Backend

In this part, we will

- Initialize backend using `npm` and install necessary packages
- Set up a MongoDB database
- Set up a server with Node and Express
- Create a database schema to define a User for registration and login purposes
- Set up two API routes, `register` and `login`, using `passport` + `jsonwebtoken` for authentication and `validator` for input validation
- Test API routes using Postman

Initializing the project

Set the current directory to wherever you want your project to live and initialize the project using `npm`.

```
mkdir fundedu
cd fundedu
npm init
```

After running the command, a utility will walk you through creating a `package.json` file. You can enter through most of these safely, but go ahead and set the entry point to `server.js` instead of the default `index.js` when prompted.

Setting up `package.json`

1. Set the "main" entry point to "server.js" instead of the default "index.js", if you haven't done so already.
2. Install the following dependencies using `npm`

```
npm i bcryptjs body-parser concurrently express is-empty jsonwebtoken mongoose
passport passport-jwt validator
```

A brief description of each package and the function it will serve:

- `bcryptjs`: used to hash passwords before storing them in the database
- `body-parser`: used to parse incoming requests
- `concurrently`: allows us to run our backend and frontend concurrently and on different ports
- `express`: sits on top of Node to make the routing, request handling, and responding easier to write
- `is-empty`: global function that will come in handy when we use `validator`
- `jsonwebtoken`: used for authorization
- `mongoose`: used to interact with MongoDB
- `passport`: used to authenticate requests, which it does through an extensible set of plugins known as strategies

- `passport-jwt`: passport strategy for authenticating with a JSON Web Token (JWT); lets you authenticate endpoints using a JWT
- `validator`: used to validate inputs (e.g. check for valid email format, confirming passwords match)

3. Install the following devDependency (-D) using `npm`

```
npm i -D nodemon
```

Nodemon is a utility that will monitor for any changes in your code and automatically restart your server, which is perfect for development.

4. Change the "scripts" object to the following

```
"scripts": {
    "start": "node server.js",
    "server": "nodemon server.js",
},
```

Later on, we'll use `nodemon run server` to run our dev server.

Setting up our database

Create a `config` directory and within it a `keys.js` file

In your `keys.js` file, place the following:

```
module.exports = {
  mongoURI: "YOUR_MONGOURI_HERE"
};
```

Setting up our server with Node and Express

The basic flow for our server setup is as follows.

- Pull in required dependencies (namely `express`, `mongoose` and `bodyParser`)
- Initialize app using `express()`
- Apply the middleware function for `bodyparser` so we can use it
- Pull in our MongoURI from `keys.js` file and connect to MongoDB database
- Set the port for the server to run on and have our app listen on this port

Run `npm run server` and the server should run and connect to MongoDB.

Setting up database schema

Create a `models` folder to define the user schema. Within `models`, create a `User.js` file.

Within `User.js`, we will

- Pull in required dependencies
- Create a Schema to represent a User, defining fields and types as objects of the Schema
- Export the model so we can access it outside of this file

User schema should contani the following:

- `name`, type = String, required
- `email`, type = String, required
- `password`, type = String, required
- `userType`, type = String, required
- `uniqid`, type = String, required
- `date`, type = Date, default = Date.now

Setting up form validation

Before setting up routes, create a directory called `validation` and create a `register.js` and `login.js` file for each route's validation.

Validation flow for `register.js` file will go as follows:

- Pull in `validator` and `is-empty` dependencies
- Export the function `validateRegisterInput`, which takes in `data` as a parameter (sent from our frontend registration form, which we'll build later)
- Instantiate `errors` object
- Convert all empty fields to an empty string before running validation checks (`validator` only works with strings)
- Check for empty fields, valid email formats, password requirements and confirm password equality using `validator` functions
- Return `errors` object with any and all errors contained as well as an `isValid` boolean that checks to see if we have any errors

Validation for `login.js` follows an identical flow to the above, but checks only email and password and exports `validateLoginInput`.

