MoFlix Rest API & Database

Artin Azimi 2021

The Challenge

Building server-side code (REST API and database). This was a 2 part project in which the REST API and database was first built, then, React was used to create the interface to display the API in a well designed fashioned. This case study will only focus on the server-side code.

React Rest API MongoDB

Access to my code for the project can be found here:

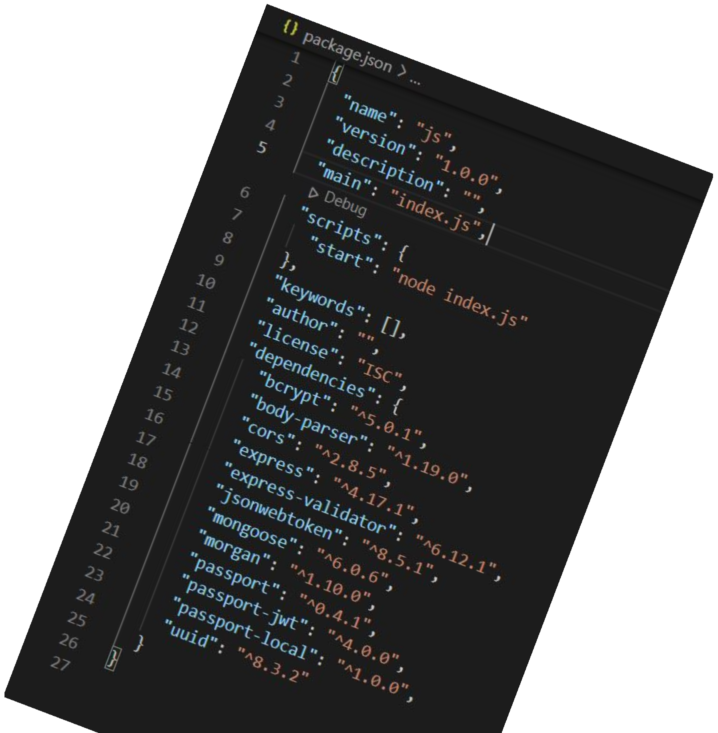
https://github.com/ArtinCF93/movie_flix_api

Objective Planning

Understanding the steps needed to take to create a Rest API and database are essential

1. Package Setup
2. Schema and Models
3. Non-Relational Database
4. Request Routing
5. Authentication & Authorization

This in return allowed me to understand what frameworks and setups I needed to make. To start off, creating a package for my project using Node.JS for dependencies. Meaning for step 2; Schema and Models, I would need to install mongoose. For step 3; Non-relational database, I would use the reliable and popular MongoDB. For step 4; Request Routing, I would need to install the Express framework. For step 5; Authentication and Authorization, I would need to install bcrypt, cors, jsonwebtoken, and passport.



```json
{
    "name": "js",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "start": "node index.js"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {
        "bcrypt": "^5.0.1",
        "body-parser": "^1.19.0",
        "cors": "^2.8.5",
        "express": "^4.17.1",
        "express-validator": "^6.12.1",
        "jsonwebtoken": "^8.5.1",
        "mongoose": "^6.0.6",
        "morgan": "^1.10.0",
        "passport": "^0.4.1",
        "passport-jwt": "^4.0.0",
        "passport-local": "^1.0.0",
        "uuid": "^8.3.2"
    }
}
```

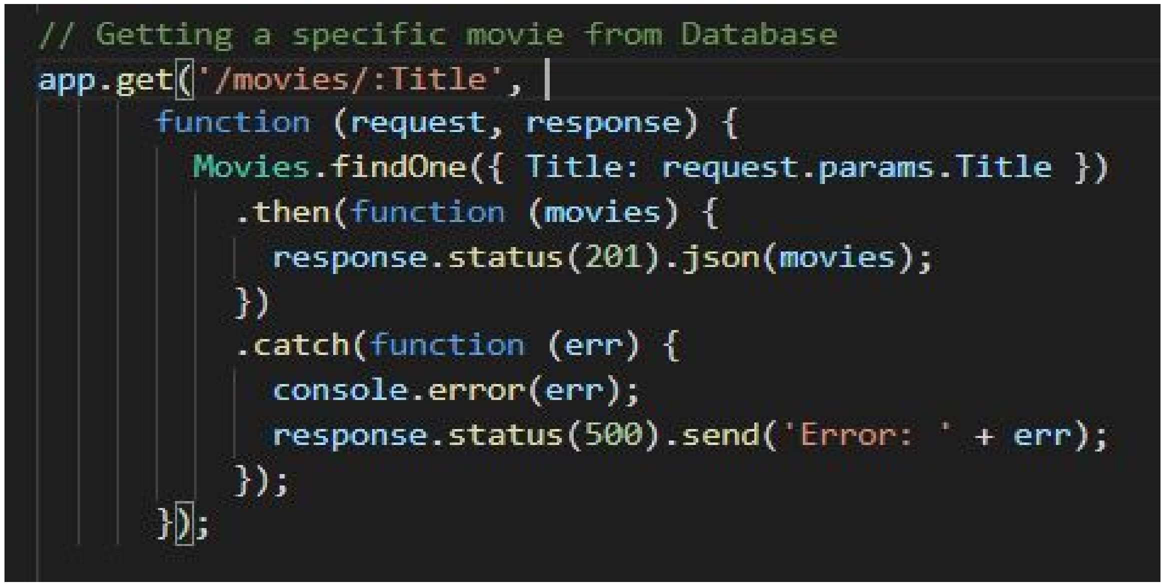
Step 4 of creating the Rest API and database was to create request routing using express. A web server is what processes request/response processes—any request to the web server set up on a later to be deployed website.There are multiple HTTP request methods, each of which specifies a different type of data transfer over the Hypertext Transfer Protocol. These methods include

GET PUT POST DELETE

These requests follows an app.METHOD(PATH, HANDLER) method. HTTP requests along with CRUD operations (Create, Read, Update, Delete) create the logic for an action. To start, I installed Express framework to my project's package and required at the top of my code.

