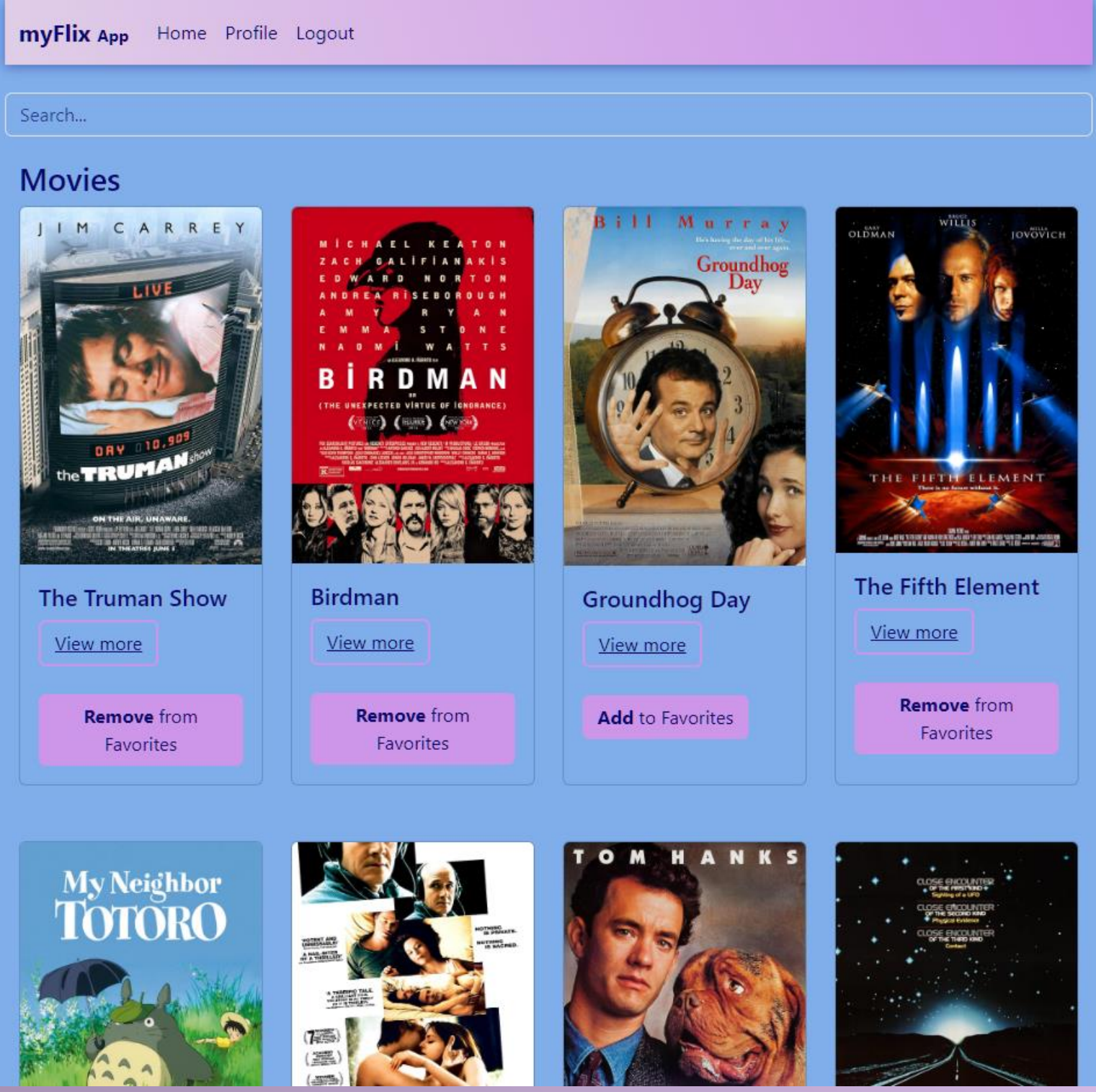

Case Study:

myFlix React App

December 2023

Author: Anastasia Fazius

Case Study: “myFlix” React App

Overview:

This client-side application is crafted to complement the server-side code (REST API (Representational State Transfer Application Programming Interface) and database) developed in the previous project - movie API, thereby forming the integral components of the myFlix web application. This project exemplifies the mastery of full-stack JavaScript technologies within the MERN stack (MongoDB, Express, React, and Node.js) by seamlessly integrating server-side functionality with a dynamic and responsive client-side. Users of my myFlix web app can explore movie details, create profiles, and curate their favorite film collection.

