Web App: myFlix

Developer: Eryn Craig

Overview

myFlix is a responsive web app built using MERN stack technology. It provides access to information about movies, including genre and director. Users can also update their profile and add to a favorite list of movies.

Objective:

The main goal of the project was to complete a full-stack app that I could add to my portfolio. I was required to build the client and server side from scratch on my own.

Challenge:

Create a fully-operational Single View Application according to instructions from user stories. From there, using provided wireframes for the design aspects, and following user stories and essential features, build a database(API), and implement a user interface that communicates with the API and includes security measures.

Duration: 2 weeks

Role: Web Developer, Full Stack

Sponsor: CareerFoundry Full Stack Course

Tools Used:

These are a combination of programming libraries, software, processes, frameworks, and languages. With every app, there is usually a large number of libraries installed, which will not be listed here.

- MERN (Mongo, Express, React, Node)
- HTML
- CSS
- JSX
- Redux
- Bootstrap
- Axios
- Parcel
- Mongoose
- REST structure
- PostgreSQL
- SQL
- Postman
- MongoDB
- Heroku
- Netlify

The Process

This project was very long and development intensive, so I will walk you through these 9 steps, split into two sections:

The Server Side:

1. Creating the server side
2. Route HTTP Requests using Express, define Endpoints
3. Model Business Layer Logic
4. Implement Security

The Client Side:

1. Create components and functions
2. Style and Add Routing
3. Implement Routes
4. Import Data from Server & Authenticate Users
5. Host online and locally

1. Creating the Server-Side

This is the beginning of creating the app. This process was broken up into many steps, which I will walk through here.

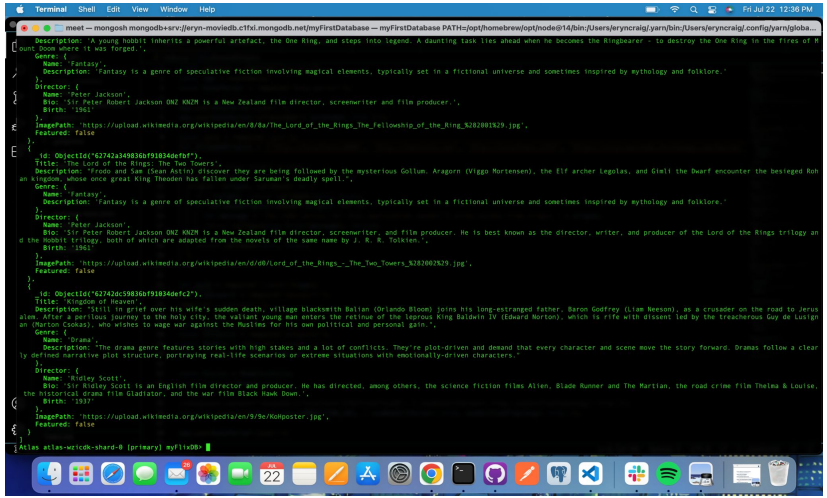
2. Route HTTP Requests using Express, define Endpoints

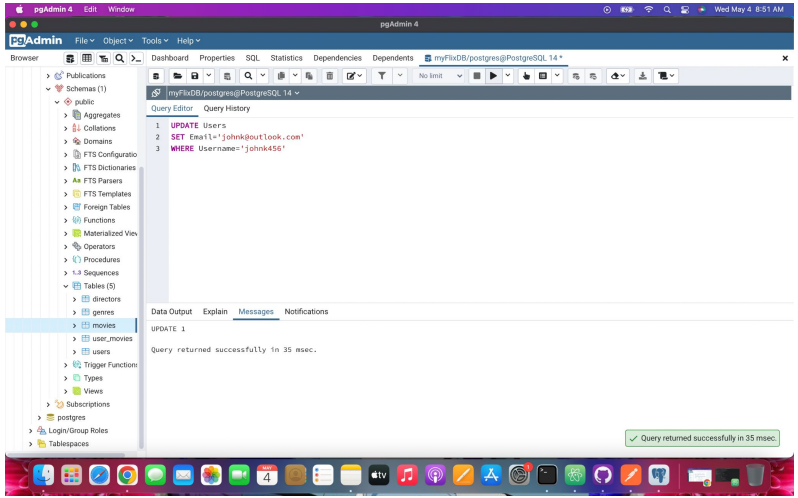
1. This is the basis of any database. “Endpoints” are essentially where final data will be located, and when a user submits a request through the browser, these endpoints allow routing to the data in the database.