Request Routing with Express

GET



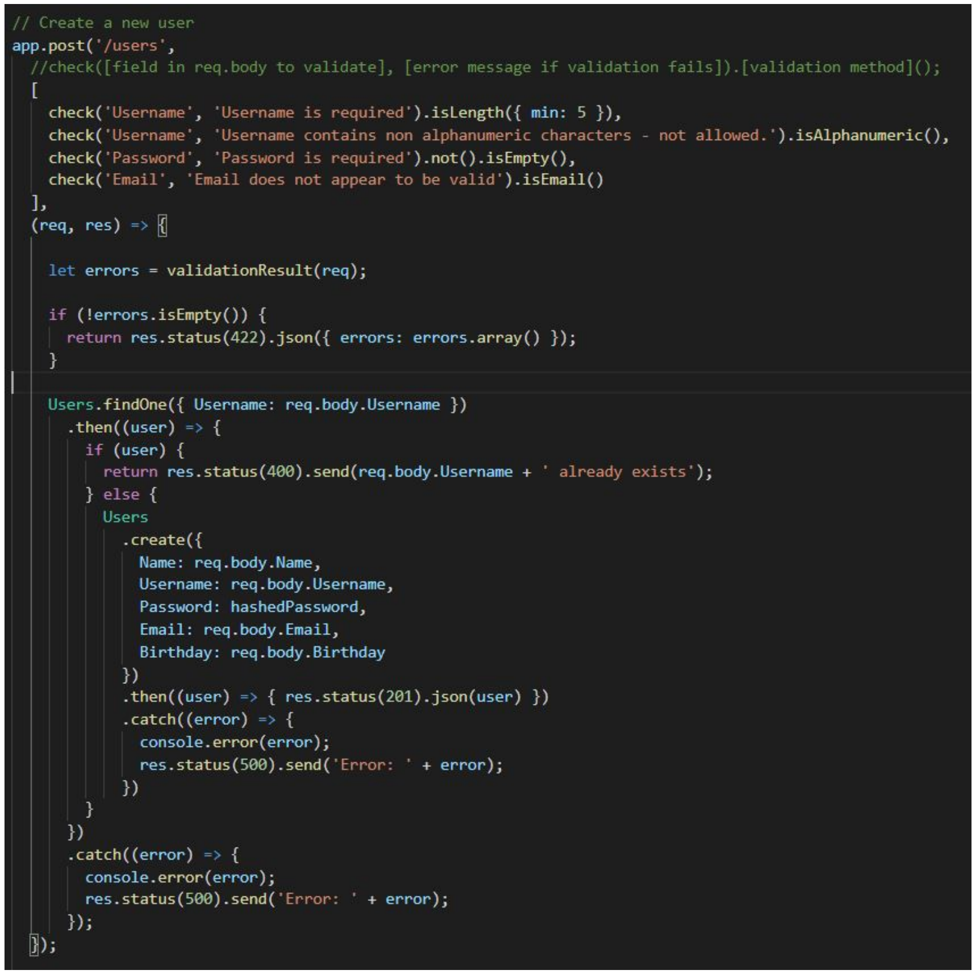
```
// Getting a specific movie from Database
app.get('/movies/:Title',
      function (request, response) {
        Movies.findOne({ Title: request.params.Title })
          .then(function (movies) {
            response.status(201).json(movies);
          })
          .catch(function (err) {
            console.error(err);
            response.status(500).send('Error: ' + err);
          });
      });
```

This is one of the examples of the GET operations created for the project. This GET operation is intended to pull data of a specific movie. The url path is identified as '/movies/:Title' with ':Title' being arbitrary to match the parameter. The function calls the 'Movies' model in which the movies collection in the database is connected to. This instructs the function to loop through the array of objects and to match the request parameter with the Title of the object, to return a response of the movie JSON object or an error.

Request Routing with Express

Continued...