Purpose and Context:

I created this project to explore the development of web applications using the tech stack known as the MERN (MongoDB, Express, React, and Node.js) stack and to showcase my skills by adding it to my portfolio-website. As part of CareerFoundry’s Full-Stack Development program, this project stands as a practical demonstration of my hands-on learning experience and proficiency in React.

Objective:

The main objective of this project was to develop the user-friendly component of myFlix, aligning it with the server-side functionality established in a prior phase. Leveraging the React library, the aim was to design an intuitive user interface capable of seamlessly interacting with the server through specific pathways known as REST API endpoints. These pathways act as bridges, facilitating communication between the client-side (what users see) and the server-side (where data is stored). The result is a polished web application that not only offers users a smooth and enjoyable experience in discovering information about movies but also provides features like filtering the movie list,

accessing detailed information, and managing user profiles along with their favorite movie lists.

Duration:

3 weeks

Credits:

My involvement in the myFlix app extended across the entire development spectrum, from conceptualization to hands-on implementation. Being one of the initial projects in the program, it posed numerous challenges for me as a beginner. The project was navigated through a comprehensive tutorial within the Full Stack Development program, supplemented by independent research, and significantly benefited from the invaluable support and guidance of my tutor, Blaise Bakundukize, and mentor, Stephen Barungi.

Tools, Skills, Methodologies:

The myFlix project showcases proficiency in a diverse set of technologies and tools, including MongoDB for database management, Express for server-side development, React for client-side interfaces, and Node.js as the runtime environment. The application harnesses the power of Parcel as its build tool, ensuring efficient bundling and deployment. React Bootstrap, a versatile UI library based on Bootstrap, enhances the app's styling and responsiveness. For routing, the project utilizes React Router, a powerful library enabling seamless navigation. Throughout the development process, React Redux was employed for state management, enhancing the efficiency of features like movie filtering.

Development of the myFlix React web application and issues to solve in stages:

1. Choosing the Right Framework/Library for the Project

In the realm of web development, selecting the appropriate framework or library is akin to choosing the right set of tools for a construction project. A library, comparable to a box of tools, provides prepackaged code modules that developers can integrate into their projects as needed. Conversely, a framework offers a predefined structure into which developers insert their code, dictating how the application is built - essentially being "opinionated" about development practices.

In the previous project (#2 on my portfolio-website - movie API), the Express framework was employed to construct the server-side of the myFlix app. Express streamlined the development process by simplifying Node.js syntax, facilitating the creation and management of the web server. For the client-side development in this project, the React library is the chosen tool, standing alongside other popular JavaScript frameworks and libraries such as Angular and Vue.js.

While there isn't a one-size-fits-all framework, React brings several advantages to the table. Considering project requirements such as scale, scope, documentation, popularity, testability, and the development environment, React emerges as a robust choice. Its component-based architecture fosters modular development, facilitating scalability and management as the project evolves. The extensive documentation and widespread popularity of React ensures a supportive community, offering readily available solutions and best practices - essential for a project emphasizing documentation, popularity, and testability. The decision to use React aligns with the project's goal of building a client-side interface that complements the existing server-side, showcasing mastery of full-stack JavaScript development within the MERN stack (MongoDB, Express, React, and Node.js).

Now, let's dive into a key aspect of React's magic - reusable components. Instead of redundantly writing code for each movie item, React empowers us to create reusable components, encapsulating specific logic into elements, typically stored in their own files. Take, for example, the self-explanatory MovieCard component, rendering within the MainView. This approach not only keeps our code accessible and efficient but also sets the stage for reusable logic that enhances the maintainability of our project.

