Report on

Refactoring Duplicated Components in the SIGMA Project Developed by ReactJS

Related to the task SIG-126

Mohammad Mollaahmadi
MOJ Secure Company
Tehran, Iran
Jun 2024

Introduction

The SIGMA project, developed using the ReactJS library, currently contains multiple instances of duplicated components. This redundancy increases maintenance efforts and can lead to inconsistencies across the codebase. This report proposes refactoring these components into single, reusable ones to enhance code maintainability and consistency.

Ensuring that the proposed solution does not negatively impact page rendering performance is critical. Maintaining optimal performance metrics is a top priority throughout the implementation of these changes.

In this document, we will discuss three sections of the code: the API calls module, the routes handling hook, and the screen components. The reason for focusing on these three sections is to enable effective refactoring of the screen components by consolidating duplicated components into single, reusable ones.

To achieve this, we must ensure seamless API integration by having a unified API caller. Integrating all API calls into one module will allow the refactored components to use this single caller consistently.

Additionally, when using a single component for various procedures, the component needs to determine which procedure is active on the screen. This requires obtaining information from the route, necessitating the integration and standardization of route handling.

As a result, we need to refactor these three parts concurrently to achieve a cohesive and efficient codebase. Other sections, such as atomic and composite components, assets, Redux, styles, types, and more, will be addressed in a separate review process and report.

Current Situation

In this section, we will review the present state of the code for the three key parts: API Calls Module, Routes Handling Hook, and Components (screen components).

API Calls Module

To handle API calls, we use the `reduxjs/toolkit/query` tool, which streamlines data fetching and caching in our React application. This tool is imported into various components, each tailored to interact with specific endpoints. The primary difference between these components lies in their base URLs, allowing them to target different services or endpoints while leveraging the same robust data fetching and state management capabilities provided by `reduxjs/toolkit/query`.

The current implementation involves multiple instances of API calls scattered across various components. This redundancy leads to increased maintenance efforts and potential inconsistencies. Each component handling its API calls independently can result in fragmented and inconsistent code, making updates or modifications to API interactions more challenging and time-consuming.

Routes Handling Hook

For managing routes, we utilize a custom hook located in `src/useRoutesCustom.jsx`. This hook efficiently handles all routes, including those specifically designated for each procedure within the cartable. Each procedure's routes are distinctly separated, reflecting varied words and patterns tailored to their respective functionalities.

Furthermore, our route handling hook supports multiple methods of passing data, accommodating diverse needs across procedures. This includes utilizing both route parameters and query parameters to facilitate flexible data transmission through the application's routes.

Route Parameters

Route parameters are placeholders in the URL path that are used to capture dynamic values when navigating to different routes within an application. They are defined in the route path itself and are typically denoted by a colon followed by a parameter name.

```
"/users/:userId"
```

Use Cases:

- **Resource Identification:** When fetching specific resources, such as user profiles (`/users/:userId`) or product details (`/products/:productId`).
- **Dynamic Routing:** Navigating to dynamic pages based on user input or selections, such as filtering results (`/products/category/:categoryName`).

Query Parameters

Query parameters are key-value pairs attached to the end of a URL, separated from the base URL and from each other by a question mark `?` and an ampersand `&`, respectively.

```
"/search?q=query&sort=asc"
```

Use Cases:

- **Filtering and Sorting:** Passing filter criteria (`?category=electronics`) or sorting preferences (`?sortBy=date&order=desc`) to display specific data subsets.
- **Pagination:** Specifying page numbers (`?page=2&limit=10`) to navigate through large sets of data.

Components (Screen Components)

As we mentioned, the current ReactJS code for the SIGMA project contains multiple instances of duplicated components, leading to increased maintenance efforts and inconsistencies.

To address this issue, I developed a script to find the similarity between files in directories such as the `src` directory. This script allows for adjusting the threshold to determine what constitutes a match. The threshold can be easily modified within the code to suit different levels of sensitivity.

