Loyalty Reward System

Table Of Contents

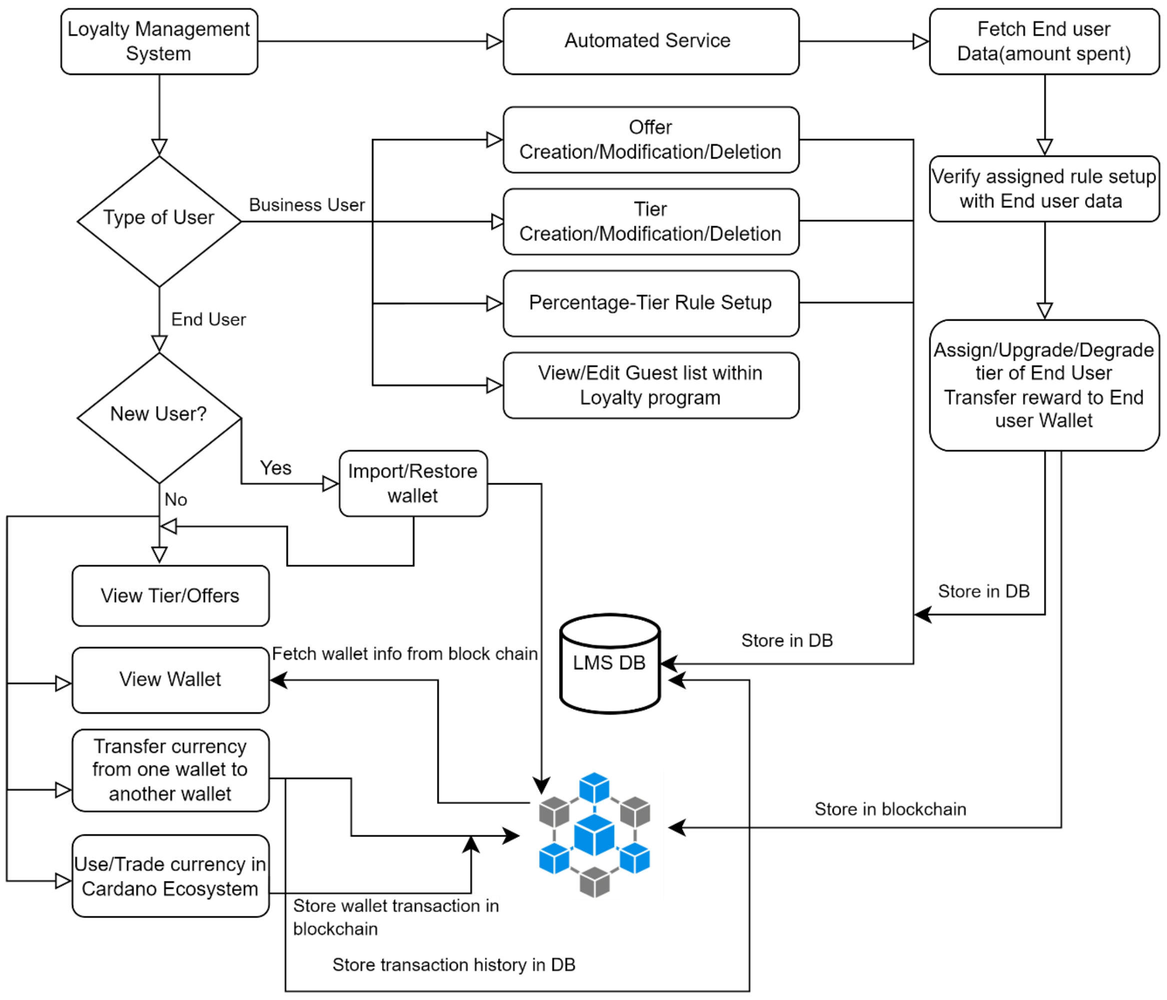
Milestone 1 -
UI Design and Tech Design

Loyalty Reward System as an Open Source for all Industry:

1. **User Interface (UI) Mockups Screen -**
    1.1. Figma Designs:
        1.1.1. UI Constrains - **https://www.figma.com/design/voH1plhjYjipGV9ILBdccl/LRS?node-id=0-1&p=f&t=gYgXvtuJE03OHtLm-0**