POST

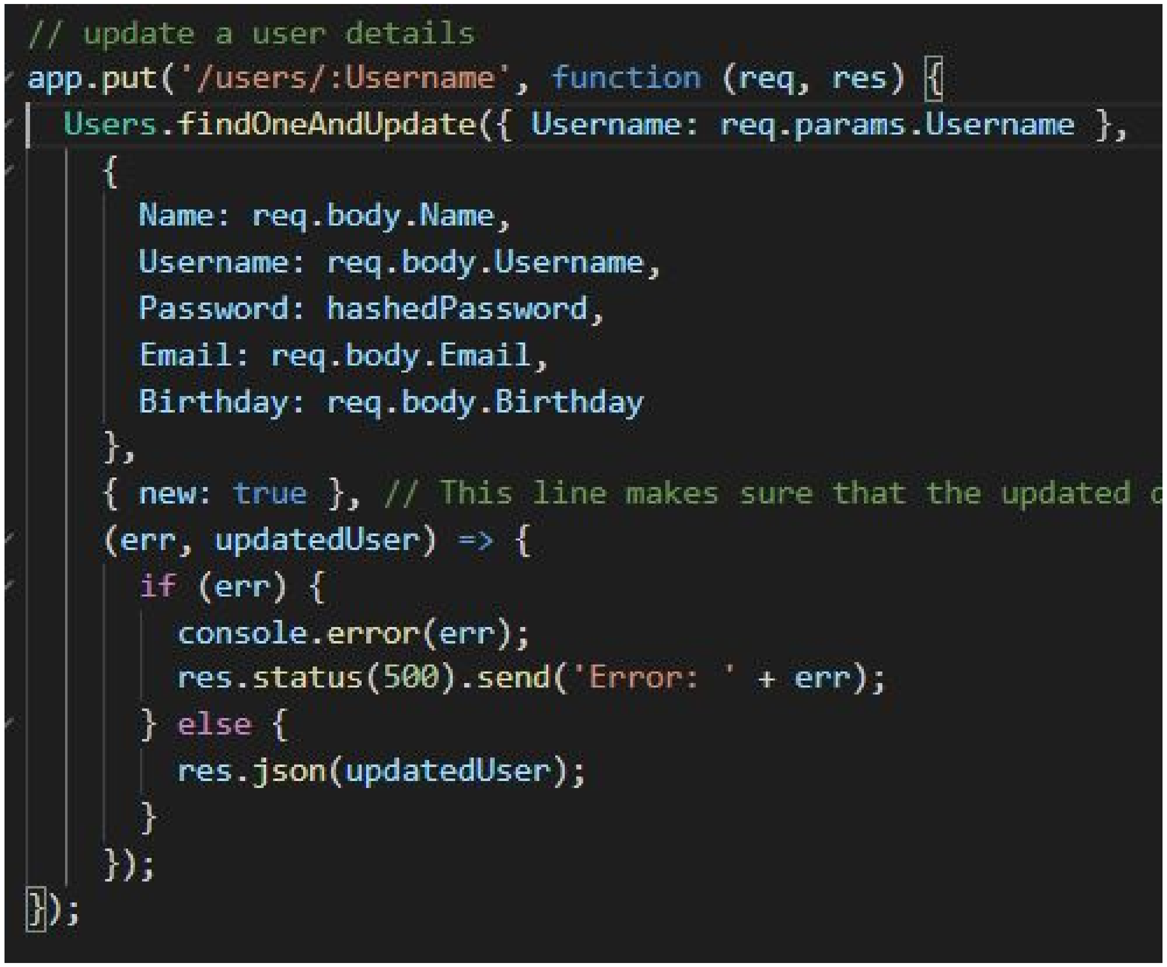


```
// Create a new user
app.post('/users',
  //check([field in req.body to validate], [error message if validation fails]).[validation method]();
  [
    check('Username', 'Username is required').isLength({ min: 5 }),
    check('Username', 'Username contains non alphanumeric characters - not allowed.').isAlphanumeric(),
    check('Password', 'Password is required').not().isEmpty(),
    check('Email', 'Email does not appear to be valid').isEmail()
  ],
  (req, res) => {

    let errors = validationResult(req);

    if (!errors.isEmpty()) {
      return res.status(422).json({ errors: errors.array() });
    }

    Users.findOne({ Username: req.body.Username })
      .then((user) => {
        if (user) {
          return res.status(400).send(req.body.Username + ' already exists');
        } else {
          Users
            .create({
              Name: req.body.Name,
              Username: req.body.Username,
              Password: hashedPassword,
              Email: req.body.Email,
              Birthday: req.body.Birthday
            })
            .then((user) => { res.status(201).json(user) })
            .catch((error) => {
              console.error(error);
              res.status(500).send('Error: ' + error);
            })
        }
      })
      .catch((error) => {
        console.error(error);
        res.status(500).send('Error: ' + error);
      });
  });
```

This post operation is to create a new user. The first part of the function; the path is identified as '/users'. Second, instructions are laid out to check for certain conditions and a response error. It then calls the Users modal to loop through the Users collection. If the 'Username' in the request body matches a 'Username' in the Users collection; it will return a response error, otherwise, it will create a new JSON object in the Users collection with the indicated entries.

Continued...