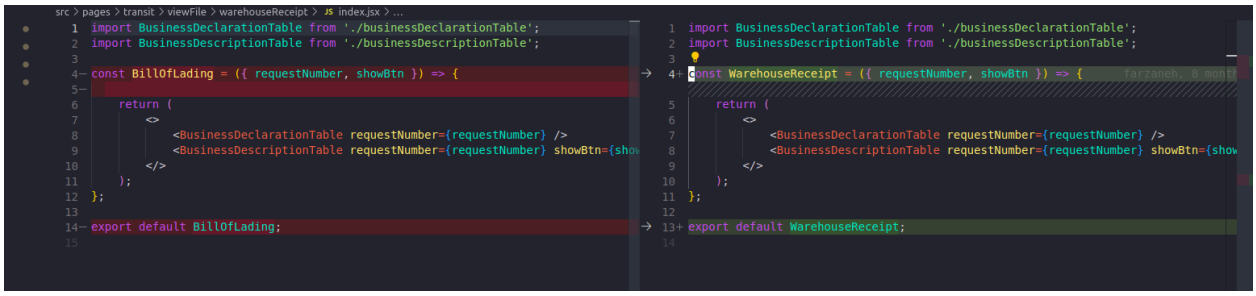
Attached, you will find the script code and the results of running the script on the project files. As an example, we highlight two cases where file addresses have a similarity of 90 percent. These samples clearly indicate where duplication occurs, which is crucial for our refactoring efforts to create reusable components and streamline the codebase.

Here are some examples of identified duplications with a similarity of 90 percent or more:

- src/pages/TemporaryExit/viewFile/warehouseReceipt/index.jsx
- src/pages/export/viewFile/warehouseReceipt/index.jsx
- src/pages/import/viewFile/billoflading/index.jsx
- src/pages/import/viewFile/warehouseReceipt/index.jsx

- src/pages/transit/viewFile/billoflading/index.jsx
- src/pages/transit/viewFile/warehouseReceipt/index.jsx
- src/pages/returned-management/viewFile/shipping-document-tab/BillOfLading.jsx
- src/pages/returned-management/viewFile/warehouse-bill-tab/WarehouseReceipt.jsx

In these files, there is almost 100% similarity between the `BillOfLading` and `WarehouseReceipt` components across the import, transit, export, temporary exit, and returned directories.



Another significant duplication found by the script involves components responsible for displaying tables inside the `BillOfLading` and `WarehouseReceipt` components. Here are the files with nearly identical `businessDescriptionTable` components:

- src/pages/TemporaryExit/viewFile/warehouseReceipt/businessDescriptionTable/index.jsx
- src/pages/import/viewFile/warehouseReceipt/businessDescriptionTable/index.jsx
- src/pages/import/viewFile/billoflading/businessDescriptionTable/index.jsx
- src/pages/returned-management/viewFile/shipping-document-tab/businessDescriptionTable/index.jsx
- src/pages/transit/viewFile/billoflading/businessDescriptionTable/index.jsx

The duplication of components such as `BillOfLading`, `WarehouseReceipt`, and `businessDescriptionTable` across various directories indicates a clear need for refactoring. By creating reusable components, we can significantly reduce maintenance efforts and inconsistencies, leading to a more streamlined and efficient codebase.

Proposed Solution

In the following section, we present our recommended approach:

API Calls Module

To streamline the API calls module, we propose developing a centralized module that manages different endpoints based on their respective base URLs for each procedure (e.g., import, export). This module will be integrated into a hook that dynamically retrieves the procedure name from the route and determines the appropriate base URL. By structuring API calls this way, we ensure efficient and consistent data fetching tailored to specific procedures identified via routing parameters.

Route Handling Hook

Each screen and page will be assigned a unique route that incorporates parameters to identify the procedure dynamically. This approach ensures that every screen is distinctly represented in the routing structure, accommodating all procedures seamlessly.

By implementing these strategies, we enhance the clarity and efficiency of API interactions and route management within the application, fostering a more organized and maintainable codebase.

Components

Refactor the duplicated components to create a single, reusable component for each group. This will involve:

- Identifying common functionality and props.
- Creating a new component that encapsulates this functionality.
- Replacing the duplicated instances with the new component.

Project Structure

To further enhance the organization and scalability of the project, we propose adopting a new project structure that categorizes components into three sections: atomic, composite, and screen components. This structure will facilitate a clear separation of concerns and better manageability.

Atomic Components: The smallest building blocks, such as buttons, input fields, and icons. These components are highly reusable and do not contain any business logic.

Example: Button, Input, Icon

Implementation: Develop individual, reusable atomic components. Ensure each component is self-contained and easily testable.

Composite Components: Components that combine multiple atomic components to form more complex elements with some logic.

Example: SideBar, Navbar

Implementation: Create composite components by combining atomic components. Encapsulate specific functionalities and business logic.

Screen Components: These components represent entire screens or significant parts of a screen. They combine atomic and composite components and contain the main logic for specific views.

Example: LoginScreen, CartableScreen, DeclarationScreen

Implementation: Assemble screen components by integrating atomic and composite components. Handle state management and overall layout.

