Module Enhancement Documentation

Project Overview

1. Project Purpose and Scope

The purpose of this project was to enhance an existing TypeScript module to improve maintainability, readability, and scalability. The existing module had complex logic and lacked consistency in error handling, modularity, and testing, making it difficult to extend and maintain. The goal was to refactor the codebase, improve performance, and add unit tests to ensure robustness.

2. Target Audience

This enhancement targets internal developers, QA engineers, and future developers who will maintain and extend the system. The improvements will help streamline the development process and make the system more stable for future features.

Development Process

1. Planning and Requirements

- **Problem Statement**: The existing TypeScript codebase had a high level of complexity, scattered logic, and repeated error-handling code. It was difficult to read, debug, and extend.
- **Functional Requirements**: Refactor code to improve readability, error handling, and testing. Ensure that all key functions are tested with adequate coverage.
- **Non-functional Requirements**: The enhanced module should be more maintainable, easier to debug, and more efficient in handling errors and API calls.

2. Tools and Technologies

- **Backend**: TypeScript (Node.js with TypeScript)
- **Testing**: Jest for unit testing, mocking external dependencies for isolated testing.

Development Process and Decisions

1. Key Design Decisions

- **Why Jest Was Chosen For Testing**:
    - Jest was selected for unit testing due to its simplicity and wide adoption in the JavaScript/TypeScript ecosystem.
- **Challenges Faced**:
    - The TypeScript code had multiple layers of nested promises and if-else blocks, which made it hard to follow.
    - There were duplicate error-handling mechanisms scattered throughout the codebase.
- **Trade-offs**:
    - Refactoring the code involved significant changes, which meant that some functionality might have been temporarily broken during the transition. However, this trade-off was necessary for long-term maintainability.

2. TypeScript-Specific Decisions

- **Types and Interfaces**:
    - Types and interfaces were introduced or improved to provide type safety, making the code more predictable and preventing certain types of errors.
    - The `useHandleSwap` function's parameters and return values were explicitly typed, ensuring better documentation and error prevention.
- **Enum for Statuses**:
    - Replaced strings with TypeScript enums for more robust and maintainable status management in the module.

---

3. Implementation Details

- **High-level Code Structure**:
    - The logic was split into separate files for better modularity: one for handling API interactions, another for business logic, and one more for error handling.
- **Major Features**:
    - The `useHandleSwap` function was refactored to simplify error handling by using a centralized error-handling function. This ensured that any errors encountered during API calls would be processed uniformly across the module.
    - **Before Refactoring**: Nested promises and if-else clauses in the `useHandleSwap` function created unnecessary complexity.
    - **After Refactoring**: The function was converted to an `async` function using `try-catch` blocks for error handling. This made the code much more readable and easy to follow.
- **Performance Optimizations**:
    - API calls were optimized using async/await to eliminate nested callback hell, improving readability and performance.
    - Error handling was optimized by consolidating redundant error handling logic into a single function.
- **Error Handling and Logging**:
    - **Before Refactoring**: Error handling was duplicated in multiple places, making the code harder to maintain.

- **After Refactoring**: A `customErrorHandle` function was created to handle errors centrally, reducing code duplication and improving consistency.

Testing and Validation

1. Testing Strategy

- **Unit Testing**: Unit tests were added for key functions such as `handleSwap`, `handleSwapFromCardanoWallet`, `handleSwapFromTronLinkWallet`, and the `customErrorHandle` function.
- **Mocking External Dependencies**: API calls and wallet interactions were mocked using Jest to isolate the logic and test individual functions in isolation.
- **Testing Strategy Overview**:
  - Each function was tested for correct behavior under normal conditions.
  - Edge cases such as invalid parameters and network failures were tested to ensure the error-handling logic was effective.

2. Test Coverage and Results

- **Test Coverage**: The test coverage was increased to 85% for the module after refactoring.
- **Bug Tracking and Issue Resolution**: All bugs encountered during testing were tracked in JIRA, with corresponding fixes made and validated through the unit tests.

Future Considerations

1. Potential Improvements

- **Scalability**: Future improvements could focus on optimizing the API further to handle a larger volume of requests.
- **Code Modularity**: The existing logic can be refactored into smaller, reusable components as the system grows.

2. Planned Features

- **Feature Additions**:
  - The module could be enhanced by adding support for additional wallets and cryptocurrencies in the `handleSwap` function.
  - Adding logging of successful operations (currently only errors are logged) to improve visibility into normal operation.

Conclusion

In conclusion, the TypeScript module was enhanced by improving code readability, performance, and error handling. The refactoring made the module more maintainable and easier to extend. Unit tests were added to ensure that future changes do not introduce regressions. With these improvements, the module is now better prepared to handle future feature additions and changes.

Key decisions involved simplifying asynchronous handling with async/await, consolidating error handling, and introducing robust testing practices to ensure the stability of the code.