2. This step is relatively easy, as it’s the basic skeletal structure of the database. The main challenge was learning several new languages and syntax all at once, including Node, UNIX and LINUX commands, SQL, and Express. Once used together, these became easy to work with.

3. Create database as non-relational using MongoDB

This is where the aforementioned data will be stored. To create this, I started without any API as I would be creating my own. Because of that, I first created my own data as a starting point, creating a relational database using PostgreSQL and the command line interface.
This also involved coming up with a database schema, or what types of data would be stored in the database. For this app’s purpose, the data would be movies and their respective information, including title, genre, synopsis, and director.



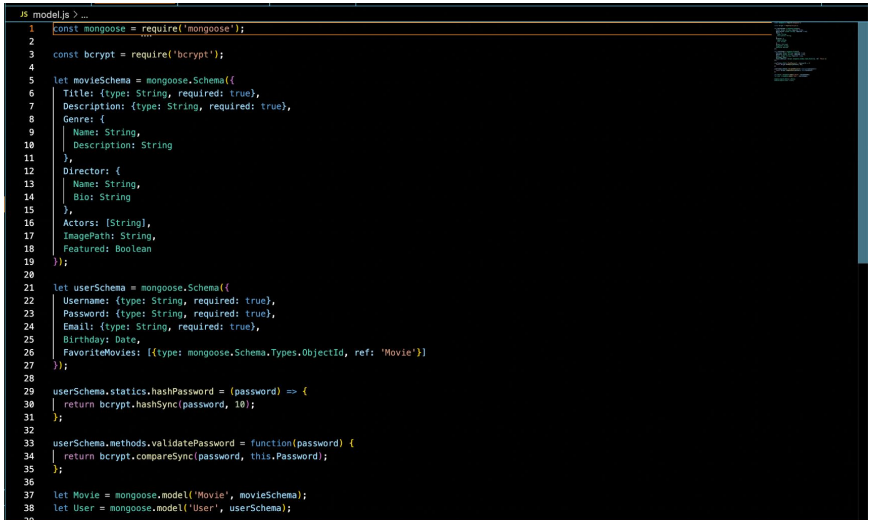


From here the endpoints to data were tested in PostgreSQL, using CRUD (Create, Read, Update, Delete) processes. This was to ensure the endpoints worked correctly and the data could be interacted with.Finally, after local testing and approval, the database was converted from a relational database to non-relational. Relational databases are great for storing data that stays consistent and uses one-to-one, one-to-many, and many-to-one relationships to store data, often using identifiers and keys to relay that data.

With non-relational databases, this is partly true, except that the data can be changed much more easily for each instance of information. The easiest way to describe the difference is that relational databases use tables with records that can relate to each other and contain information, while non-relational (in this case, document-based) data stores documents of data in collections, and each document has an id that can be accessed to modify it. To create this database, the data I created was combined with MongoDB.