By implementing this structure, we can use screen components directly within pages, ensuring a clean and organized codebase. This approach promotes reusability, reduces redundancy, and simplifies maintenance.

Benefits of Refactoring

Reduced Maintenance

Having a single source of truth for each component reduces the risk of inconsistencies and simplifies updates.

Improved Consistency

Ensures a uniform look and feel across the application.

Enhanced Readability

Simplifies the codebase, making it easier for new developers to understand.

Implementation Plan

This plan comprises three key parts: components, API calls module, and routes handling hook. We will discuss each in the following.

1. API Calling

Step 1: Develop a General Hook

Create a generalized hook that can handle various API calls by accepting the name of the service as a parameter.

This hook should integrate with the Redux Toolkit to manage the state efficiently.

Step 2: Integrate with the Redux Toolkit

Ensure the hook is connected to Redux Toolkit actions and reducers to handle data fetching and state management.

Note: Potential changes to API URIs may be required, and close cooperation with backend developers is essential to ensure smooth implementation.

2. Components

Step 1: Identify common functionality and props for each set of duplicated components.

Conduct a thorough analysis of the existing components to determine shared functionalities and properties.

Step 2: Create a new component that encapsulates this functionality.

Develop a reusable component that combines the identified functionalities and props.

Step 3: Use the developed hook to have an instance of API caller in the component if it is needed.

According to the functionality of the component, we have to use the developed hook to call the APIs of different procedures.

Step 4: Replace the duplicated components with the new components across the project.

Systematically update the codebase to replace instances of duplicated components with the new reusable component.

Step 5: Test the refactored components to ensure they function as expected.

Perform comprehensive testing to verify that the refactored components work correctly and maintain existing functionality.

3. Routes

Step 1: Simplify Route Integration

Refactor the routes to allow easy changes to the base paths, ensuring flexibility for different procedures like import and export.

Note: Potential changes to the menu URLs may be required, and close cooperation with backend developers is essential to ensure smooth implementation.

By integrating this approach, we ensure that the route structure remains consistent and easy to manage when adding or modifying procedures.

Impact Analysis

1. Timeline:

The refactoring process is estimated to take approximately 70-90 hours, depending on the number of components involved and their complexity.

Given the necessary changes discussed in the previous sections, the following steps will be undertaken:

Step	Description	Time Estimate
1	Develop a New Hook for API Calls	14-21 Hours
2	Refactor Similar Components	56-69 Hours
3	Update Routes	

The two final steps must be carried out concurrently to ensure a cohesive integration of the new API handling mechanism and the revised component and route structures.

2. Team Workload:

The team will need to allocate time for both refactoring and testing. It is evident that conflicts may arise as the team continues to work on the project and develop new features. To mitigate this and minimize impact, we will maintain a list of changed files and compare it to the development branch daily. This will ensure all changes are consistently applied to the new refactored version, preventing discrepancies and maintaining alignment with ongoing development efforts.

3. Dependencies:

As we mentioned, collaboration with the backend team will be necessary to unify the API URLs and request bodies, as well as menu routes. This will ensure consistency across the entire application and prevent any potential integration issues.

Conclusion

Refactoring the duplicated components into reusable ones will simplify the codebase, decrease maintenance efforts, and enhance consistency throughout the project. The outlined implementation plan should be followed to carry out the refactoring process.

Suggestions and Future Plans

1. **Integrate Storybook into the Project:**

Implement the Storybook to document, develop, and visualize atomic and composite components in isolation. This will enhance the development process by providing a clear, interactive interface for testing and showcasing UI components.

2. **Refactor Atomic and Composite Components:**

Use Storybook to facilitate the refactoring of atomic and composite components, ensuring they are modular, reusable, and well-documented. This will improve code maintainability and consistency across the project.

3. **Unify Logic of Codes:**

Standardize error handling across the project to ensure consistent and predictable behavior. This includes implementing a centralized error-handling mechanism.

4. **Unify Importing Assets:**

Consolidate the process of importing assets, particularly icons, to maintain consistency and reduce redundancy. Establish a centralized directory or system for managing assets to streamline their usage across the project.

5. **Add TypeScript to the Project:**

Incorporate TypeScript to enhance code quality and maintainability through static typing. This will help catch errors early in the development process and improve the overall robustness of the codebase.

6. **Use Cypress in the Project:**

Integrate Cypress for end-to-end testing to ensure the reliability and stability of the application. Cypress provides a powerful framework for testing the entire application workflow, catching bugs, and ensuring consistent behavior across different scenarios.