# **Project Documentation: Personalized Content Dashboard**

Author: Yeshwanth Krishna

Date: July 21, 2025

The Github repo is available at (<https://github.com/Yeshwanth-kr/personalized-content-dashboard>)

The Live app is available at (<https://personalized-content-dashboard-yeshwanthkr.vercel.app/>)

## **1. Introduction & Project Overview**

This document provides a comprehensive overview of the "Personalized Content Dashboard" project, developed as a part of the SDE Intern - Frontend Development assignment. The primary goal of this project was to build a modern, interactive, and user-centric platform where users can consume and organize personalized content from various sources.

The application is built as a Single Page Application (SPA) using a robust and modern frontend stack, focusing on performance, scalability, and user experience. It features a dynamic feed of news and movie recommendations, a responsive dashboard layout, and advanced UI features like dark mode and drag-and-drop organization.

## **2. Core Architecture & Technical Decisions**

The application is built on a foundation of industry-standard tools and best practices. The choice of each technology was deliberate, aiming to meet the project's requirements efficiently.

* **Framework: Next.js (with App Router)**
  + **Reasoning:** Next.js was chosen for its powerful features, including server-side rendering capabilities, file-based routing (via the App Router), and built-in optimizations like the <Image /> component. The App Router, in particular, allows for modern React features like Server Components and a clear, co-located structure for components, pages, and API routes.
* **Language: TypeScript**
  + **Reasoning:** TypeScript was used to ensure type safety across the entire application. This significantly reduces runtime errors, improves code quality, and makes the codebase easier to maintain and refactor, especially as complexity grows.
* **State Management: Redux Toolkit**
  + **Reasoning:** Redux Toolkit was selected for managing global application state. It provides a standardized and efficient way to handle complex state logic, such as user preferences and fetched API data. Its createSlice and createAsyncThunk APIs simplify reducer logic and asynchronous actions, making state management predictable and boilerplate-free.
* **Styling: Tailwind CSS**
  + **Reasoning:** A utility-first CSS framework, Tailwind CSS was used for its rapid UI development capabilities. It allowed for the creation of a custom, responsive design without writing custom CSS. Its dark: variant was essential for implementing the dark mode feature efficiently.
* **API Data Fetching: Axios & Next.js API Routes**
  + **Reasoning:** Axios was used for making HTTP requests due to its simple and intuitive API. To handle CORS restrictions and protect API keys, server-side proxies were created using **Next.js API Routes**. This is a critical architectural decision for production environments, as it prevents direct browser-to-API communication, which is often blocked by free-tier API plans.

## **3. Core Features Implementation**

### **3.1. Personalized Content Feed & API Integration**

The core of the application is the unified content feed.

* **Data Fetching:** The fetchAllContent async thunk in the contentSlice orchestrates fetching data from multiple sources concurrently using Promise.all.
* **API Proxies:** To circumvent CORS errors and API limitations in production, dedicated API routes (/api/news and /api/recommendations) were created. The frontend calls these local endpoints, which then securely fetch data from the external News and OMDb APIs on the server.
* **Data Normalization:** Upon receiving data, a normalization step was implemented within the Redux slice. This ensures that every content item, regardless of its source, is mapped to a consistent ContentItem type with a unique, string-based id. This was crucial for fixing bugs related to the "Favorites" feature and ensuring data consistency.

### **3.2. Dashboard Layout & State Management**

* **Component Structure:** The UI is broken down into modular components (Sidebar, Header, ContentCard, etc.) for reusability and maintainability.
* **View Management:** The main page (/src/app/page.tsx) acts as a client component that manages the currently active view (e.g., "My Feed," "Favorites") using local state (useState). This state determines which section component is rendered.
* **Favorites Logic:** The favoritesSlice manages an array of favorite items. The toggleFavorite reducer handles both adding and removing items, and the logic is dispatched from the ContentCard component.

### **3.3. Search Functionality**

* **Debouncing:** To optimize performance, a custom useDebounce hook was implemented. This hook delays the update of the search term in the Redux store, preventing re-renders and filtering on every single keystroke. The search term is stored globally in the contentSlice so that the PersonalizedFeed can filter its content accordingly.

### **3.4. Advanced UI/UX Features**

* **Drag-and-Drop:** Implemented using react-dnd. The PersonalizedFeed component maintains the order of cards in a local state. The DraggableContentCard wraps the standard ContentCard and uses useDrag and useDrop hooks to manage the reordering logic.
* **Dark Mode:** A uiSlice in Redux manages the isDarkMode state. A ThemeManager component reads this state and dynamically adds or removes the .dark class from the root <html> element, allowing Tailwind's dark: variants to apply the correct styles.
* **Animations:** Framer Motion was used to add a loading spinner and smooth, fading transitions between the "Feed" and "Favorites" views, enhancing the user experience.

## **4. Testing Strategy**

A multi-layered testing strategy was implemented to ensure application quality and reliability.

* **Unit Testing (Jest):** Focused on isolated logic. The useDebounce hook was tested to verify its timing and value updates.
* **Integration Testing (Jest & React Testing Library):** Focused on component interactions. The FavoritesSection was tested to ensure it correctly renders based on the Redux store's state, specifically handling the empty state edge case.
* **End-to-End Testing (Cypress):** Focused on complete user flows. Critical paths like searching for content and adding/removing items from favorites were tested to ensure the application works as expected from a user's perspective.

## **5. Challenges & Solutions**

* **Challenge:** CORS errors and API limitations in the production environment.
  + **Solution:** Implemented server-side proxies using Next.js API Routes. This shifted the responsibility of fetching data from the client to the server, bypassing browser restrictions and protecting API keys.
* **Challenge:** Inconsistent data structures and IDs from different APIs.
  + **Solution:** Implemented a data normalization step within the Redux slice immediately after fetching. This ensured all data conformed to a single, predictable ContentItem type, which resolved several UI bugs related to the "Favorites" feature.
* **Challenge:** Jest configuration for a Next.js TypeScript project.
  + **Solution:** Created jest.config.mjs and babel.config.js to correctly configure the test environment, enabling the use of the jsdom environment and proper transformation of TypeScript and ES module syntax.

## **6. Conclusion**

The Personalized Content Dashboard project successfully meets all the requirements outlined in the assignment. It demonstrates a strong understanding of modern frontend architecture, state management with Redux, API handling, complex UI implementation, and comprehensive testing practices. The final application is performant, user-friendly, and built on a scalable and maintainable codebase.