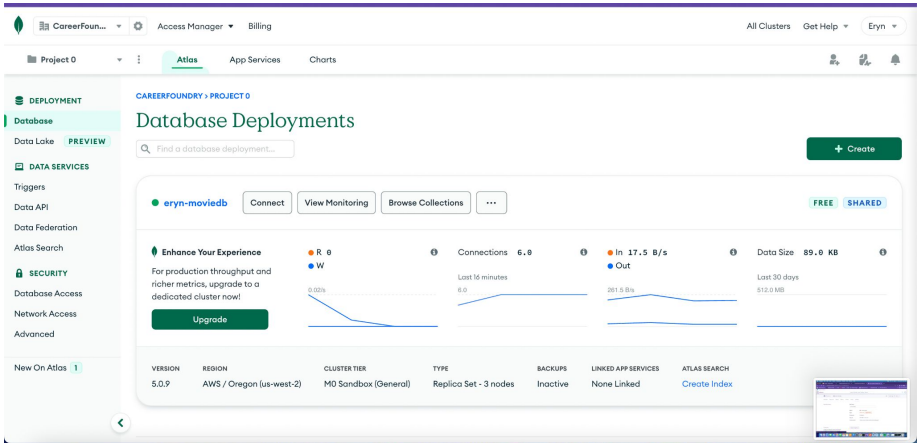
3. Model Business Layer Logic

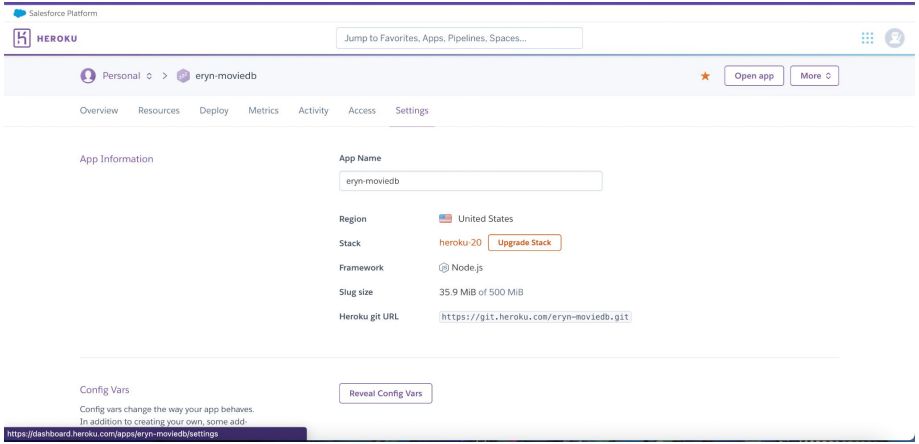
Using Mongoose (a tool for Object Document Mapping) I created models in order for the database to be read by the server and client. Before this, it's essentially only stored data. These models essentially consist of sets of variables representing the data to be processed. If data doesn't match the requirement of the schema, it doesn't pass or get used (later it wouldn't show up on the user interface).
Then this model is referenced as a schema in the index of the database, which is a central part to moving and processing data.



4. Implement Security, Authorization, Data Validation, and host database on Heroku

Finally and very much an important part, security protections had to be put in place. This was done using JWT security and token authorization to secure each (future) user's access.
This security also restricts how much users can manipulate the data. For instance, it would be very bad if they could access the data directly and alter it.
Finally the database needed to be hosted. For this, I used mongoDB Atlas and Heroku. From there the data could be queried remotely and was no longer hosted on my machine.





Creating the Client-Side

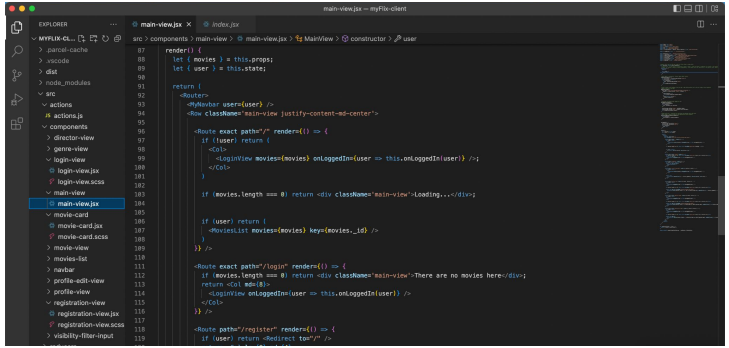
Use React to structure the client-side, as an SPA

From this point, I used React and Parcel (a dependency bundler, via more Node) to set up the base of the client side of the application. This meant that base files were created, along with some basic skeletal code for the root of the project. This is important because it initiates the project with required dependencies, such as the React library/framework.

1. Create components and functions

After the base app was set up, components were created that represent each individual “view” of the application. This is what defined the app as an SPA (Single Page Application). This means that instead of loading an entire new file every time a user navigated to a new page, the app stores all the views in a single file, or page. In the coding side of this, however, it means that each view or function is broken down into individual components, making maintenance and scalability easier down the road. At this point the main/home view is implemented, plus login, registration, profile, and movie info views.

Once the basic code for each view was

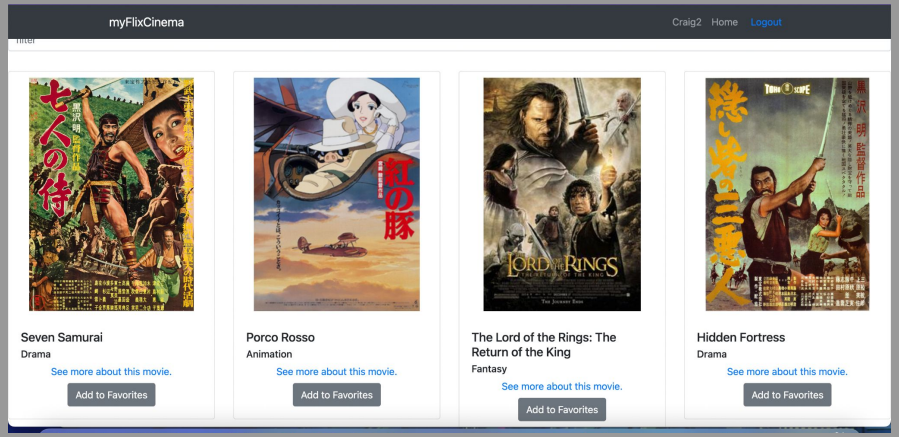


written, I started writing functions that would allow user interaction. Some of these included fetching data from the database so it could be viewed in the user interface, and some basic functions like rendering a message in the interface to ensure that the components were connected, communicating and rendering in the browser. Later, more functions would be added to complete the app.

