF-TOKEN` cookie anymore, access the http://localhost:5000/hello/world route to add the cookie back.
If you don't see this expected behavior while testing, then check your backend server logs in your terminal where you ran `npm start` as well as the syntax in the `session.js` route file.

Commit your code for the logout route once you are done testing!

User Signup API Route

In the `backend/routes/api/users.js` file, import the following code at the top of the file and create an Express router:

```
const express = require('express');
const asyncHandler = require('express-async-handler');

const { setTokenCookie, requireAuth } =
require('../../utils/auth');
const { User } = require('../../db/models');

const router = express.Router();
```

Next, add the `POST /api/users` route to the router using the asyncHandler function and an asynchronous route handler. In the route handler, call the `signup` static method on the `User` model. If the user is successfully created, then call `setTokenCookie` and return a JSON response with the user information. If the creation of the user is unsuccessful, then a Sequelize Validation error will be passed onto the next error-handling middleware.

```
// backend/routes/api/users.js
// ...

// Sign up
router.post(
  '/',
  asyncHandler(async (req, res) => {
    const { email, password, username } = req.body;
    const user = await User.signup({ email, username, password });

    await setTokenCookie(res, user);

    return res.json({
      user,
    });
  }),
);
```

Make sure to export the `router` at the bottom of the file.

```
// backend/routes/api/users.js
// ...

module.exports = router;
```

Test the Signup Route

Test the signup route by navigating to the http://localhost:5000/hello/world test route and making a fetch request from the browser's dev tools console. Remember, you need

to pass in the value of the `XSRF-TOKEN` cookie as a header in the fetch request because the login route has a `POST` HTTP verb.
If at any point you don't see the expected behavior while testing, check your backend server logs in your terminal where you ran `npm start`. Also, check the syntax in the `users.js` route file as well as the `signup` method in the `user.js` model file.

Try to signup a new valid user.

```
fetch('/api/users', {
  method: 'POST',
  headers: {
    "Content-Type": "application/json",
    "XSRF-TOKEN": `<value of XSRF-TOKEN cookie>`
  },
  body: JSON.stringify({
    email: 'spidey@spider.man',
    username: 'Spidey',
    password: 'password'
  })
}).then(res => res.json()).then(data => console.log(data));
```

Remember to replace the `<value of XSRF-TOKEN cookie>` with the value of the `XSRF-TOKEN` cookie found in your browser's dev tools. If you don't have the `XSRF-TOKEN` cookie anymore, access the http://localhost:5000/hello/world route to add the cookie back.
Next, try to hit the Sequelize model validation errors by testing the following which should give back a `Validation error`:

- `email` is not unique (signup with an existing `email`)
- `username` is not unique (signup with an existing `username`)

If you don't see the `Validation error` for any of these, check the syntax in your `backend/db/models/user.js` model file.

Commit your code for the signup route once you are done testing!

Get Session User API Route

The `GET /api/session` get session user route will return the session user as JSON under the key of `user` . If there is not a session, it will return a JSON with an empty object. To get the session user, connect the `restoreUser` middleware.
Add the route to the `router` in the `backend/routes/api/session.js` file.

```
// backend/routes/api/session.js
// ...

// Restore session user
router.get(
  '/',
  restoreUser,
  (req, res) => {
    const { user } = req;
    if (user) {
      return res.json({
        user: user.toSafeObject()
      });
    } else return res.json({});
  }
);

// ...
```

Test the Get Session User Route

Test the route by navigating to http://localhost:5000/api/session. You should see the current session user information if you have the `token` cookie. If you don't have a token cookie, you should see an empty object returned.
If you don't have the `XSRF-TOKEN` cookie anymore, access the http://localhost:5000/hello/world route to add the cookie back.
If you don't see this expected behavior, then check your backend server logs in your terminal where you ran `npm start` and the syntax in the `session.js` route file and the `restoreUser` middleware function.

Commit your code for the get session user route once you are done testing!

Phase 5: Validating the Request Body

Before using the information in the body of the request, it's good practice to validate the information.

