Smart Wallet Requirements Analysis

User

A smart wallet is a next-generation digital wallet that combines traditional payment functionality with advanced blockchain technology and programmable features. Unlike basic cryptocurrency wallets that simply store and transfer digital assets, smart wallets can execute complex operations through smart contracts, automate transactions, and interact directly with decentralized applications (dApps). They offer enhanced security through features like multi-signature authentication, social recovery options, and customizable spending limits. Smart wallets also enable users to manage multiple cryptocurrencies, tokens, and NFTs in one interface, while providing advanced features like account abstraction, gas fee optimization, and batch transactions. This technology is revolutionizing how users interact with Web3 platforms by simplifying complex blockchain operations and offering a more user-friendly experience comparable to traditional financial services.

Identify functional and non-functional requirements from the system description.

AI Model

Functional Requirements:

1. **Smart Contract Execution**: Execute complex operations through smart contracts.
2. **Automated Transactions**: Automate transactions based on predefined rules.
3. **Decentralized Application Interaction**: Interact directly with decentralized applications.
4. **Multi-Signature Authentication**: Support multi-signature authentication for enhanced security.
5. **Social Recovery Options**: Provide social recovery options for account recovery.
6. **Customizable Spending Limits**: Allow users to set customizable spending limits.
7. **Multi-Asset Management**: Manage multiple cryptocurrencies, tokens, and NFTs in one interface.
8. **Account Abstraction**: Support account abstraction for advanced user management.
9. **Gas Fee Optimization**: Optimize gas fees for efficient transactions.

10. **Batch Transactions**: Support batch transactions for efficient processing.

Non-Functional Requirements:

1. **Security**: Ensure enhanced security through advanced features.
2. **User-Friendliness**: Offer a user-friendly experience comparable to traditional financial services.
3. **Simplification**: Simplify complex blockchain operations for easy usage.
4. **Interoperability**: Interact seamlessly with Web3 platforms.
5. **Scalability**: Support efficient processing of batch transactions and automation.
6. **Reliability**: Ensure reliable execution of smart contracts and automated transactions.
7. **Performance**: Optimize gas fees and processing times for efficient transactions.

User

Create a use case model for the system.

AI Model

Use Case Model for Smart Wallet System

Actors:

1. **User**: Individuals who own and operate the smart wallet.
2. **Decentralized Applications (dApps)**: External applications that interact with the smart wallet.
3. **Blockchain Network**: The underlying blockchain infrastructure that supports the smart wallet.

Use Cases:

1. **UC1: Set Up Smart Wallet**
    - Actor: User
    - Description: User creates and configures their smart wallet.

2. **UC2: Deposit Cryptocurrency**
    - Actor: User
    - Description: User deposits cryptocurrency into their smart wallet.
3. **UC3: Execute Smart Contract**
    - Actor: User
    - Description: User initiates a smart contract execution.
4. **UC4: Automate Transaction**
    - Actor: User
    - Description: User sets up automated transactions based on predefined rules.
5. **UC5: Interact with dApps**
    - Actor: User
    - Description: User interacts with decentralized applications using the smart wallet.
6. **UC6: Manage Multiple Assets**
    - Actor: User
    - Description: User manages multiple cryptocurrencies, tokens, and NFTs within the smart wallet.
7. **UC7: Optimize Gas Fees**
    - Actor: Smart Wallet System
    - Description: The system optimizes gas fees for efficient transactions.
8. **UC8: Recover Account**
    - Actor: User
    - Description: User recovers their account using social recovery options.
9. **UC9: Set Custom Spending Limits**
    - Actor: User
    - Description: User sets customizable spending limits.

10. **UC10: Perform Batch Transactions**
    - Actor: User
    - Description: User performs batch transactions for efficient processing.

