Report on

Refactoring Duplicated Components in the SIGMA Project Developed by ReactJS

Third Step: Components Modification

Related to the tasks SIG-126, MOJ-202, MOJ-203 and SIG-220

Mohammad Mollaahmadi and Nima Esfahani
MOJ Secure Company
Tehran, Iran
Apr 2025

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. We recently reviewed the project's current state and devised a three-step plan for refactoring: the API calls module, the routes handling hook, and the screen components.

This document focuses on the third phase—refactoring and restructuring the components—and outlines our proposed approach to improving them. Specifically, the report is organized into three key parts:

Component Structure Design: A categorized list of essential components (atomic, composite, screen, and layout) required to support a consistent and scalable design system.

Component Similarity Analysis: A list of screen components sharing over 90% similarity, identified through an automated JavaScript script. These can be consolidated to reduce redundancy.

Recommended Changes: A summary of the structural modifications needed to improve maintainability, consistency, and scalability of the project going forward.

This report provides a clear vision for the codebase's future structure and helps define the tasks needed to achieve a more maintainable and robust system.

Implementation

Step 1: Define a Comprehensive Component Structure

In the initial phase, we established a well-structured set of essential components required for building and implementing the design system across the entire project. To maintain clarity and scalability, we categorized components based on their functionality and composition:

1. **Atomic Components:** Fundamental UI elements that perform a single function and cannot be broken down further.
2. **Composite Components:** Built from two or more atomic components to provide more complex functionality.
3. **Screen Components:** High-level components composed of multiple composite and atomic components, representing a complete screen or feature.
4. **Layout Components:** Responsible for managing the layout of content, such as modals or page containers.

Note: In the event of adopting a micro-frontend architecture in the future, atomic, composite, and layout components should be shared across micro-frontends. Screen components, however, should remain isolated to each micro-application unless they are confirmed to be common, which will be identified in upcoming steps.

We've compiled a list of required components. Some already exist and will need refactoring to align with the new system, while others will need to be created from scratch.

Atomic Components

1. Alert
2. Button (Text, Icon)
3. Chps
4. DatePicker
5. Input
6. Label
7. Link
8. Loading Spinner
9. Message
10. MultiSelect
11. RadioButton
12. Select
13. Stepper
14. Switch
15. TextArea
16. Title
17. Toast

Composite Components

1. Comment
2. PDF Viewer
3. ProfileImage (display, upload)
4. Tab
5. Table
6. UploadFile
7. Accept or Reject Modal
8. SearchInput

Layout Components

1. Accordion
2. Action Box
3. Card
4. Container
5. Modal

Screen Components

1. Insert Comment
2. Declaration Edit Request
3. Declaration Form
4. Declaration Form PDF
5. Declaration Reports
11. Travel Info Screen
12. Travel Report Screen
13. Users Accesses
14. Value Management
6. Declaration Summation Screen
7. Financial Bills Screen
8. Seals List Screen
9. Through Transport
10. Through Transport Detail
15. Warehouse Bijak Information Screen

Note: We need to relocate the CertificateInspection and CertificateProduction screens from the Import module to the Components screens, so they can be used in both the Review and Import processes.

Step 2: Component Similarity Analysis

In addition to proposing a new, well-structured component design, we conducted a comprehensive review of existing screen components to identify opportunities for reusability across different modules (micros).

To facilitate this, we developed and executed a JavaScript script that scanned the project directory and compared all files to detect components with over **90% similarity**.

The analysis identified **97 cases** of similarity. Following a detailed review, we have outlined the components that could potentially be made reusable, assuming no future business constraints arise.

The complete list of these components — aligned with the screen components mentioned in the previous section — is available in the attached Excel file.

Note: All relevant files, along with the script used for this analysis, are included in the attachments accompanying this report.

Step 3: Recommended Changes

After identifying the components that require modification to achieve a well-structured and reusable design system, we recommend implementing certain changes. These adjustments are essential to improve project maintainability and facilitate the development of new features more efficiently.

1. Standardize Icons in the `/assets` Directory

- Convert all icons to SVG format.
- Create a separate SVG file for each icon.
- Enable icons to accept properties such as `color`, `tooltipContext`, and `tooltipPlace`.

2. Consolidate `/type` and `/constants` Directories

- Remove the `/constants` directory and transfer relevant data into the `/type` structure.
- Organize all data types used in components under `/type`.
- Create a dedicated type file for each module, along with a general type file for the project-wide types.

3. Merge `/utils` and `/hooks` Directories

- Review the definitions of "utils" and "hooks" and restructure them to create two clean, well-separated directories:
    - **Util Methods:** General-purpose utility functions.
    - **Hooks:** Custom React hooks with clear responsibilities.

4. Integrate Component Styles

- Create a separate style file for each component.
- Move existing module-specific styles into the corresponding component's style directory.

5. Optimize `/routes` Structure

- Remove unused and duplicated routes, particularly in nested levels, to simplify navigation handling.

6. Improve URL Opening Behavior

- Refine the logic for opening complex nested routes, ensuring the correct default menu state is handled when accessing URLs directly.

7. Refactor Redux Usage

- Review the use of Redux throughout the project.

- Remove unnecessary usages and, where possible, localize the state management inside components to reduce complexity.

8. Resolve Console Errors

- Identify and fix all remaining console errors to improve the project's stability and developer experience.

9. Secure JWT Handling

- Investigate storing JWT tokens securely in local storage.
- Clean up local storage by removing unnecessary information.

10. Breadcrumb Refactoring

- Gather a clear list of requirements for the breadcrumb component.
- Refactor the existing breadcrumb implementation to align with the new requirements.

Conclusion

This document is a proposal, not a report of completed changes. It outlines the current issues and presents recommendations to improve the SIGMA project's structure, maintainability, and scalability. No changes have been made to the codebase yet. To successfully implement these improvements, we should proceed carefully and address them step by step, minimizing risks and ensuring stability. A detailed change list has been provided to guide the implementation process and help prioritize tasks based on project needs and resource availability.