In essence, the decision to embrace React resonates with our vision of a dynamic and scalable project, where smart choices in technology contribute to a seamless and efficient development journey.

2. Build Process

Setting Up the React Development Environment

Before diving into React development, it's crucial to prepare several components:

- CLI / Terminal / Shell
- Node.js and Node Package Manager
- Git
- Web browsers
- Code editors

Command Line Interface (CLI), also known as the terminal or shell, is a text-based interface where developers interact with their computer and execute commands. In React development, the CLI is indispensable for various tasks, including project initialization, package management, and running scripts. Node.js and its package manager are essential for React development, while Git plays a key role in version control. Popular web browsers like Chrome or Firefox are recommended, and Visual Studio Code (VSCode) is the chosen code editor for the myFlix project.

Build Tools

During the build process, we employ Build Tools, acting as conductors orchestrating a streamlined process that groups operations into a single step. The myFlix project utilizes Parcel as the chosen build tool. Parcel simplifies the build process by handling tasks such as transpilation, bundling, and live reloading. It offers minimal configuration, fast bundle times, and compatibility with various file types.

Creating a New Repository

To kickstart the project, we create a repository on GitHub and give it a meaningful name. Setting up version control involves creating a new repository using GitHub Desktop. We initialize the "package.json" file with npm and configure a ".gitignore" file to manage dependencies and project files efficiently. Dependencies are what your app needs for both development and production, while development dependencies are only necessary during development.

Configuring Parcel for the myFlix App

Parcel, our chosen build tool, undertakes the task of transpiling and bundling the myFlix app code. Its configuration involves the utilization of Babel for transpiling JSX (JavaScript XML) and SCSS (Sassy CSS) files. Here's a quick breakdown of terms:
JSX: A syntax extension for JavaScript, JSX allows developers to write HTML elements and components in a JavaScript file. It's a concise and readable way to describe the structure of user interfaces.
SCSS: An extension of CSS, SCSS introduces features like variables, nesting, and mixins, enhancing the maintainability and readability of stylesheets.
ReactDOM: In React development, ReactDOM is the library responsible for rendering React components into the DOM (Document Object Model). It acts as the bridge between React components and the underlying structure of the web page.
In summary, the build process is integral to web development, ensuring optimized code for browsers. Tools like Babel and Parcel streamline operations, enhancing organization, readability, and maintenance of code in projects like myFlix. Additionally, Parcel's live reloading feature plays a crucial role, refreshing your app in the browser with each code change for a smoother development experience.

3. Creating Components

In building the myFlix app with React, our development process revolves around various views adhering to specific functional and technical requirements. A "view" here refers to the presentation of content on the frontend, typically comprising an interactive graphical interface, and optionally, charts, tables, forms, or other UI components. This visible layer is what users see in their browser when interacting with the app.
The frontend of our app comprises a main view that seamlessly transitions between the following subviews:
Login View
Signup (User Registration) View
Movie View
Profile View
Additionally, we've identified specific components that contribute to these views:
A Movie Card
A Movies Filter
A Movies List
A Navigation Bar

Each of these views and components will be treated as a separate entity in React, allowing for modularity and reusability - a fundamental principle in React development.
We've adopted a Single-Page Application (SPA) approach for myFlix, consolidating all UI parts and components within a single page. React components serve as the building blocks of our UI, encapsulating logic and styling for individual pieces. Similar to JavaScript functions, components are independent and reusable, promoting a scalable and maintainable codebase. Components can also contain other components, enabling the creation of intricate design and interaction elements.
In the "myFlix-client" directory, we begin the component creation process in React. The initial component, called MainView, acts as a container for other components. To achieve this, open the newly created "main-view.jsx" file and start writing the code.
The MainView component, in essence, encapsulates the overall structure of the app. A function is assigned to MainView, defining the visual representation of the component - essentially, what will be displayed on the screen. Within this function, we use JSX, a syntax similar to HTML, facilitating the creation of UI elements.
A couple of fundamental concepts to grasp when creating React components include:

1. A Component Can Only Have One Root Element: It's crucial to adhere to the rule that a component should have a single root element when returning a chunk of UI. This ensures clarity and consistency in the structure of React components.