Setting up API routes

Create a new folder for our api routes:

```
mkdir routes
cd routes
mkdir api
```

In `api`, create a `users.js` file for registration and login.

At the top of users.js, pull in the required dependencies and load the input validations & user model.

```
const express = require("express");
const router = express.Router();
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const keys = require("../../config/keys");

// Load input validation
const validateRegisterInput = require("../../validation/register");
const validateLoginInput = require("../../validation/login");

// Load User model
const User = require("../../models/User");
```

Create the Register endpoint

For the `register` endpoint, we will:

- Pull the `errors` and `isValid` variables from the `validateRegisterInput(req.body)` function and check input validation
- If valid input, use MongoDB's `User.findOne()` to see if the user already exists
- If user is a new user, fill in the fields with data sent in the body of the request
- Use `bcryptjs` to hash the password before storing it in the database

Setup passport

In the `config` directory, create a `passport.js` file.

Before setting up `passport`, add the following to the `keys.js` file.

```
module.exports = {
  mongoURI: "YOUR_MONGOURI_HERE",
  secretOrKey: "secret"
};
```

You can read more about the `passport-jwt` strategy at this link. It describes how the JWT authentication strategy is constructed, including parameters, variables and functions such as `options`, `secretOrKey`, `jwtFromRequest`, `verify`, and `jwt_payload`.

Put the following in the `passport.js` file:

```
const JwtStrategy = require("passport-jwt").Strategy;
const ExtractJwt = require("passport-jwt").ExtractJwt;
const mongoose = require("mongoose");
const User = mongoose.model("users");
const keys = require("../config/keys");
const opts = {};

opts.jwtFromRequest = ExtractJwt.fromAuthHeaderAsBearerToken();
```

```
opts.secretOrKey = keys.secretOrKey;

module.exports = passport => {
  passport.use(
    new JwtStrategy(opts, (jwt_payload, done) => {
      User.findById(jwt_payload.id)
        .then(user => {
          if (user) {
            return done(null, user);
          }
          return done(null, false);
        })
        .catch(err => console.log(err));
    })
  );
};
```

Also, note that the `jwt_payload` will be sent via our login endpoint below.

Create the Login endpoint

For our login endpoint, we will

- Pull the `errors` and `isValid` variables from the `validateLoginInput(req.body)` function and check input validation
- If valid input, use MongoDB's `User.findOne()` to see if the user exists
- If user exists, use `bcryptjs` to compare submitted password with hashed password in the database
- If passwords match, create the JWT Payload
- Sign the jwt, including payload, `keys.secretOrKey` from `keys.js`, and set a `expiresIn` time (in seconds)
- If successful, append the token to a Bearer string (remember in `passport.js`, we `setopts.jwtFromRequest = ExtractJwt.fromAuthHeaderAsBearerToken();`)

Export the router at the bottom of `users.js`.

```
module.exports = router;
```

Make the following additions to `server.js`:

```
// near the top
const passport = require("passport");
const users = require("./routes/api/users");
```

```
// Passport middleware
app.use(passport.initialize());
```

```
// Passport config
require("./config/passport")(passport);

// Routes
app.use("/api/users", users);
```

Testing our API routes using Postman

Testing Register endpoint

Open Postman and

- Set the request type to `POST`
- Set the request url to http://localhost:5000/api/users/register
- Navigate to the Body tab, select `x-www-form-urlencoded`, fill in your registration parameters and hit Send

You should receive a HTTP status response of 200 OK and have the new user returned as JSON.

Check your database with Robo3T and you should see a new user created with the above credentials.

Testing Login endpoint

In Postman:

- Set the request type to `POST`
- Set the request url to http://localhost:5000/api/users/login
- Navigate to the Body tab, select `x-www-form-urlencoded`, fill in your login parameters and hit Send

You should receive a HTTP status response of 200 OK and have the jwt returned in the response.

Testing errors

Test validation errors in cases where a user signs up and logs in (e.g. invalid email formats, passwords that don't match). When you test the API out in Postman, you should see your `errors` object returned.