2. Technical Design

2.1. Data Flow Diagram:



Loyalty Management System
Automated Service
Fetch End user Data(amount spent)
Offer Creation/Modification/Deletion
Tier Creation/Modification/Deletion
Percentage-Tier Rule Setup
View/Edit Guest list within Loyalty program
Verify assigned rule setup with End user data
Assign/Upgrade/Degrade tier of End User Transfer reward to End user Wallet
Type of User
Business User
End User
New User?
Yes
No
Import/Restore wallet
View Tier/Offers
View Wallet
Transfer currency from one wallet to another wallet
Use/Trade currency in Cardano Ecosystem
Fetch wallet info from block chain
LMS DB
Store in DB
Store in DB
Store in blockchain
Store wallet transaction in blockchain
Store transaction history in DB

2.2. System Architecture Design:

2.2.1. Overview

The Loyalty Reward System is designed to handle user loyalty programs and build a strong relationship with customers, integrated with a blockchain for transparency and immutability. The architecture is divided into several layers to ensure scalability, security, and maintainability.

2.2.2. Presentation Layer

- **User Interface (UI)**: Web and mobile interfaces for users and administrators.
- **Web Interface**: Built using React.

2.2.3. Application Layer

- **Wallet Creation Service**: Create a wallet for interacting with blockchain.
- **Reward Creation Service**: Assign and manage rewards to users.
- **Tier Management Service**: Assign tier to customer and manage tier.
- **Reward Redemption Service**: Using the rewards by the customer.
- **User Profile Management Service**: Manages user profiles.
- **Admin Panel Service:** Setting up rules for LRS.
- **Authentication and Authorization Service**: Manages user authentication and access control.
- **API Gateway**: Facilitates communication between the UI and backend services, and with external integrated systems.

2.2.4. Integration Layer

- Any System/Entity of Business required the Loyalty Reward System for the Integration.