2. JSX Syntax: JSX resembles HTML but is tailored for use in React components. It facilitates the creation of UI elements with a syntax that closely resembles standard HTML, enhancing code readability and maintainability.

As the myFlix app takes shape, the creation of components using JSX forms the cornerstone of a modular, scalable, and maintainable frontend architecture. This approach not only aligns with React best practices but also lays the foundation for a user-friendly and responsive single-page application.

4. Loading Data from an API

In the progression of our myFlix React App project, a pivotal phase unfolded as we delved into the intricate process of loading real data from an API. This marked a crucial step forward, transforming our app from a static showcase to a dynamic, data-driven experience.
The integration of real data from an API was essential to breathe life into our myFlix app. Simulated data provided a foundation, but real-time content enriches the user

experience, offering authentic and constantly evolving information. This step aligned our project with industry standards, fostering a genuine and engaging interaction for users exploring movies, profiles, and more.

The primary goal was to connect our React app with the backend API, seamlessly fetching and rendering actual data within our components. This not only enhanced the authenticity of our myFlix app but also set the stage for features like personalized user recommendations, dynamic movie updates, and more.

To achieve this goal the following principles and tools were used:

React Hooks: Utilized to tap into the component lifecycle with functional components.

Async Operations: Employed for seamless data fetching from the API.

PropTypes: Ensured data type validation for robust and error-resistant components.

Handling asynchronous API calls required careful consideration. The use of async/await helped manage the complexity, ensuring that data was fetched and processed before rendering components. Implementing PropTypes was initially challenging due to its declarative nature. Extensive documentation review and hands-on experimentation resolved this, ensuring data integrity within components.

Choosing Hooks over class components streamlined our code and embraced the functional paradigm, aligning with modern React best practices. Opting for async/await for API calls facilitated a more readable and maintainable code structure, enhancing the overall development experience.

Loading data from an API was not just a technical step; it was a transformative process that elevated our myFlix app. Real-time content now courses through its veins, setting the stage for a dynamic, user-centric experience. This stage not only met its immediate goal but laid the foundation for the continued evolution of our project. The journey of myFlix continues, fueled by the synergy of React and dynamic API data.

5. Client-Side App Routing: Navigating the Pathways of myFlix

In this segment, we delve into the art of client-side app routing, exploring how to seamlessly guide myFlix users through the various views within our single-page application (SPA). The focus here is on implementing state-based routing, enabling users to navigate effortlessly while maintaining unique URLs for each view - a crucial feature for user-friendly interactions.

The Need for Routing:

As we steer the myFlix project toward greater complexity, the need for efficient navigation becomes apparent. To address this, we turn to routing, a mechanism that

allows us to store state information in the URL, facilitating seamless movement between different sections of our SPA.

Adding a State-Based Router:

Implementing a router from scratch, particularly a state-based router, can be challenging. However, the solution lies in leveraging powerful libraries, and for React apps, React Router stands out as the go-to choice. Our first step involves integrating React Router into the project by adding it to the dependencies section of our package.json file.

State-Based Routing & Navigation:

The primary objective of this phase is to establish state-based routing for our React SPA. This enables users to not only navigate effortlessly between distinct views but also obtain unique URLs for each, making it easy to share specific content. The implementation involves installing React Router packages, refactoring the MainView rendering, and creating an intuitive navigation bar. Unauthenticated users find links to the Login/Signup page, while authenticated users enjoy easy access to Home (MainView) and Profile links. This phase not only enhances the user experience by enabling fluid navigation but also opens the door for future feature expansions. As we guide myFlix users along these digital pathways, we pave the way for a more interactive, user-centric, and seamlessly connected SPA.

