

Project Documentation: SB News

1. Introduction

This document provides comprehensive technical and functional documentation for the **SB News** project, a dynamic, modern web application designed for news aggregation and display. The platform serves as a central hub for users to consume news from various categories, offering a clean, intuitive, and highly responsive user interface.

- **Project Title:** SB News
- **Team Members:**
 - **Sabarivasan**
 - **Pradeep**
 - **Sriharan**
 - **Sathish**

2. Project Overview

Purpose and Goals

The primary goal of the SB News project is to create a fast, reliable, and user-friendly news portal using modern web development practices. The application's purpose is to aggregate news articles from a third-party API and present them in a visually appealing and easily navigable format.

We aimed to demonstrate proficiency in:

- Building a scalable and maintainable component-based architecture with React.
- Implementing efficient state management for data flow across the application.
- Creating a responsive design that adapts to various screen sizes.
- Integrating with an external API to handle asynchronous data fetching.

Key Features

The application is built with the following core functionalities:

- **Dynamic Homepage:** The landing page dynamically displays featured articles in a prominent hero section and organizes the remaining content into "Top Stories" and "Business" sections. This layout ensures a fresh and engaging experience on every visit.
 - **Categorized Navigation:** The header includes a navigation bar with links to specific news categories: **General, Technology, Politics, Health, and Art & Culture**. Clicking on these links renders a dedicated page showing articles filtered by the selected category.
 - **Search Functionality:** A search icon in the header expands into a search bar, allowing users to find specific news articles by keyword. The search results are rendered dynamically on the same page.
 - **Mobile-First Design:** The entire application is built with responsiveness in mind. The layout, components, and navigation fluidly adjust to provide an optimal viewing experience on mobile phones, tablets, and desktops.
 - **Lazy Loading Images:** Images are loaded lazily to improve initial page load performance, ensuring that the application remains fast even on slower network connections.
-

3. Architecture

The application is structured as a single-page application (SPA) using a component-based architecture. This approach enhances code reusability, simplifies maintenance, and improves development efficiency.

Component Structure

The application's UI is broken down into a hierarchy of components. The following diagram illustrates the primary component structure:



- **App.js**: The root component that wraps the entire application, handling global layout and routing.
- **Header.js**: A static component that contains the navigation and search functionality. It receives no props but manages its own internal state for the search bar.
- **HomePage.js**: A container component responsible for fetching the main news data and rendering the **FeaturedSection** and multiple **ArticleGrid** components.
- **ArticleGrid.js**: A reusable component that takes an array of articles as a prop and renders a grid of **NewsCard** components.
- **NewsCard.js**: The most granular reusable component, responsible for displaying a single article's image, title, and link.

State Management

The application employs React's built-in **Context API** for global state management. This avoids "prop drilling" and makes the application's data flow more predictable.

- **Global State:** A `NewsContext` is created to manage all news-related data. The `NewsProvider` component wraps the entire application (`App.js`), making the following state accessible to any nested component:
 - `newsData`: The primary array of news articles.
 - `isLoading`: A boolean to indicate whether data is currently being fetched from the API.
 - `error`: A string to store any error messages from the API.
- **Local State:** For component-specific UI state, the `useState` hook is used. Examples include:
 - The `Header` component uses local state to toggle the visibility of the search input field.
 - A component like a filter dropdown (if implemented) would use local state to track its open/closed status.

Routing

The application uses **React Router** for declarative navigation. The routes are defined in `App.js` and link the UI to specific components based on the URL.

- `/`: Renders the `HomePage` component.
 - `/:category`: A dynamic route that renders the `CategoryPage` component. The `:category` URL parameter is used to fetch and display news from the corresponding category (e.g., `localhost:3000/technology`).
 - `*`: A fallback route that renders a `NotFoundPage` component, providing a user-friendly error page for invalid URLs.
-

4. Setup Instructions

This guide assumes you have **Node.js** (v18+) and **npm** or **yarn** installed on your system.

1. Clone the Repository:

2. Bash

```
git clone https://github.com/your-username/sb-news-project.git
cd sb-news-project
```

3.

4.

