Project Documentation

Overview

The Getir To-Do App is a full-stack web application developed to manage a list of tasks. It allows users to perform CRUD operations without the need for authentication, aligning with the project's requirement for a single, accessible to-do list. The application focuses on responsive design, user experience, and code maintainability.

Technologies Used

Frontend

- **React.js:** For building the user interface with reusable components.
- **Redux Toolkit:** For efficient state management with simplified Redux logic.
- **Material-UI (MUI):** For consistent and responsive UI components.
- **TypeScript:** To add static typing for better code quality and maintainability.
- **Formik and Yup:** For robust form handling and validation.
- **Axios:** For centralized HTTP requests to the backend API.
- **Framer Motion:** For adding animations to enhance user experience.

Backend

- **Node.js and Express.js:** For creating the RESTful API.
- **MongoDB and Mongoose:** For data storage and object modeling.
- **CORS:** To handle Cross-Origin Resource Sharing between frontend and backend.

Testing

- **Jest and React Testing Library:** For unit testing React components.
- **Cypress:** For end-to-end testing to simulate user interactions.

Deployment

- **Render.com:** For hosting both the frontend and backend applications.

Design Choices and Implementation

1. Responsive Design with Material-UI

- **Utilizing MUI's Grid System and Breakpoints:**
    - Employed MUI's Grid system and breakpoint properties to create a responsive layout that adapts to various screen sizes.
    - Ensured that components rearrange and resize appropriately on mobile devices, tablets, and desktops.
- **Styled Components with MUI:**
    - Leveraged MUI's styled API to create custom styled components, mimicking the popular styled-components library.
    - This approach allows for writing CSS-in-JS with access to the theme object, enabling consistent styling across the application.

2. State Management with Redux Toolkit

- **Simplified Redux Logic:**
    - Used Redux Toolkit to streamline Redux setup with less boilerplate.
    - Employed createSlice and createAsyncThunk for creating slices and handling asynchronous actions, respectively.
- **Slices for Task Management:**
    - Created a tasksSlice to handle task-related state and asynchronous operations like fetching, adding, updating, and deleting tasks.

3. Environment Variables and Configuration

- **Environment Management:**
    - Utilized environment variables to switch between development and production API endpoints.
- **Note:**
    - In a production environment, .env files should be excluded from version control, and environment variables should be managed securely. In this case I've included .env files in the repository for simplicity in the project.

4. Centralized Axios Instance

- **Why Centralize Axios:**
    - Configured a centralized Axios instance (api.ts) with a base URL to standardize API communication.
    - Facilitates easier maintenance and scalability, allowing for centralized error handling and the addition of common headers, such as JWT tokens for authentication, if needed in the future.

5. Notification Context

- **Why Use Context over Redux:**
    - Implemented a NotificationContext using React Context API to handle notifications globally.
    - Chose Context over Redux for this feature because notifications are a cross-cutting concern that doesn't need to be part of the global Redux store.
    - Using Context avoids unnecessary complexity and boilerplate associated with Redux for a feature that doesn't require state persistence or time-travel debugging.

6. Use of Hooks

- **useEffect and Dependency Management:**
    - Addressed potential infinite loops in useEffect by managing dependencies and utilizing useCallback to memoize functions.
    - By memoizing showNotification from the dependency array, prevented unnecessary re-renders and potential infinite loops.

7. Testing

- **Unit Testing with Jest and React Testing Library:**
    - Wrote simple unit tests for components to ensure they behave as expected.
    - Created custom test utilities to include necessary providers and contexts in tests.
- **End-to-End Testing with Cypress:**
    - Developed a very basic Cypress test to simulate adding a new task.

8. Deployment

- **Render.com for Hosting:**
    - Selected Render for deployment due to its ability to host both static sites and web services.

    - Streamlined the deployment process by integrating with GitHub repositories.
- **Environment Configuration:**
    - Set up environment variables on Render to ensure the frontend communicates with the deployed backend API.
- **The application is deployed and accessible at:**
    - Frontend URL: https://getir-todo-frontend.onrender.com
    - Backend URL: https://getir-todo-app.onrender.com/api/tasks

Notable Features and Enhancements

Responsive Design

- **Device Adaptability:**
    - Designed the web application to provide an optimal user experience across various devices, ensuring responsiveness without a strict mobile-first approach.
- **Dynamic Adjustments:**
    - Utilized MUI's responsive utilities to adapt layouts, font sizes, and interactive elements based on screen size. Components like the FilterBar and AddTaskButton rearrange for better usability on smaller screens.

Filtering and Task Management

- **Filtering Options:**
    - Implemented a FilterBar component that allows users to search tasks by keyword, filter by category, and status.
    - Enhances user experience by making task management more efficient.
- **Task Categories and Statuses:**
    - Extended tasks to include categories (e.g., Work, Health) and statuses (Pending, In Progress, Completed).

Deadline Management

- **Date Picker Integration:**
    - Added a date picker for setting task deadlines using MUI's DatePicker.

User Experience Enhancements

- **Animations with Framer Motion:**
    - Incorporated animations to improve user engagement. It enhances visual feedback during interactions, such as adding or deleting tasks.
- **Notifications and Feedback:**
    - Provided feedback to users through notifications for actions like task creation, updates, and errors.

Project Structure

- **Frontend (**/frontend**):**
    - **__tests__**: Unit tests for components.
    - **api**/: Centralized Axios instance and API configurations.
    - **assets**/: Static assets such as images.
    - **components**/: Reusable UI components (e.g., TaskModal, TaskList, FilterBar, Loader).
    - **constants**/: Application-wide constants and enums.
    - **contexts**/: NotificationContext for global notification handling.
    - **hooks**/: Custom hooks (e.g., useAppDispatch, useAppSelector).
    - **pages**/: Page components (e.g., Tasks.tsx).
    - **selectors**/: Redux selectors for accessing state.
    - **slices**/: Redux slices for tasks state management.
    - **store**/: Redux store configuration.
    - **theme.ts**: Custom Material-UI theme configurations.
    - **types**/: TypeScript type definitions and interfaces.
    - **utils**/: Utility functions and helpers.

- **Backend (**/backend**):**
    - **models/:** Mongoose schema for the Task model.
    - **routes/:** Express routes for API endpoints.