You will use a package, `express-validator`, to validate the body of the requests for routes that expect a request body. The `express-validator` package has two functions, `check` and `validationResult` that are used together to validate the request body. `check` is a middleware function creator that checks a particular key on the request body. `validationResult` gathers the results of all the `check` middlewares that were run to determine which parts of the body are valid and invalid.
In the `backend/utils` folder, add a file called `validation.js`. In this file, define an Express middleware called `handleValidationErrors` that will call `validationResult` from the `express-validator` package passing in the request. If there are no validation errors returned from the `validationResult` function, invoke the

next middleware. If there are validation errors, create an error with all the validation error messages and invoke the next error-handling middleware.

```
// backend/utils/validation.js
const { validationResult } = require('express-validator');

// middleware for formatting errors from express-validator
middleware
// (to customize, see express-validator's documentation)
const handleValidationErrors = (req, _res, next) => {
  const validationErrors = validationResult(req);

  if (!validationErrors.isEmpty()) {
    const errors = validationErrors
      .array()
      .map((error) => `${error.msg}`);

    const err = Error('Bad request.');
    err.errors = errors;
    err.status = 400;
    err.title = 'Bad request.';
    next(err);
  }
  next();
};

module.exports = {
  handleValidationErrors,
};
```

The `handleValidationErrors` function is exported at the bottom of the file. You will test this function later when it's used.

Here's another great time to commit!

Validating Login Request Body

In the `backend/routes/api/session.js` file, import the `check` function from `express-validator` and the `handleValidationError` function you just created.

```
// backend/routes/api/session.js
// ...
const { check } = require('express-validator');
const { handleValidationErrors } =
require('../../utils/validation');
// ...
```

The `check` function from `express-validator` will be used with the `handleValidationErrors` to validate the body of a request.

The `POST /api/session` login route will expect the body of the request to have a key of `credential` with either the `username` or `email` of a user and a key of `password` with the password of the user.
Make a middleware called `validateLogin` that will check these keys and validate them:

```
// backend/routes/api/session.js
// ...

const validateLogin = [
  check('credential')
    .exists({ checkFalsy: true })
    .notEmpty()
    .withMessage('Please provide a valid email or username.'),
  check('password')
    .exists({ checkFalsy: true })
    .withMessage('Please provide a password.'),
  handleValidationErrors,
];
```

The `validateLogin` middleware is composed of the `check` and `handleValidationErrors` middleware. It checks to see if `req.body.credential` and `req.body.password` is not empty. If one of them is empty, then an error will be returned as the response.
Next, connect the `POST /api/session` route to the `validateLogin` middleware. Your login route should now look like this:

```
// backend/routes/api/session.js
// ...

// Log in
router.post(
  '/',
  validateLogin,
  asyncHandler(async (req, res, next) => {
    const { credential, password } = req.body;

    const user = await User.login({ credential, password });

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

    await setTokenCookie(res, user);

    return res.json({
      user,
    });
  }),
);
```

Test the Login Validation

Test `validateLogin` by navigating to the http://localhost:5000/hello/world test route and making a fetch request from the browser's dev tools console. Remember, you need to pass in the value of the `XSRF-TOKEN` cookie as a header in the fetch request because the login route has a `POST` HTTP verb.
If at any point you don't see the expected behavior while testing, check your backend server logs in your terminal where you ran `npm start`. Also, check the syntax in the `users.js` route file as well as the `handleValidationErrors` middleware.
Try setting the `credential` user field to an empty string. You should get a `Bad Request` error back.

```
fetch('/api/session', {
  method: 'POST',
  headers: {
    "Content-Type": "application/json",
    "XSRF-TOKEN": `<value of XSRF-TOKEN cookie>`
  },
  body: JSON.stringify({ credential: '', password: 'password' })
}).then(res => res.json()).then(data => console.log(data));
```

Remember to replace the `<value of XSRF-TOKEN cookie>` with the value of the `XSRF-TOKEN` cookie found in your browser's dev tools. If you don't have the `XSRF-TOKEN` cookie anymore, access the http://localhost:5000/hello/world route to add the cookie back.
Test the `password` field by setting it to an empty string. You should get a `Bad Request` error back with `Please provide a password` as one of the errors.

```
fetch('/api/session', {
  method: 'POST',
  headers: {
    "Content-Type": "application/json",
    "XSRF-TOKEN": `<value of XSRF-TOKEN cookie>`
  },
  body: JSON.stringify({ credential: 'Demo-lition', password: '' })
}).then(res => res.json()).then(data => console.log(data));
```

Once you finish testing, commit your code!