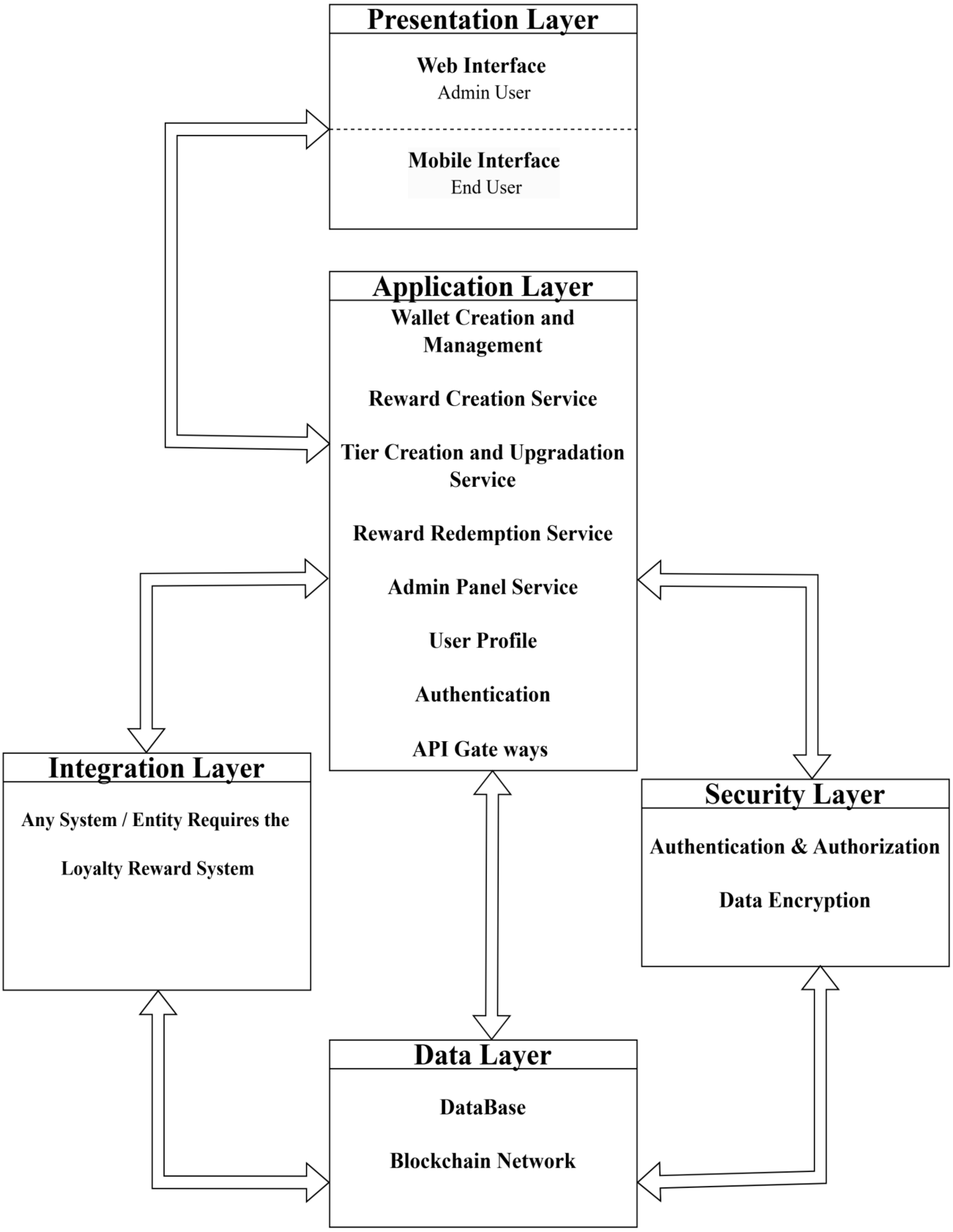
2.2.5. Security Layer

- **Authentication and Authorization**:
  Uses OAuth or JWT for secure user authentication.
- **Data Encryption**:
  Protects data in transit and at rest.
- **Audit Logs**:
  Records all system activities for monitoring and compliance.

2.2.6. Data Layer

- **Database**:
  Primary storage for user data, transaction, metadata, tier information, reward details.
- **Blockchain Network**:
  Cardano blockchain handles transaction data, reward redemption, tier management and reward generation as immutable proofs.

System Architecture Design



2.3. Technology Stack

2.3.1. Frontend

- React.js

2.3.2. Backend

- Node.js with Express.js for REST APIs
- JWT (JSON Web Tokens) for user authentication

2.3.3. Database

- MongoDB (NoSQL)

2.3.4. Notification Service

- SendGrid for email notifications

2.3.5. Table Design (Generic):

CardanoLRS/DesignDocs/Table_Design_Generic.xlsx at main · AIQUANT-Tech/CardanoLRS

2.3.6. API Design (Generic):

CardanoLRS/DesignDocs/API_Design_Hotel_Usecase.pdf at main · AIQUANT-Tech/CardanoLRS

2.4. Smart Contract Specifications:

Libraries to be incorporated in the Haskell File:

Libraries	Purpose	Key Features
Plutus.V2.Ledger.Api	Core data types and logic for Plutus scripts	Validator, Datum, Redeemer
Plutus.V2.Ledger.Contexts	Access transaction context for validation	ScriptContext, TxInfo
Plutus.V2.Ledger.Scripts	Compile and manage validator scripts	mkValidatorScript, Validator
PlutusTx	Compile Haskell code to Plutus Core and manage data types	compile, unstableMakeIsData
PlutusTx.Prelude	Provides arithmetic, logical operators, and debugging functions for Plutus Core compatibility	traceIfFalse, (+), div
Ledger	Interact with Cardano ledger, manage addresses and hashes	pubKeyHash, unPaymentPubKeyHash
Ledger.Typed.Scripts	Ensures type safety for smart contracts	TypedValidator, mkTypedValidator

The smart contract is designed to manage reward redemption and store the transaction details on the Cardano blockchain. It ensures transparency, immutability, and trust by implementing the following functionalities:

Overview of Smart Contract:

validateRedemption

- Role: This function validates the redemption of rewards by ensuring the user's balance has sufficient rewards. It also updates the user's rewards balance after a successful redemption.
- Explanation:
  - A user may request to redeemRewards (e.g., converting rewards to discounts or benefits).
  - The function first checks the user's current reward balance to ensure they have enough rewards to fulfill the redemption request.
  - When the user meets the criteria (e.g., rewards), the system proceeds to update their balance by deducting the redeemed rewards.
  - When the criteria are not met, the function will reject the transaction, ensuring the integrity of the reward system.
- Significance: This is a critical part of a loyalty reward system, ensuring fair use of rewards and protecting the system from invalid transactions.

validateTransaction

Role: Validates the transaction type and delegates to specific validation logic.