2. Style and Add Routing

This part is very important - without styling, the user interface would be hard for non-developers to use or understand, or it would simply have no aesthetic value, or both. At this point the app was still in development, but some basic styling was added so that it could be modified alongside the application as it grew. This was done using React Bootstrap, which meant a lot of the styling had already been written for me, and I simply had to import it into my app. After that I could tweak it as I saw fit or as was required by the user stories.

At the same time, some extra routing was added to make sure the client-side of the app (what users would see) would interact with the server side. These routes were set up using axios and CRUD methods.

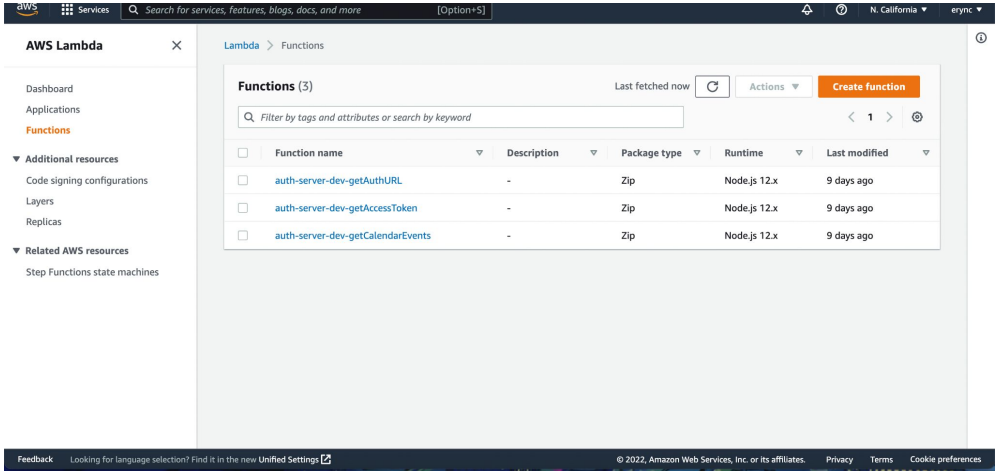


3. Implement Redux

At this point much of the app has been written, and information is rendering as it should. However, I needed to implement what's called a Flux pattern, so that updates to the app would only flow in one direction and not corrupt the whole app by moving backwards or into the wrong data. This was done using Redux, which is a library and pattern for maintaining and updating state, according to its website. To put it simply, it means that when a user inputs something, or data is updated, the "state" of the application will update accordingly and only as I tell it to via the code. For example, when a user logs in, the app will update the user state to hold the user login data so that they can use the app. Otherwise, they might log in, but the credentials could just be sent to the server and nothing would happen on the client side, because that "state" wouldn't have changed. That's where Redux would be implemented.

4. Import Data from Server & Authenticate Users

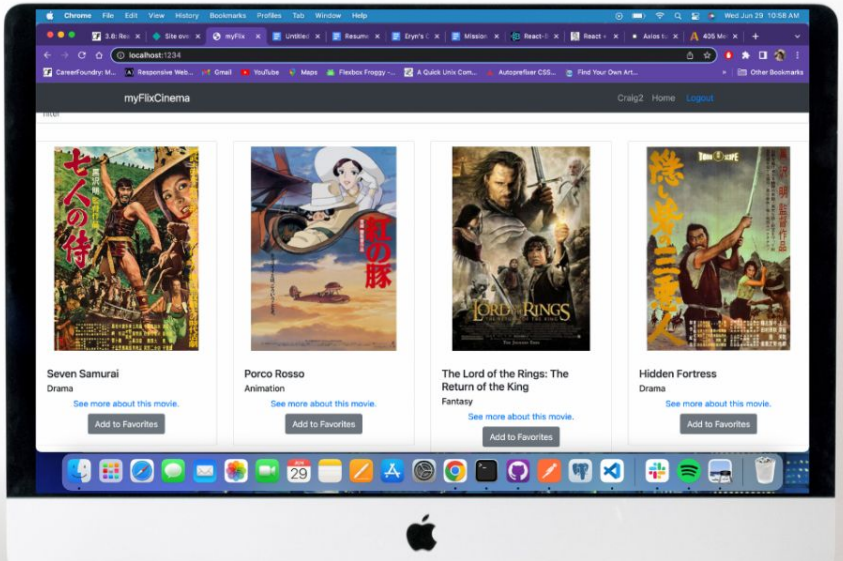
At this point the main thing left to do was actually connect my remote server to my client side. This was done via authorization and security implementation (such as token verification), and via endpoints using AWS server hosting.