5. Install Dependencies:

Navigate into the frontend directory and install the required packages.

6. Bash

```
cd client
```

```
npm install
```

7.

8.

9. Configure Environment Variables:

The application requires an API key from a news data provider (e.g., NewsAPI.org).

- Create a file named `.env` in the `client` directory.
- Add your API key to this file in the following format:
- Code snippet

```
REACT_APP_NEWS_API_KEY=your_api_key_here
```

○

○

- **Note:** You must sign up for a free developer account to get your key.
-

5. Folder Structure

The project's frontend is organized to be highly modular and easy to navigate.

```
client/
├── node_modules/
├── public/
├── src/
│   ├── assets/
│   │   ├── images/ (Logo, placeholder images)
│   │   └── fonts/
│   ├── components/ (Reusable UI elements)
│   │   ├── Header/
│   │   │   ├── Header.js
│   │   │   └── Header.css
│   │   ├── NewsCard/
│   │   │   ├── NewsCard.js
│   │   │   └── NewsCard.css
│   │   └── Loader.js
│   ├── context/
│   │   └── NewsContext.js (Context API setup)
│   ├── pages/
│   │   ├── HomePage.js
│   │   ├── CategoryPage.js
│   │   └── NotFoundPage.js
│   ├── hooks/
│   │   └── useFetchNews.js (Custom hook for data fetching)
│   ├── services/
│   │   └── newsApi.js (API call logic)
│   ├── styles/
│   │   └── global.css
│   ├── App.js
│   └── index.js
├── .env
├── package.json
└── README.md
```

- **components/**: Houses all reusable and presentational components. Each major component has its own subdirectory for organization.
 - **pages/**: Contains the top-level components that correspond to specific routes.
 - **context/**: The `NewsContext.js` file, which sets up the provider and consumer for global state.
 - **hooks/**: Custom hooks to encapsulate reusable logic, such as data fetching or state management. The `useFetchNews` hook is a key utility that handles the API call, loading state, and error handling.
 - **services/**: The `newsApi.js` file contains a dedicated function for interacting with the news API, keeping the API logic separate from the components.
-

6. Running the Application

Follow these simple steps to start the development server:

1. Ensure you are in the `client` directory.
2. Run the following command in your terminal:
3. Bash

`npm start`

- 4.
 - 5.
 6. The application will automatically open in your default browser at `http://localhost:3000`. The server will also automatically reload the page whenever you make changes to the code.
-

7. Component Documentation

Key Components

- **HomePage.js:**
 - **Purpose:** Serves as the landing page, displaying a curated selection of news. It fetches the initial data from the API upon mounting and passes it down to child components.
 - **Data Flow:** Uses the `useContext(NewsContext)` hook to access the global `newsData` state.
 - **Render Logic:** Renders the `FeaturedSection` with a single prominent article and two `ArticleGrid` components for the "Top Stories" and "Business" sections.
- **Header.js:**
 - **Purpose:** The main navigation component, present on every page. It includes the logo, navigation links, and a search icon.
 - **State:** Manages the local state of the search input and a boolean to control the visibility of the search bar.
 - **Interactions:** Handles click events for the search icon to toggle the search input's visibility and for the navigation links to trigger a route change.

Reusable Components

- **NewsCard.js:**
 - **Purpose:** A highly reusable, presentational component for displaying a single news article. It is designed to be self-contained and only relies on the props passed to it.
 - **Props:**
 - `article` (object, required): An object containing all the details of the news article.
 - `key` (string, required): A unique identifier for the component.
 - **Example Usage:**
 - JavaScript

```
<NewsCard
  article={{
    title: "Sorry Elon: Chinese Company Overtakes Tesla...",
    urlToImage: "...",
    url: "...",
    description: "...
  }}
/>
```

-
-
- **Render Logic:** Uses the provided `article` object to dynamically render the image, title, and a link.

8. State Management

Global State

The global state is managed by a single `NewsContext` object, which is created in `src/context/NewsContext.js`. This context is responsible for:

- Storing the master list of news articles.
- Tracking the loading state to show a loader while data is being fetched.
- Holding any API-related error messages.