6. Leveraging Redux for Predictable State Management

In the unfolding chapters of myFlix, a pivotal phase emerged - the integration of a movie list filter on the MainView. To navigate the app's growing complexity, I turned to Redux, a stalwart library rooted in the Flux design pattern. Redux became the cornerstone of our state management, instilling order and predictability crucial for a dynamic codebase. In my Redux journey, I began by installing Redux and Redux Toolkit, laying the foundation for efficient state management. Slicing the state into reducers - movies and filters - allowed for a focused and organized approach to managing different facets of the app's state. Defining actions and action creators within these slices provided the tools for controlled and predictable state updates. Configuring the Redux store centralized state management, serving as a single source of truth for the entire application. Modifying components to seamlessly interact with the store using useSelector() and useDispatch() enhanced their responsiveness to dynamic state changes.

Bringing the filter to life involved introducing the movies-filter component, simplifying dynamic modifications to the movies filter's input field content. Configuring the

component to dispatch actions set the stage for real-time filter updates. The birth of the movies-list component marked a turning point, taking charge of rendering the movies-filter component and the MainView with elegance. By tapping into the filter and movies state from the Redux store, I solidified the integration of Redux into this feature.
As Redux seamlessly wove into the fabric of myFlix, it emerged as the linchpin of our state management strategy. Get ready to experience the fluid integration of Redux, bringing forth a robust filter mechanism that enhances the user experience on the MainView. The journey of myFlix continues, guided by the power of Redux.

7. Hosting the App

As I aimed to make myFlix accessible to a broader audience, the next logical step was to host the app on Netlify. Here's a straightforward guide on how I smoothly transitioned my locally developed myFlix React App into a globally accessible web application.
Firstly, I navigated to my project folder and created a new file named netlify.toml to handle smooth redirection to the root path. This setup is crucial for Netlify's deployment.
To ensure Netlify could handle the deployment seamlessly, I installed Parcel as a local developer dependency. It was essential to match the installed version with the one specified for @parcel/transformer- sass in my package.json under "devDependencies."
This local installation enabled Netlify to execute the Parcel build command on its hosting server during deployment.
After making changes to my myFlix app, I committed and pushed them to the GitHub repository. This step ensured that Netlify could access the latest version of my project.
Upon heading to Netlify and signing up, I utilized the "Import from Git" feature to connect my GitHub account. I selected the repository I wanted to deploy (in this case, my "my-flix client" repo) and chose the main branch.
Configuring basic build settings involved specifying the build command, base directory, and publish directory. I also took a moment to personalize the default site name in Site Settings.
To prevent CORS-related errors, I ensured my site's URL was added to CORS' allowed origins. Adjusting CORS configurations in my myFlix API project was crucial to achieving this. CORS, or Cross-Origin Resource Sharing, is a security feature implemented by web browsers to control how web pages in one domain can request and interact with resources hosted on another domain. In simpler terms, it's a set of rules that dictate whether a web application running at one origin (e.g., domain, protocol, or port) is allowed to make requests for resources from a different origin.

With these steps, myFlix smoothly transitioned from a local development environment to a globally accessible web application hosted on Netlify. This deployment not only expanded the reach of my project but also set the stage for a seamless user experience across diverse geographical locations.

Project Reflection:

In conclusion, the development journey of the myFlix React web application has been a transformative experience, showcasing the seamless integration of full-stack JavaScript technologies within the MERN stack. The original objective, to create a user-friendly client-side interface that complements the server-side functionality, was achieved. Users can now effortlessly explore movie details, create profiles, and curate their favorite film collections, all within an intuitive and dynamic application.

One of the most challenging yet rewarding aspects of the project was the implementation of Redux for state management. Navigating the complexities of state in a growing codebase required careful consideration, and Redux emerged as a powerful solution, bringing order and predictability to the app's state.

Reflecting on this project, I've gained valuable insights into the nuances of building scalable and maintainable web applications. From choosing the right technology stack to implementing features like client-side routing and state management, this journey has equipped me with a robust skill set.

This project has not only solidified my proficiency in the MERN stack but has also instilled a deeper understanding of the collaborative synergy between client-side and server-side development. As I move forward, I plan to apply the lessons learned from myFlix to future projects, ensuring a user-centric and technically sound approach.

In summary, the myFlix React web application stands as a testament to the power of thoughtful design, meticulous development, and continuous learning. It has been a rewarding journey, and I look forward to further refining and expanding my skills in the dynamic field of full-stack web development.