Extensions:

- UC3.1: Smart Contract Execution Fails (e.g., due to insufficient funds)
- UC4.1: Automated Transaction Fails (e.g., due to invalid rules)
- UC8.1: Account Recovery Fails (e.g., due to invalid social recovery options)

Includes:

- UC2 includes UC7 (Optimize Gas Fees)
- UC5 includes UC3 (Execute Smart Contract)
- UC10 includes UC7 (Optimize Gas Fees)

This use case model outlines the interactions between the user, decentralized applications, and the blockchain network, highlighting the key functionalities and extensions of the smart wallet system.

User

Create use case specifications for identified use cases.

AI Model

Use Case Specifications

UC1: Set Up Smart Wallet

- **Description**: User creates and configures their smart wallet.
- **Preconditions**: User has a blockchain account and a device with internet connectivity.
- **Triggers**: User decides to set up a smart wallet.
- **Basic Flow**:
    1. User downloads and installs the smart wallet application.
    2. User creates a new wallet or imports an existing one.

    3. User sets a strong password and enables biometric authentication (optional).
    4. User configures wallet settings (e.g., currency, language).
    5. System generates a unique wallet address.
- **Alternative Flows**:
    - User restores a previously backed-up wallet.
    - User imports a wallet from a different platform.
- **Postconditions**: User has a fully functional smart wallet.

UC2: Deposit Cryptocurrency

- **Description**: User deposits cryptocurrency into their smart wallet.
- **Preconditions**: User has a smart wallet set up and a blockchain account with sufficient funds.
- **Triggers**: User decides to deposit cryptocurrency.
- **Basic Flow**:
    1. User selects the deposit option.
    2. User chooses the cryptocurrency to deposit.
    3. User enters the amount to deposit.
    4. System generates a deposit address.
    5. User confirms the transaction.
- **Alternative Flows**:
    - User scans a QR code to initiate the deposit.
    - User imports a transaction history from a different platform.
- **Postconditions**: User's smart wallet balance is updated.

UC3: Execute Smart Contract

- **Description**: User initiates a smart contract execution.
- **Preconditions**: User has a smart wallet set up and sufficient funds.
- **Triggers**: User decides to execute a smart contract.

- **Basic Flow**:
  1. User selects the smart contract to execute.
  2. User reviews the contract terms and conditions.
  3. User confirms the execution.
  4. System executes the smart contract.
- **Alternative Flows**:
  - User modifies the contract parameters before execution.
  - User cancels the execution.
- **Postconditions**: Smart contract is executed, and user's wallet balance is updated.

Please let me know if you'd like me to continue with the rest of the use cases!

User

Create a domain model based on use case specifications.

AI Model

Domain Model

Based on the use case specifications, the following domain model is derived:

Entities:

1. **User**
   - Attributes: userId, name, password, biometricData (optional)
   - Relationships: owns Wallet, executes SmartContract, interacts with DecentralizedApplication
2. **Wallet**
   - Attributes: walletId, userId, balance, currency, settings (e.g., language)
   - Relationships: contains MultipleAssets, generates DepositAddress
3. **MultipleAssets**

    - Attributes: assetId, walletId, assetType (cryptocurrency, token, NFT), balance
    - Relationships: part of Wallet
4. **SmartContract**
    - Attributes: contractId, name, termsAndConditions, executionStatus
    - Relationships: executed by User, interacts with DecentralizedApplication
5. **DecentralizedApplication (dApp)**
    - Attributes: dappId, name, description
    - Relationships: interacts with User, SmartContract, and Wallet
6. **DepositAddress**
    - Attributes: addressId, walletId, cryptocurrency
    - Relationships: generated by Wallet
7. **Transaction**
    - Attributes: transactionId, walletId, assetId, amount, timestamp
    - Relationships: part of Wallet, executes SmartContract

Value Objects:

1. **Amount**
    - Attributes: value, currency
2. **Cryptocurrency**
    - Attributes: symbol, name

Enums:

1. **AssetType** (cryptocurrency, token, NFT)
2. **ExecutionStatus** (pending, executed, failed)

This domain model captures the key concepts and relationships derived from the use case specifications, providing a solid foundation for further design and implementation. Let me know if you'd like me to elaborate or refine any aspects!