PUT

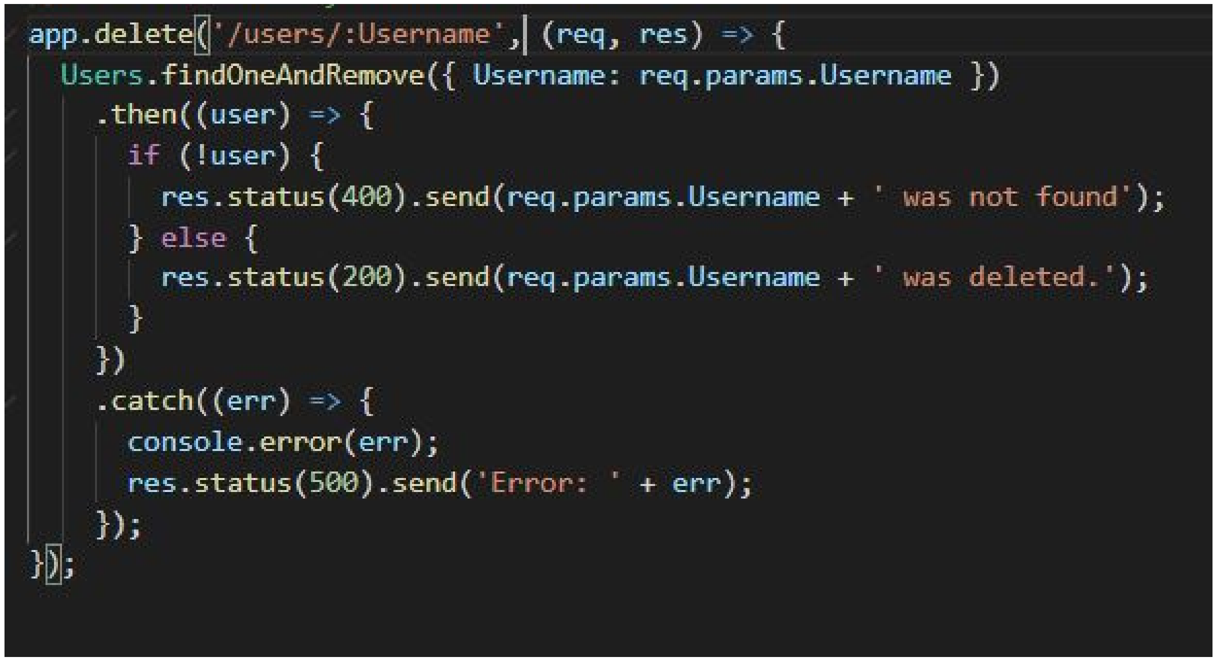


```
// update a user details
app.put('/users/:Username', function (req, res) {
  Users.findOneAndUpdate({ Username: req.params.Username },
    {
      Name: req.body.Name,
      Username: req.body.Username,
      Password: hashedPassword,
      Email: req.body.Email,
      Birthday: req.body.Birthday
    },
    { new: true }, // This line makes sure that the updated d
    (err, updatedUser) => {
      if (err) {
        console.error(err);
        res.status(500).send('Error: ' + err);
      } else {
        res.json(updatedUser);
      }
    });
});
```

This is one of the example of the PUT operations created for the project. This PUT operation is intended to update a user's details. The function calls the the Users modal to loop through the Users collection to identify the 'Username' that matches the parameter. The function then is instructed to update each attribute of the matched user with the request body with the .findOneAndUpdate() function. { new: true } is stated to make sure that the updated document is returned. Another if statement is made in case of an error, otherwise return the updated user.

Continued...

DELETE



```
app.delete('/users/:Username', (req, res) => {
  Users.findOneAndRemove({ Username: req.params.Username })
    .then((user) => {
      if (!user) {
        res.status(400).send(req.params.Username + ' was not found');
      } else {
        res.status(200).send(req.params.Username + ' was deleted.');
      }
    })
    .catch((err) => {
      console.error(err);
      res.status(500).send('Error: ' + err);
    });
});
```

This is one of the examples of the DELETE operations created for the project. This DELETE operation is intended to delete a user's details. The function calls the the Users modal to loop through the Users collection to identify the 'Username' that matches the parameter. The function then is instructed to delete the matched user with the .findOneAndRemove() function. It then follows by following the condition that if a user is not found, it responds with an error message, otherwise a confirmation message. Of course a catch error at the end if anything wrong occurred.

Authorization & Authentication

Securing information, setting up authentication and authorization was the last part of the project. To set things up, there were multiple libraries to install to my project package.

passport **passport local** **passport-jwt** **jsonwebtokens**

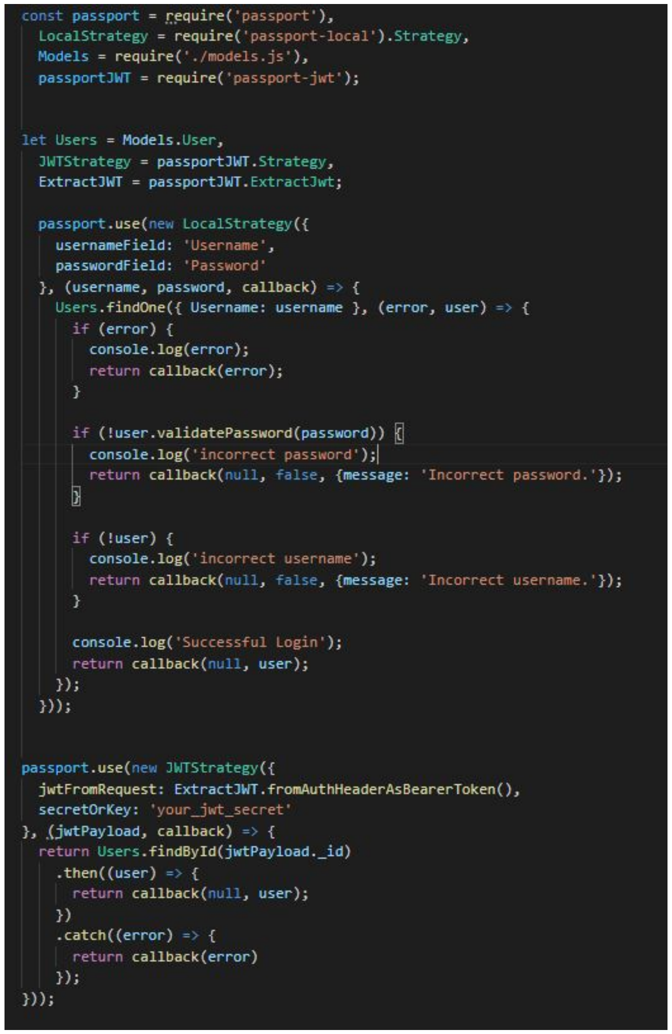
For my API, I used Passport to implement basic HTTP authentication to log registered users into my application, as well as JWT authentication for subsequent requests to my API.