5. Host on Local Machine & Netlify

This completed the app, called myFlix! With everything working as it should, I could connect to the app on my local machine as a host, or have it hosted as a website using Netlify. Both options were used, so the app is available for users.

Conclusion


myFlixCinema
Craig2 Home Logout
Seven Samurai
Drama
See more about this movie.
Add to Favorites
Porco Rosso
Animation
See more about this movie.
Add to Favorites
The Lord of the Rings: The Return of the King
Fantasy
See more about this movie.
Add to Favorites
Hidden Fortress
Drama
See more about this movie.
Add to Favorites

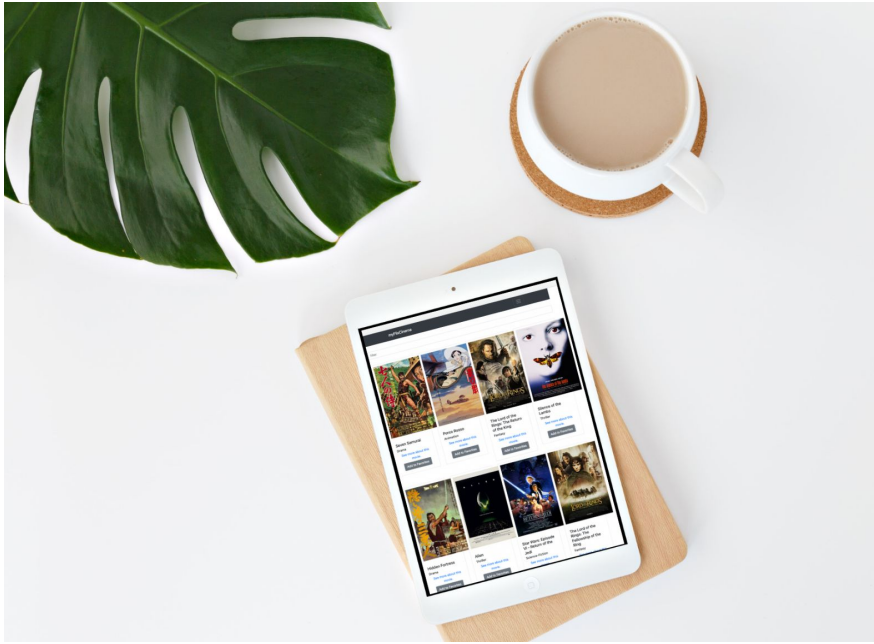
This project was a major challenge for me. It was a foray into unknown territory for me, as I had not used React prior to this, and using libraries and frameworks was also new. However, now that I've done it, I've really cemented the knowledge and have a great understanding of how React works and how flexible it can be when coupled with the right tools, and a good database.

The final results are something I would like to improve, but they did meet all necessary requirements for the project. The main objectives were to create an app using React, compose the server-side from scratch, and unite the two ends to make a complete, SPA, web-based application. The result met these objectives, and in the future I'd like to delve further into Bootstrap and make the app much more organized, expanded, and add more components. The wonderful thing is, this is all possible because I used React and a non-relational database server. I can update as I want or need! I will also be able to strengthen my React knowledge as well as my comprehensive understanding of server-side programming.

Reflections

I could not have completed this project without the help of a great tutor and mentor. Along the way I made mistakes and was forced to debug and fix my own work, which in turn gave me a lot of perspective on the programs I was using to build my own program. I now know how React, Redux, Node, MongoDB, Express, and the libraries I used work, and what they can use to function or not, as well as how to find issues. It was a lot of work, and I'm happy with the result. The only thing I would change is some layout preferences, accessibility, more data, and better design. You can see the complete app here:
To login: user: Test4
Pass: 1234567

https://erynsawesomemyflix.netlify.app/



Contact

Eryn Craig

erynlcraig@gmail.com

https://www.linkedin.com/in/eryncraig/