User

Identify system operations from use case specifications.

AI Model

System Operations

Based on the use case specifications, the following system operations can be identified:

User Management

1. **createUser**: Creates a new user account.
2. **importWallet**: Imports an existing wallet.
3. **restoreWallet**: Restores a previously backed-up wallet.
4. **setPassword**: Sets or updates a user's password.
5. **setBiometricAuth**: Enables or disables biometric authentication.

Wallet Management

1. **createWallet**: Creates a new wallet.
2. **importWallet**: Imports an existing wallet.
3. **getWalletBalance**: Retrieves the current wallet balance.
4. **updateWalletSettings**: Updates wallet settings (e.g., language, currency).

Asset Management

1. **depositAsset**: Deposits an asset (cryptocurrency, token, NFT) into the wallet.
2. **getAssetBalance**: Retrieves the current asset balance.
3. **transferAsset**: Transfers an asset to another wallet or address.
4. **executeSmartContract**: Executes a smart contract.

Smart Contract Management

1. **createSmartContract**: Creates a new smart contract.
2. **getSmartContract**: Retrieves a smart contract by ID or name.
3. **executeSmartContract**: Executes a smart contract.
4. **getSmartContractStatus**: Retrieves the execution status of a smart contract.

Transaction Management

1. **createTransaction**: Creates a new transaction.
2. **getTransaction**: Retrieves a transaction by ID.
3. **getTransactionHistory**: Retrieves a list of past transactions.
4. **confirmTransaction**: Confirms a transaction.

Decentralized Application (dApp) Management

1. **interactWithDapp**: Interacts with a decentralized application.
2. **getDappList**: Retrieves a list of available dApps.
3. **getDappDetails**: Retrieves details about a specific dApp.

These system operations provide a comprehensive overview of the functionality required to support the use cases. They can be used as a basis for further design, implementation, and testing. Let me know if you'd like me to elaborate or refine any of these operations!

User

Create design sequence diagrams for system operations.

AI Model

Here are some design sequence diagrams for the system operations:

1. createUser

User -> System: request CreateUser
System -> User: prompt for username and password
User -> System: provide username and password
System -> System: validate username and password
System -> User: confirm user creation

2. depositAsset

User -> System: request DepositAsset
System -> User: prompt for asset type and amount
User -> System: provide asset type and amount
System -> System: validate asset type and amount

System -> System: generate deposit address
System -> User: display deposit address
User -> System: confirm deposit

3. executeSmartContract

User -> System: request ExecuteSmartContract
System -> User: prompt for smart contract ID and parameters
User -> System: provide smart contract ID and parameters
System -> System: validate smart contract ID and parameters
System -> System: execute smart contract
System -> User: display execution result

4. interactWithDapp

User -> System: request InteractWithDapp
System -> User: prompt for dApp ID and input data
User -> System: provide dApp ID and input data
System -> System: validate dApp ID and input data
System -> System: interact with dApp
System -> User: display dApp response

5. getTransactionHistory

User -> System: request GetTransactionHistory
System -> System: retrieve transaction history
System -> User: display transaction history

These sequence diagrams illustrate the interactions between the user and the system for each operation. They can be used to further refine the design and implementation of the system. Let me know if you'd like me to create diagrams for additional operations!

User

Create design class diagrams based on the domain model and sequence diagrams.