Passport

I started off by creating a new passport.js file to configure my new passport strategies with requiring the necessary libraries at the top of the file.

In this code, two Passport strategies are defined.

The first one, called “LocalStrategy,” defines my basic HTTP authentication for login requests. LocalStrategy takes a username and password from the request body and uses the 'Users' modal built by Mongoose to check my database for a user with the same username. If there’s a match, the callback function will be executed (this will be my login endpoint, which will be explored in a bit).



```javascript
const passport = require('passport'),
  LocalStrategy = require('passport-local').Strategy,
  Models = require('./models.js'),
  passportJWT = require('passport-jwt');


let Users = Models.User,
  JWTStrategy = passportJWT.Strategy,
  ExtractJWT = passportJWT.ExtractJwt;

  passport.use(new LocalStrategy({
    usernameField: 'Username',
    passwordField: 'Password'
  }, (username, password, callback) => {
    Users.findOne({ Username: username }, (error, user) => {
      if (error) {
        console.log(error);
        return callback(error);
      }

      if (!user.validatePassword(password)) {
        console.log('incorrect password');
        return callback(null, false, {message: 'Incorrect password.'});
      }

      if (!user) {
        console.log('incorrect username');
        return callback(null, false, {message: 'Incorrect username.'});
      }

      console.log('Successful Login');
      return callback(null, user);
    });
  }));


passport.use(new JWTStrategy({
  jwtFromRequest: ExtractJWT.fromAuthHeaderAsBearerToken(),
  secretOrKey: 'your_jwt_secret'
}, (jwtPayload, callback) => {
  return Users.findById(jwtPayload._id)
    .then((user) => {
      return callback(null, user);
    })
    .catch((error) => {
      return callback(error)
    });
}));
```

Passport - jwt

Setting up the JWT authentication, i called the strategy,“JWTStrategy,” and it allows me to authenticate users based on the JWT submitted alongside their request. In the code, the JWT is extracted from the header of the HTTP request. This JWT is called the “bearer token”

It is essential that I create a 'secretOrKey' which i have named it generically 'your_jwt_secret' for project purposes. This strategy also loops through the 'Users' modal which is connected to the Users collection to payload the jwt token.

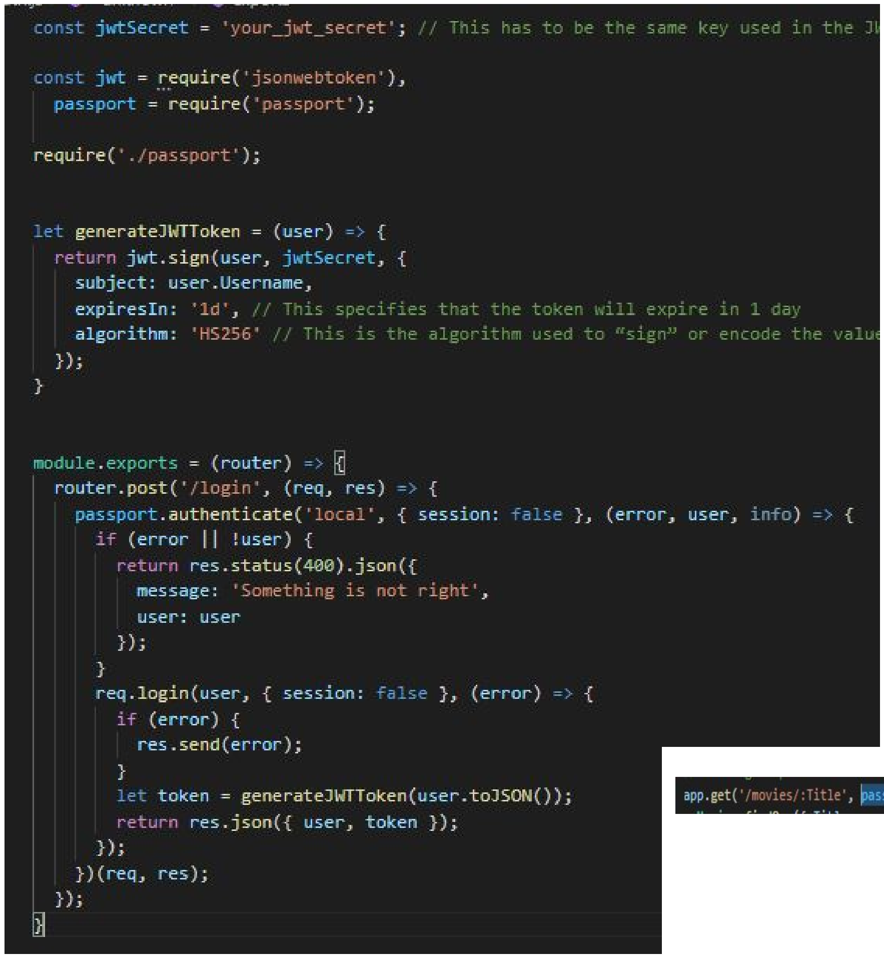
Authorization & Authentication

Continued...

Now that I put Passport strategies in place, I moved on to create the actual logic that will authenticate registered users when they log in using their username and password (basic HTTP authentication), as well as generating a JWT that will authenticate their future requests.

