Entry Manager

Design Document by Tongtong Jia

Overview

The Entry Manager application is built using modern frontend development practices, with a focus on modularity, scalability, and maintainability. This document outlines the key design choices, patterns, principles, and features implemented in the project.

1. Architectural Pattern: Fractal Pattern

Folder Structure:

```
src/
├── App.css
├── App.js
├── App.test.js
├── index.css
├── index.js
├── logo.svg
├── reportWebVitals.js
├── setupTests.js
├── components/
│   ├── Container/
│   │   ├── ContainerComponent.js
│   │   ├── useStyles.js
│   │   └── __tests__/
│   │       └── ContainerComponent.test.js
│   ├── common/
│   │   ├── AddButton.js
│   │   ├── Button.js
│   │   ├── Header.js
│   │   └── SubmitButton.js
│   ├── Table/
│   │   ├── Table.js
│   │   ├── TableActionButtons.js
│   │   ├── TableBodyComponent.js
│   │   ├── TableHeadComponent.js
│   │   └── __tests__/
│   │       ├── TableBodyComponent.test.js
│   │       └── TableHeadComponent.test.js
│   └── ModalForm/
│       ├── ModalForm.js
│       ├── ModalFormBody.js
│       ├── ModalHeaderComponent.js
│       ├── fields/
│       │   ├── CheckboxWithTooltip.js
│       │   ├── ErrorDisplay.js
│       │   ├── FieldFactory.js
│       │   ├── FormFieldWithTooltip.js
│       │   ├── FormFields.js
│       │   ├── SelectInputWithTooltip.js
│       │   ├── TextInputWithTooltip.js
│       │   └── style/
│       │       └── CheckboxWithTooltip.css
│       ├── formConfig.js
├── features/
```

```
│   └── entries/
│       ├── entriesSlice.js
│       ├── selectors.js
│       └── __tests__/
│           ├── entriesSlice.test.js
│           └── selectors.test.js
├── hooks/
│   ├── useDebounce.js
│   ├── useFormValidation.js
│   └── __tests__/
│       ├── useDebounce.test.js
│       └── useFormValidation.test.js
├── models/
│   ├── Entry.js
│   └── __tests__/
│       └── Entry.test.js
├── store/
│   ├── rootReducer.js
│   └── store.js
└── utils/
    ├── ValidationStrategy.js
    ├──formatFieldValue.js
    └── __tests__/
        ├── ValidationStrategy.test.js
        └──formatFieldValue.test.js
```

Why Fractal Pattern? The Fractal pattern is ideal for structuring the project because it promotes reusability and modularity. Each component is self-contained and can be composed with other components to build more complex features. This results in a scalable architecture where components can be developed, tested, and maintained independently.

Application of Fractal Pattern

- **Src:** Organised into folders based on their functional domain (e.g. Components, Table, ModalForm). Each component has its own styles, subcomponents, and utilities, leading to a clean and manageable codebase.
- **Modularity:** Components are reusable in different contexts without dependencies on other components, enhancing the maintainability and scalability of the application.

2. Centralised Configuration: `fieldsConfig` in `formConfig`

Why Centralised Configuration? The `fieldsConfig` centralised configuration pattern was chosen to simplify the management of form fields. This approach makes it easy to modify tooltip messages, validation rules, label text, and field types in a single location, reducing the chances of errors and improving maintainability.

Benefits of Centralised Configuration

- **Ease of Maintenance:** Changes to field configurations can be made in one place, automatically reflecting across the entire application.
- **Consistency:** Ensures consistent application of labels, validation rules, and tooltips across all forms.
- **Scalability:** Adding new fields or modifying existing ones becomes straightforward, allowing the application to scale easily.

Application of Centralised Configuration

- **`fieldsConfig`:** A central configuration object that defines each field's properties, including name, label, type, tooltipText, and rules. This configuration is passed to form components, ensuring that forms are dynamically generated based on these settings.

3. State Management: Redux

Why Redux? Redux was chosen for state management because it provides a predictable state container that centralises the application's state in a single store. This makes it easier to manage and debug the application's state, especially as it grows in complexity.