Validating Signup Request Body

In the `backend/routes/api/users.js` file, import the `check` function from `express-validator` and the `handleValidationError` function you created.

```
// backend/routes/api/users.js
// ...
const { check } = require('express-validator');
const { handleValidationErrors } =
require('../../utils/validation');
// ...
```

The `POST /api/users` signup route will expect the body of the request to have a key of `username`, `email` and `password` with the password of the user being created.
Make a middleware called `validateSignup` that will check these keys and validate them:

```
// backend/routes/api/users.js
// ...
const validateSignup = [
  check('email')
    .exists({ checkFalsy: true })
    .isEmail()
    .withMessage('Please provide a valid email.'),
  check('username')
    .exists({ checkFalsy: true })
    .isLength({ min: 4 })
    .withMessage('Please provide a username with at least 4
characters.'),
  check('username')
    .not()
    .isEmail()
    .withMessage('Username cannot be an email.'),
  check('password')
    .exists({ checkFalsy: true })
    .isLength({ min: 6 })
    .withMessage('Password must be 6 characters or more.'),
  handleValidationErrors,
];
```

The `validateSignup` middleware is composed of the `check` and `handleValidationErrors` middleware. It checks to see if `req.body.email` exists and is an email, `req.body.username` is a minimum length of 4 and is not an email, and `req.body.password` is not empty and has a minimum length of 6. If at least one of the `req.body` values fail the check, an error will be returned as the response.
Next, connect the `POST /api/users` route to the `validateSignup` middleware. Your signup route should now look like this:

```
// backend/routes/api/users.js
// ...

// Sign up
router.post(
  '/',
  validateSignup,
  asyncHandler(async (req, res) => {
    const { email, password, username } = req.body;
    const user = await User.signup({ email, username, password });
    await setTokenCookie(res, user);

    return res.json({
      user,
    });
  }),
);
```

Test the Signup Validation

Test `validateSignup` by navigating to the http://localhost:5000/hello/world test route and making a fetch request from the browser's dev tools console. Remember, you need to pass in the value of the `XSRF-TOKEN` cookie as a header in the fetch request because the login route has a `POST` HTTP verb.
If at any point you don't see the expected behavior while testing, check your backend server logs in your terminal where you ran `npm start`. Also, check the syntax in the `users.js` route file as well as the `handleValidationErrors` middleware.
First, test the signup route with an empty `password` field. You should get a `Bad Request` error back with `'Please provide a password'` as one of the errors.

```
fetch('/api/users', {
  method: 'POST',
  headers: {
    "Content-Type": "application/json",
    "XSRF-TOKEN": `<value of XSRF-TOKEN cookie>`
  },
  body: JSON.stringify({
    email: 'spidey@spider.man',
    username: 'Spidey',
    password: ''
  })
}).then(res => res.json()).then(data => console.log(data));
```

Remember to replace the `<value of XSRF-TOKEN cookie>` with the value of the `XSRF-TOKEN` cookie found in your browser's dev tools. If you don't have the `XSRF-TOKEN` cookie anymore, access the http://localhost:5000/hello/world route to add the cookie back.
Then try to signup with more invalid fields to test out the checks in the `validateSignup` middleware. Make sure to cover each of the following test cases which should give back a `Bad Request` error:

- `email` field is an empty string
- `email` field is not an email
- `username` field is an empty string
- `username` field is only 3 characters long
- `username` field is an email
- `password` field is only 5 characters long

If you don't see the `Bad Request` error for any of these, check your syntax for the `validateSignup` middleware.

Commit your code once you're done testing!

Wrapping up the Backend

You can now remove the `GET /hello/world` test routes. **Do not remove the `POST /api/test` route just yet. You will be using it in the next part.**

Awesome work! You just finished setting up the entire backend for this project! In the next part, you will implement the React frontend.

Did you find this lesson helpful?