For the first and most important part to start was to create a new new auth.js file and import the passport file as a module. It was essential to import the jwt secretKey as that essentially holds the LocalStrategy defined in the passport.js.

I first created a function saved and named generateJWTToken(), which creates a token for the specified username, its expiration time, and the algorithm (used to “sign” or encode the values of the JWT).

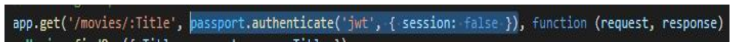


```
const jwtSecret = 'your_jwt_secret'; // This has to be the same key used in the JW

const jwt = require('jsonwebtoken'),
  passport = require('passport');

require('./passport');


let generateJWTToken = (user) => {
  return jwt.sign(user, jwtSecret, {
    subject: user.Username,
    expiresIn: '1d', // This specifies that the token will expire in 1 day
    algorithm: 'HS256' // This is the algorithm used to “sign” or encode the value
  });
}


module.exports = (router) => {
  router.post('/login', (req, res) => {
    passport.authenticate('local', { session: false }, (error, user, info) => {
      if (error || !user) {
        return res.status(400).json({
          message: 'Something is not right',
          user: user
        });
      }
      req.login(user, { session: false }, (error) => {
        if (error) {
          res.send(error);
        }
        let token = generateJWTToken(user.toJSON());
        return res.json({ user, token });
      });
    })(req, res);
  });
}
```

I then moved onto create a new route for the login process.

The logic of the function being that if the user does not pass the passport.authenticate function, return an error, otherwise generate a jwt token for the requested logged in user and return the user with a token.

Once creating both the passport.js, and auth.js; it was time to add the jwt authentication strategy to the rest of my url endpoints. To apply JWT authentication to a specific endpoint, I would pass it as a second parameter between the URL and callback function as such;



```
app.get('/movies/:Title', passport.authenticate('jwt', { session: false }), function (request, response)
```

In Conclusion...

This was a rewarding experience that taught me a lot about how an API and database function. Creating an API is not just about creating the JSON object of data itself but in contrast how you can access, who can have access to the data and how the permission is granted, as well as how that data can be manipulated.

This was a solo project that I worked on as a student at CareerFoudnry Bootcamp for portfolio contribution that took about 2 months. This was a fairly easy to understand concept of how to make a REST API and database. However when working on this project, I accomplished these steps in a different order than explained in this case study. The project was originally outlined in these steps;

1. Package setup

2. Request Routing

3. Schema and models

3. Authorization and Authentication

4. Relational Database MongoDB.

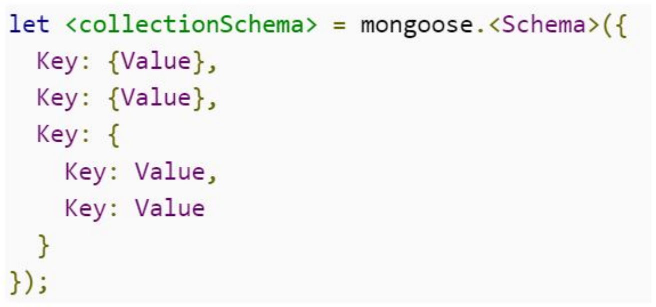
I wanted to set up and explain this case study in the order in which I would complete a similar task in the future. With this being a bootcamp project, the way in which this project was set up with instructions, there was many times I had to go backwards and change previous code that was written drastically which was tedious. I find it more efficient to create things whole step by step than going backwards and forwards. That means understanding what libraries, frameworks, features, files, and functionalities I need to implement to be more organized and timely to not have to go back and change things around drastically because of having to add something later. So hence, I wanted to show that in this case study by explaining things in the order that makes more sense.

Schema and Models

It is important use models to instruct constructions of documents according to a specified schema, to shape your JSON objects or Data. A model specifies what data to store and how to store it for each document in a collection. This whole process is often referred to as business logic.

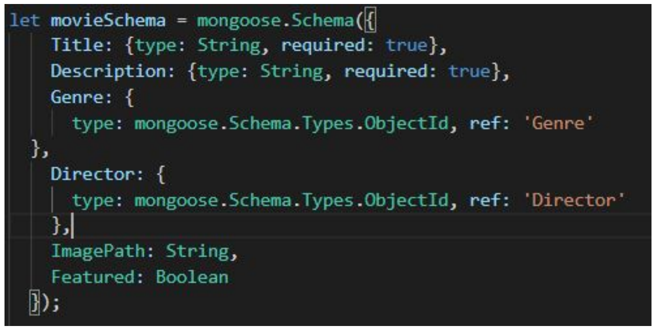
For this project, to build models for my collections, I used and installed Mongoose to my project package.

After importing the package into my "models.js" file, I defined the "schema." which follows the syntax as follows



```
let <collectionSchema> = mongoose.<Schema>({
  Key: {Value},
  Key: {Value},
  Key: {
    Key: Value,
    Key: Value
  }
});
```

Following this syntax, my movie schema would be set up as such;

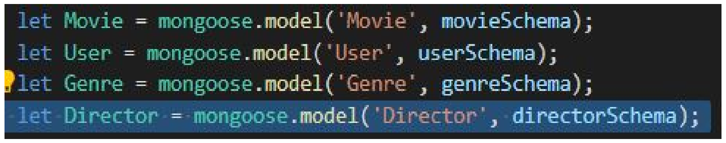


```
let movieSchema = mongoose.Schema({
    Title: {type: String, required: true},
    Description: {type: String, required: true},
    Genre: {
      type: mongoose.Schema.Types.ObjectId, ref: 'Genre'
  },
    Director: {
      type: mongoose.Schema.Types.ObjectId, ref: 'Director'
    },
    ImagePath: String,
    Featured: Boolean
});
```

You will notice the line;

'type: mongoose.Schema.Types.ObjectId, ref: 'Director''

in the 'movieSchema'. This is to reference a different schema so that the data of the 'directorSchema' is nested within the 'movieSchema'. The 'Director' referencing the model created as shown below.