Reducers and Combined Reducers

- **Combined Reducer:** The application uses a combined reducer (`rootReducer`) to manage multiple slices of state, promoting separation of concerns.
- **Reducers:** The application utilises reducers for managing entries (`entriesSlice`). Each reducer is responsible for updating a specific slice of the state, which is combined into the root reducer.

Slices

- **Entries Slice:** Manages the state of entries, including actions for adding, updating, deleting, and setting edit data. This slice encapsulates the logic related to entries, keeping the application state management modular and organised.

Selectors

- **Memoized Selectors:** Selectors are used to derive state from the Redux store. Memoization (via reselect) ensures that selectors return cached results if the input state hasn't changed, improving performance by preventing unnecessary re-renders.
- **Selector Examples:**
  - `selectFilteredEntries`: Filters entries based on search terms.
  - `selectEditDataEntry`: Retrieves the entry data to be edited.

4. Hooks

Custom Hooks

- **`useDebounce`:** A custom hook that debounces user input in the search bar, preventing excessive renders and API calls by waiting for a specified delay before applying the search term.
- **`useFormValidation`:** A custom hook that encapsulates form validation logic, making it reusable across different forms. This hook integrates with validation strategies to provide dynamic validation based on the form's configuration.

React Hooks

- **`useState`:** Manages local state within components, such as search terms and form data.
- **`useEffect`:** Utilised for side effects, like setting form data when editing an entry.
- **`useSelector` and `useDispatch`:** Redux hooks that replace the traditional connect function, allowing components to interact with the Redux store more easily.

5. UI Frameworks: MUI and Bootstrap

Why MUI and Bootstrap?

- **MUI:** Material-UI (MUI) is used for its modern components, ease of use, and built-in theming capabilities. MUI provides a polished look and feel, along with rich components like tooltips and icons.
- **Bootstrap:** Bootstrap is used alongside MUI for its grid system and basic components, making it easy to create a responsive layout. This combination demonstrates flexibility and proficiency in using multiple UI libraries.
- Note: In this project, both MUI and Bootstrap were deliberately used together to showcase ability to use modern tools such as MUI as well as Bootstrap.

Integration of MUI and Bootstrap

- **`useStyles`:** MUI's `useStyles` hook creates custom styles with theming support, allowing consistent styling across components while leveraging MUI's theme system.
- **Bootstrap Components:** Bootstrap components like `Modal`, `Form`, and `Button` are integrated seamlessly with MUI, allowing for a mix of utility and aesthetic appeal.

6. Validation Logic

Validation Strategy Pattern The validation logic is implemented using the Strategy Pattern, allowing for different validation strategies to be applied to form fields. This approach makes the validation logic extensible and reusable.

Validation Strategies

- **Validation Strategies:** The `ValidationStrategy.js` file defines strategies like `required`, `positiveInteger`, and `uniqueId`. These strategies are dynamically applied based on the form field's configuration.
- **Contextual Validation:** The validation logic is context-aware, meaning it can differentiate between adding a new entry and editing an existing one. This ensures that validations are applied correctly based on the operation.

7. Modular Design

Promotion of Modular Design The application is designed with modularity in mind, allowing for independent development, testing, and maintenance of each component.

Reusable Components

- **Components:** `ModalForm`, `Table`, and `FormFields` are highly reusable across different parts of the application.
- **FieldFactory:** The `FieldFactory` dynamically creates form fields based on their type, promoting reusability and consistency.

8. Modern Design Patterns

Factory Pattern

- **FieldFactory:** The Factory Pattern is used in the `FieldFactory` to create different types of form fields (e.g., `TextInputWithTooltip`, `SelectInputWithTooltip`). This encapsulates the creation logic, making the code more maintainable.

Singleton Pattern

- **Redux Store:** The Redux store acts as a Singleton, ensuring that there is only one instance of the store throughout the application. This centralisation of state aligns with the Singleton Pattern principles.

Strategy Pattern

- **Validation Strategy:** The Strategy Pattern is applied in the form validation logic, where different validation strategies are used dynamically based on the form field's configuration.

Adapter Pattern

- **Custom Hooks:** Hooks like `useDebounce` and `useFormValidation` act as adapters, transforming and adapting the raw state and actions into a more usable form for components.