The `NewsProvider` component, which wraps `App.js`, fetches the initial data and updates this global state, ensuring that all components have access to the same, up-to-date information without having to make their own API calls.

Local State

Local state is reserved for UI-specific data that doesn't need to be shared across the entire application.

- **Example:** In a search component, `useState` is used to manage the `query` string and the `isSearchOpen` boolean.
 - **Example:** A simple dropdown menu for sorting articles would use local state to track which option is currently selected.
-

9. User Interface

The UI follows a clean, minimalist design with a focus on readability and accessibility.

- **Color Palette:** The primary color is a pure white background, complemented by deep black for text. Accent colors are used sparingly for links and interactive elements to guide the user's eye.
- **Typography:** The application uses a sans-serif font family for a modern, clean look. Headers are large and bold, while body text is a comfortable reading size.
- **Hero Section:** The screenshot highlights the hero section at the top of the homepage, which features a single prominent article with a large image and headline, followed by a grid of smaller, related articles.
- **Responsiveness:** On smaller screens, the horizontal navigation bar collapses into a hamburger menu, and the article grids stack vertically for easier viewing.

10. Styling

CSS Frameworks/Libraries

The project uses **Tailwind CSS**, a utility-first CSS framework. This choice was made to accelerate the styling process and enforce a consistent design system. Instead of writing custom CSS, developers can use a series of utility classes to style components directly in the JSX.

- **Common Classes:** The application extensively uses classes like `flex`, `grid`, `p-4` (padding), `shadow-lg`, and responsive prefixes like `md:flex` to define the layout and styling.

Theming

The application currently implements a single, light theme. There is no built-in mechanism for users to switch to a dark theme. The color palette is defined in the `tailwind.config.js` file, allowing for easy updates to the overall theme in the future.

11. Testing

Testing Strategy

A robust testing strategy ensures the application is reliable and free of regressions.

- **Unit Testing (Jest & React Testing Library):** Individual components and utility functions are tested in isolation. We focus on ensuring:
 - Components render without crashing.
 - Props are passed and rendered correctly.
 - User interactions (e.g., clicks) trigger the correct state changes or function calls.
 - Hooks and utility functions return the expected values.
- **Integration Testing:** We test how components work together. For instance, we would test that the `HomePage` correctly renders the `ArticleGrid` with the data it receives from the global state.
- **End-to-End (E2E) Testing (Cypress):** We use Cypress to simulate full user journeys through the application. A typical E2E test would involve:
 - Visiting the homepage (`/`).
 - Clicking on the "Technology" navigation link.
 - Verifying that the URL changes to `/technology`.
 - Asserting that the articles displayed on the page are relevant to the "Technology" category.

Code Coverage

We use Jest's built-in coverage reports to track test coverage. The goal is to maintain a minimum of 85% coverage for all critical components and business logic.

12. Screenshots or Demo

- **Homepage:** The screenshot provided showcases the main layout, including the hero section, navigation, and article grids.
- **Mobile View:** [Please include a screenshot of the mobile-optimized layout here]
- **Category Page:** [Please include a screenshot showing the articles filtered by a specific category, like "Technology"]
- **Live Demo:** [Link to a live demo, e.g., a Vercel or Netlify deployment]

13. Known Issues

- **API Rate Limit:** The free tier of the news API has a strict rate limit. The application may cease fetching new data if the daily limit is exceeded.
- **No Pagination:** Currently, the application only displays a limited number of articles (e.g., 20) per category. There is no pagination to load more articles, which is a key area for future improvement.
- **Basic Search:** The search functionality is a basic keyword match. It does not support advanced queries or filters.

14. Future Enhancements

- **User Authentication and Profiles:** Implement user registration and login functionality. This would enable personalized features.
- **Personalized Feeds:** Allow users to select their favorite topics and news sources to create a customized feed.
- **Dark Mode:** Add a toggle button for a dark theme to improve user experience, particularly in low-light environments.
- **Backend Integration:** Transition from a public API to a custom backend to manage content, user data, and overcome API rate limits.
- **Advanced Search and Filters:** Implement more robust search functionality with filters for source, date, and language.