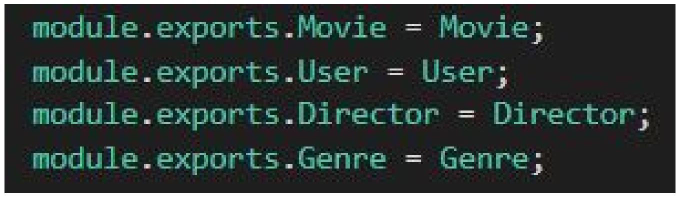


```
let Movie = mongoose.model('Movie', movieSchema);
let User = mongoose.model('User', userSchema);
let Genre = mongoose.model('Genre', genreSchema);
let Director = mongoose.model('Director', directorSchema);
```

Schema and Models

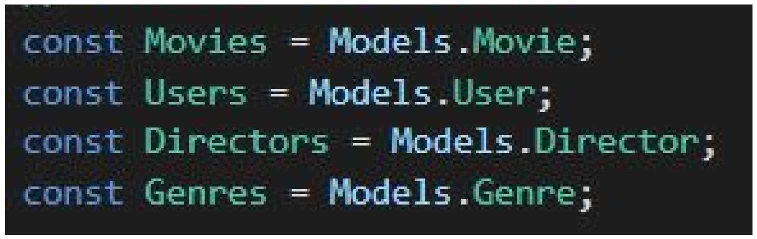
Continued...

The next step is to export the models to use in the the main documents where request routing is written; in this case, my index.js file



```
module.exports.Movie = Movie;
module.exports.User = User;
module.exports.Director = Director;
module.exports.Genre = Genre;
```

After requiring the models at the top of Index.js, I created variables of each model to be used within the routing requests to be made later

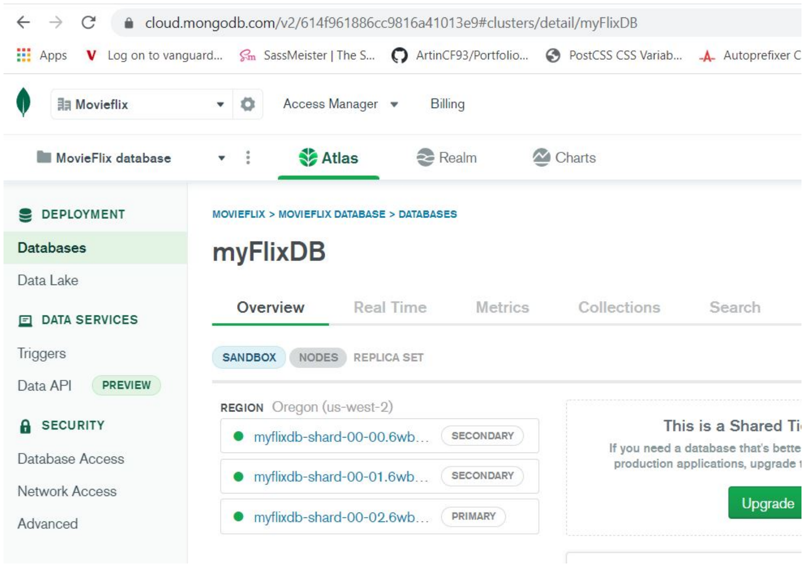


```
const Movies = Models.Movie;
const Users = Models.User;
const Directors = Models.Director;
const Genres = Models.Genre;
```

Non-Relational Database

Having a database is an essential part of storing an API. For this project, i decided to go with MongoDB, a popular NoSQL database. This started with downloading MongDB and setting up my computer environment variable settings to configure the path to be able to use MongoDB in my powershell. However, for this project I created my collections and documents using MongoDB Atlas (web-based graphical user interface for MongoDB) directly to create my API's collections and documents.

I first started out by opening an account with MongoDB. Then moved onto creating a cluster and named it myFlixDB



MOVIEFLIX > MOVIEFLIX DATABASE > DATABASES

myFlixDB

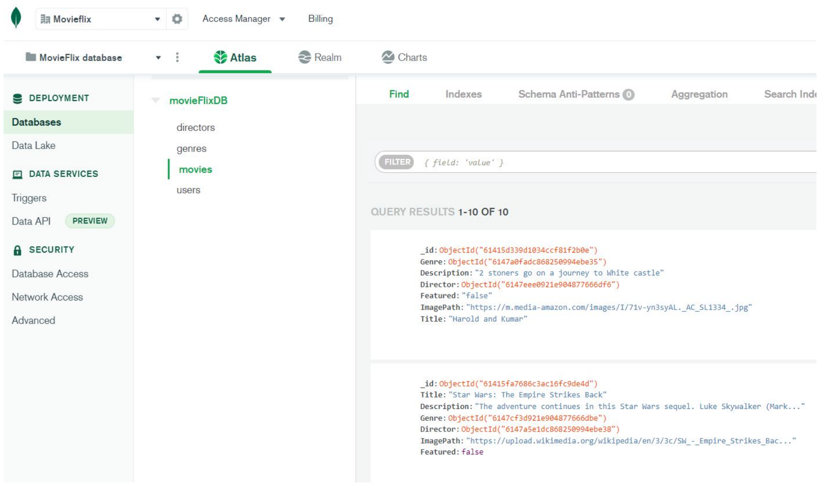
REGION Oregon (us-west-2)

myflixdb-shard-00-00.6wb…	SECONDARY
myflixdb-shard-00-01.6wb…	SECONDARY
myflixdb-shard-00-02.6wb…	PRIMARY