AI Model

Here are some design class diagrams based on the domain model and sequence diagrams:

User Management

- **User**
    - Attributes: userId, username, password, biometricData (optional)
    - Methods: createUser(), setPassword(), setBiometricAuth()
- **UserManager**
    - Methods: createUser(), getUser(), updateUser()

Wallet Management

- **Wallet**
    - Attributes: walletId, userId, balance, currency, settings (e.g., language)
    - Methods: createWallet(), getWalletBalance(), updateWalletSettings()
- **WalletManager**
    - Methods: createWallet(), getWallet(), updateWallet()

Asset Management

- **Asset**
    - Attributes: assetId, walletId, assetType (cryptocurrency, token, NFT), balance
    - Methods: depositAsset(), getAssetBalance(), transferAsset()
- **AssetManager**
    - Methods: depositAsset(), getAsset(), transferAsset()

Smart Contract Management

- **SmartContract**
    - Attributes: contractId, name, termsAndConditions, executionStatus
    - Methods: createSmartContract(), getSmartContract(), executeSmartContract()
- **SmartContractManager**
    - Methods: createSmartContract(), getSmartContract(), executeSmartContract()

Transaction Management

- **Transaction**
    - Attributes: transactionId, walletId, assetId, amount, timestamp
    - Methods: createTransaction(), getTransaction(), confirmTransaction()
- **TransactionManager**
    - Methods: createTransaction(), getTransaction(), confirmTransaction()

Decentralized Application (dApp) Management

- **Dapp**
    - Attributes: dappId, name, description
    - Methods: interactWithDapp(), getDappList(), getDappDetails()
- **DappManager**
    - Methods: interactWithDapp(), getDappList(), getDappDetails()

These class diagrams illustrate the relationships and interactions between the classes, which can be used to guide the implementation of the system. Let me know if you'd like me to elaborate or refine any of these diagrams!

User

Develop a Java Implementation for the system as specified in the class diagram and sequence diagrams.

AI Model

Here is a Java implementation for the system based on the class diagram and sequence diagrams:

User Management

Java

```
public class User {
  private int userId;
  private String username;
  private String password;
}


  private byte[] biometricData;


  public User(int userId, String username, String password, byte[] biometricData) {
      this.userId = userId;
      this.username = username;
      this.password = password;
      this.biometricData = biometricData;
  }


  public int getUserId() {
      return userId;
  }


  public String getUsername() {
      return username;
  }


  public String getPassword() {
      return password;
  }


  public byte[] getBiometricData() {
      return biometricData;
  }
}
```

```
public class UserManager {
    public User createUser(String username, String password, byte[] biometricData) {
        // Implement user creation logic
    }

    public User getUser(int userId) {
        // Implement user retrieval logic
    }

    public void updateUser(User user) {
        // Implement user update logic
    }
}
```

Wallet Management

Java

```
public class Wallet {
    private int walletId;
    private int userId;
    private double balance;
    private String currency;
    private String settings;

    public Wallet(int walletId, int userId, double balance, String currency, String settings) {
        this.walletId = walletId;
        this.userId = userId;
        this.balance = balance;
        this.currency = currency;
        this.settings = settings;
    }

    public int getWalletId() {
        return walletId;
    }

    public int getUserId() {
        return userId;
    }

    public double getBalance() {
        return balance;
    }

    public String getCurrency() {
        return currency;
    }

    public String getSettings() {
        return settings;
    }
}

public class WalletManager {
    }

    public Wallet createWallet(int userId, double balance, String currency, String settings) {
        // Implement wallet creation logic
    }

    public Wallet getWallet(int walletId) {
        // Implement wallet retrieval logic
    }

    public void updateWallet(Wallet wallet) {
        // Implement wallet update logic
    }
}
```

Asset Management

Java

```
public class Asset {
    private int assetId;
    private int walletId;
    private String assetType;
    private double balance;

    public Asset(int assetId, int walletId, String assetType, double balance) {
        this.assetId = assetId;
        this.walletId = walletId;
        this.assetType = assetType;
        this.balance = balance;
    }
    }


    public int getAssetId() {
        return assetId;
    }


    public int getWalletId() {
        return walletId;
    }


    public String getAssetType() {
        return assetType;
    }


    public double getBalance() {
        return balance;
    }
}


public class AssetManager {
    public Asset depositAsset(int walletId, String assetType, double amount) {
        // Implement asset deposit logic
    }


    public Asset getAsset(int assetId) {
        // Implement asset retrieval logic
    }
    public void transferAsset(Asset asset, int toWalletId) {
        // Implement asset transfer logic
    }
}
```