9. SOLID and GRASP Principles

SOLID Principles

- **Single Responsibility Principle (SRP):** Each component, utility, and hook has a single responsibility. For example, `TableBodyComponent` is only responsible for rendering the table's body, while validation logic is handled separately by `useFormValidation`.
- **Open/Closed Principle (OCP):** The application is designed to be open for extension but closed for modification. New features can be added (e.g., new validation strategies) without modifying existing code.
- **Liskov Substitution Principle (LSP):** Components and hooks are designed to be interchangeable without altering the behaviour of the system. For example, form fields can be easily swapped without affecting the overall functionality.
- **Interface Segregation Principle (ISP):** Interfaces (in the form of props) are kept small and focused. Components only expose the necessary props, ensuring that they are not burdened with unused properties.
- **Dependency Inversion Principle (DIP):** High-level modules (e.g., `ContainerComponent`) do not depend on low-level modules but on abstractions (e.g., selectors, hooks). This reduces coupling and enhances testability.

GRASP Principles

- **Controller:** The `ContainerComponent` acts as a controller, managing user input, coordinating actions, and updating the view.
- **Creator:** Components are responsible for creating instances of other components or utilities they use, adhering to the Creator principle.
- **High Cohesion:** Each component is cohesive, focusing on a single task, which simplifies maintenance and enhances readability.
- **Loose Coupling**: Fractal patterns promote loose coupling by making components modular, reusable, consistent, encapsulated, and scalable.

10. Memoization in Selectors

Why Memoization? Memoization in selectors (using reselect) ensures that selectors only recompute derived data when necessary, avoiding unnecessary computations and re-renders. This improves the performance of the application, especially when dealing with large datasets or complex calculations.

Implementation

- **Memoized Selectors:** Selectors like `selectFilteredEntries` and `selectEditDataEntry` are memoized, ensuring that they only recalculate when their input state changes.

11. Features and Functionality

Add New Entry

- **Functionality:** The "Add New Entry" feature allows users to add a new entry to the list. When the "Add" button is clicked, a modal form appears where users can fill in the details for the new entry.
- **Integration:** The form data is managed with local state and validated using the `useFormValidation` hook. Once validated, the form data is dispatched to the Redux store via the `addEntry` action.

Search Bar

- **Functionality:** The search bar allows users to filter the entries displayed in the table based on their search criteria.
- **Integration:** The search term is debounced using the `useDebounce` hook, and the filtered entries are retrieved using the `selectFilteredEntries` selector.

Edit and Delete

- **Functionality:** Users can edit or delete existing entries. The edit action pre-fills the form with the existing entry's data, while the delete action prompts the user for confirmation before removing the entry.
- **Integration:** The `handleEditClick` and `handleDeleteClick` functions dispatch the respective actions (`setEditData`, `deleteEntry`) to the Redux store.

Tooltip Messages

- **Functionality:** Tooltip messages provide guidance to users by offering additional information about each form field.
- **Integration:** Tooltips are dynamically generated based on the `tooltipText` property in the `fieldsConfig`, ensuring that they are easy to update and maintain.

Validation Logic

- **Functionality:** The application enforces validation rules to ensure data integrity. For example, the ID field must be a positive integer and unique, required fields needs to be filled with valid data formats. In case of violation, an appropriate message is displayed as an error handling.
- **Integration:** Validation is managed by the `useFormValidation` hook, which applies different validation strategies based on the field configuration.

12. Project Setup and Running the Application

Prerequisites

- **Node.js:** Ensure that Node.js is installed on your machine.
- **npm:** Ensure that npm (comes with Node.js) is installed.

Installation

- **Clone the Repository:**
  ```
  git clone https://github.com/mushroom8ge/entry-manager.git
  ```

- **Move to the repository**
  ```
  cd entry-manager
  ```

- **Install Dependencies:**
  ```
  npm install
  ```

Running the Application Locally

- **Start the Development Server:**
  ```
  npm start
  ```
  This will start the development server and open the application in your default web browser at `http://localhost:3000`.
- **Running Tests:**
  ```
  npm test
  ```

Conclusion

This document provides a comprehensive overview of the design choices, features, and integrations in the Entry Manager application. The project showcases the use of modern software design principles and best practices, ensuring a scalable, maintainable, and high-performance application. Additionally, the presence of thorough test files and global styles highlights the application's commitment to reliability and consistency.