This is a Shared Ti

If you need a database that's bette production applications, upgrade t

I moved on to create a database in myFlixDB cluster and named it movieFlixDB. An API in a database is broken down as collections consisting of documents. Meaning in this instance, I wanted to build an API for a movie collection. The movie collection data consists of a Title, Director, Genre, Director, ImagePath, Feature



movieFlixDB
- directors
- genres
- movies
- users

QUERY RESULTS 1-10 OF 10

```
_id: ObjectId("61415d339d1034ccf81f2b0e")
Genre: ObjectId("6147a0fadc868250994ebe35")
Description: "2 stoners go on a journey to White castle"
Director: ObjectId("6147eee0921e904877666df6")
Featured: "false"
ImagePath: "https://m.media-amazon.com/images/I/71v-yn3syAL._AC_SL1334_.jpg"
Title: "Harold and Kumar"
```

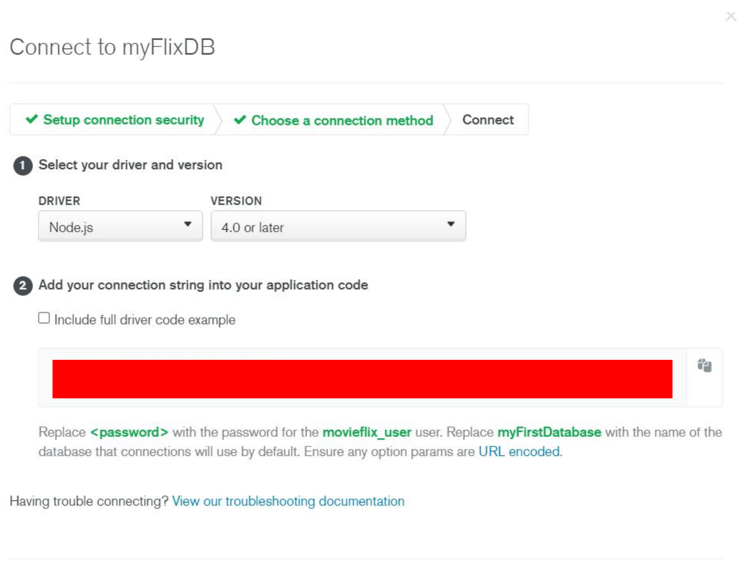
```
_id: ObjectId("61415fa7686c3ac16fc9de4d")
Title: "Star Wars: The Empire Strikes Back"
Description: "The adventure continues in this Star Wars sequel. Luke Skywalker (Mark..."
Genre: ObjectId("6147cf3d921e904877666dbe")
Director: ObjectId("6147a5e1dc868250994ebe38")
ImagePath: "https://upload.wikimedia.org/wikipedia/en/3/3c/SW_-_Empire_Strikes_Bac..."
Featured: false
```

Genre and Director will be separate collections that will be nested within the main movie collection

Non-Relational Database

Continued...

The next step was to connect my database to a deployed site to set up request routing success. In this case, I chose to use Heroku. I first connected it by taking the connection string acquired from the screen below (which is covered in red for my privacy)



Connect to myFlixDB

✓ Setup connection security ✓ Choose a connection method Connect

1 Select your driver and version

DRIVER: Node.js

VERSION: 4.0 or later

2 Add your connection string into your application code

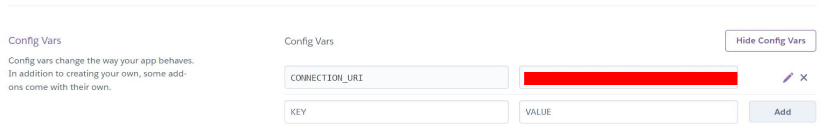
☐ Include full driver code example

[redacted]

Replace **<password>** with the password for the **movieflix_user** user. Replace **myFirstDatabase** with the name of the database that connections will use by default. Ensure any option params are URL encoded.

Having trouble connecting? View our troubleshooting documentation

I then took the connection string and placed it in the 'Configure Vars' input found in the settings tab of my Heroku account.



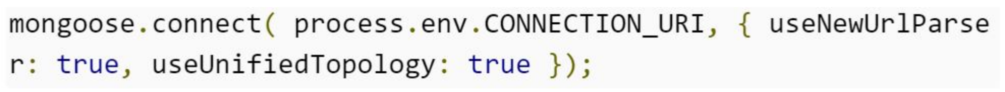
Config Vars

Config vars change the way your app behaves. In addition to creating your own, some add-ons come with their own.

Config Vars

CONNECTION_URI	[redacted]
KEY	VALUE

Back in my “index.js” file, I used mongoose to connect the process.env.CONNECTION_URI to my code which I later used for my request routing



```
mongoose.connect( process.env.CONNECTION_URI, { useNewUrlParser: true, useUnifiedTopology: true });
```