Smart Contract Management

Java

```
public class SmartContract {
    private int contractId;
    private String name;
    private String termsAndConditions;
    private boolean executionStatus;

    public SmartContract(int contractId, String name, String termsAndConditions, boolean executionStatus) {
        this.contractId = contractId;
        this.name = name;
        this.termsAndConditions = termsAndConditions;
        this.executionStatus = executionStatus;
    }

    public int getContractId() {
        return contractId;
    }

    public String getName() {
        return name;
    }

    public String getTermsAndConditions() {
        return termsAndConditions;
    }

    public boolean getExecutionStatus() {
        return executionStatus;
    }
}

public class SmartContractManager {
    public SmartContract createSmartContract(String name, String termsAndConditions) {
        // Implement smart contract creation logic
    }

    public SmartContract getSmartContract(int contractId) {
        // Implement smart contract retrieval logic
    }

    public void executeSmartContract(SmartContract contract) {
        // Implement smart contract execution logic
    }
}
```

Transaction Management

Java

```java
public class Transaction {
    private int transactionId;
    private int walletId;
    private int assetId;
    private double amount;
    private Date timestamp;

    public Transaction(int transactionId, int walletId, int assetId, double amount, Date timestamp) {
        this.transactionId = transactionId;
        this.walletId = walletId;
        this.assetId = assetId;
        this.amount = amount;
        this.timestamp = timestamp;
    }

    public int getTransactionId() {
        return transactionId;
    }

    public int get
```

User

Develop tests including unit tests, integration tests, and system tests for the implementation.

AI Model

Here are some examples of tests you could write for the implementation:

Unit Tests

- Test that the User class correctly stores and retrieves user data
- Test that the Wallet class correctly stores and retrieves wallet data
- Test that the Asset class correctly stores and retrieves asset data
- Test that the SmartContract class correctly stores and retrieves smart contract data
- Test that the Transaction class correctly stores and retrieves transaction data

Integration Tests

- Test that the UserManager class can successfully create, retrieve, and update users
- Test that the WalletManager class can successfully create, retrieve, and update wallets
- Test that the AssetManager class can successfully deposit, transfer, and retrieve assets
- Test that the SmartContractManager class can successfully create, retrieve, and execute smart contracts
- Test that the TransactionManager class can successfully create, retrieve, and confirm transactions

System Tests

- Test that the system can successfully create a new user and wallet
- Test that the system can successfully deposit assets into a wallet
- Test that the system can successfully transfer assets between wallets
- Test that the system can successfully execute a smart contract
- Test that the system can successfully create and confirm a transaction

These tests should cover the main functionality of the system and ensure that it works as expected. Additionally, you may also want to consider writing tests for error handling and edge cases.

Here is an example of what a unit test for the User class might look like in Java:

Java

```
public class UserTest {
    @Test
    public void testUserConstruction() {
        User user = new User(1, "username", "password", "biometricData");
        assertEquals(1, user.getUserId());
        assertEquals("username", user.getUsername());
        assertEquals("password", user.getPassword());
        assertEquals("biometricData", user.getBiometricData());
    }
}
```

And here is an example of what an integration test for the UserManager class might look like in Java:

Java

```
public class UserManagerTest {
    @Test
    public void testCreateUser() {
        UserManager userManager = new UserManager();
        User user = userManager.createUser("username", "password", "biometricData");
        assertNotNull(user);
        assertEquals(1, user.getUserId());
    }
}
```

Note that these are just examples and you may need to modify them to fit your specific implementation. Additionally, you will want to write many more tests to cover all of the functionality of the system.