Explanation:

- Purpose: To manage multiple types of transactions in the system by ensuring that the correct validation logic is applied based on the transaction type.
- Workflow:

    - The function inspects the txType field of the transaction.
    - Based on the txType, it calls the appropriate validation function.
    - For example, if the txType is related to RewardGeneration, the function will call validateRewardGeneration to verify the validity of the reward-related transaction.
- Example Use Case:
    - A transaction with txType = "RewardGeneration" is processed.
    - The function delegates the validation to validateRewardGeneration, ensuring that the rewards are generated correctly and according to the business rules.
- Significance:
    - Centralizes transaction validation logic, ensuring maintainability and scalability.
    - Allows the addition of new transaction types without altering the core validation logic.

storeRewardDetails
Role: Tracks and updates the RewardGeneration process.
Detailed Explanation:

- Purpose: To record and manage the details of RewardGeneration transactions on the blockchain.
- Workflow:
    - A unique rewardId is generated for each reward transaction to ensure traceability.
    - The initial reward (rewardBalance) is recorded.
    - The updated reward balance is calculated by adding the new rewards to the existing balance.
    - This ensures that the reward balance reflects all previous and current transactions accurately.
- Example Use Case:
    - A user earns 50 new rewards.
    - **storeRewardDetails** calculates the updated balance by adding 50 to the user's current balance (e.g., 150 + 50 = 200).
    - These details, along with a unique rewardId, are stored on the blockchain.
- Significance:
    - Ensures transparency and accuracy in reward tracking.
    - Provides a detailed record of reward generation, which is crucial for auditing and resolving disputes.

getUserRewards

Role: Retrieves the user's current rewards balance.
Detailed Explanation:

- Purpose: To provide a mechanism for fetching the current rewards balance of a user from the blockchain.
- Current Implementation:
    - The function currently returns a placeholder value of 100.
    - This is a stub implementation and must be replaced with logic that interacts with UTxOs (Unspent Transaction Outputs) on the blockchain.
    - UTxOs are used to store rewards data in Cardano, and this function would query those UTxOs to determine the user's actual rewards balance.

- Example Use Case:
    - A user requests their current reward balance.
    - The function queries the blockchain's UTxOs associated with the user's address to fetch and return the correct balance.
- Significance:
    - Enables real-time retrieval of accurate rewards data.
    - Acts as a building block for user interfaces or APIs that display reward balances to users.

2.4.1 Auditability and Storage of Reward Redemption:

Purpose

To enable users to redeem earned rewards for products, services, or discounts, ensuring transparency, immutability, and fairness.

2.4.1.1 Data Structure:

2.4.1.1.1 RedeemRequest

```
Fields:
data RedeemRequest = RedeemRequest
   {
     userId :: BuiltinByteString  // Id of an user.
   , redeemReward :: Integer // Reward requested for redemption.
   , referenceId :: BuiltinByteString  // Id of service being redeemed.
  ,requiredReward:: Integer //Rewards required to redeem the services.
   , timestamp :: POSIXTime  // The time for redemption request.
   }
```

Key Functionality: The functionality of the RedeemRequest is as follows:

- Input Handling:
    - The inputs to the RedeemRequest smart contract are provided in the form of Datum when the transaction is constructed.
    - The Datum includes the RedeemRequest structure, populated with the relevant data fields (e.g., userId, redeemReward, referenceId, requiredReward, timestamp).
- Validation:
    - The smart contract will validate the following:
        1. User Eligibility:
           Verify that the userId is valid and exists in the system.
        2. Sufficient Rewards:
           Check if redeemReward is less than or equal to the available rewards for the user in the blockchain state.
        3. Satisfying Conditions:
           Ensure that redeemReward meets or exceeds the requiredReward for the specified referenceId.
- Blockchain Storage:
    - Once validated, the redemption request (along with all its fields) will be stored on the Cardano blockchain. This ensures immutability and traceability.

Explanation:

- Users provide userId, redeemReward, referenceId, requiredReward, and timestamp for redemption.
- The provided inputs are serialized into a Datum and attached to the blockchain transaction.
- The smart contract validates user eligibility, sufficient rewards, and required conditions for redemption.
- The validated RedeemRequest is recorded on the blockchain for transparency and immutability.
- The system updates reward balances and ensures seamless service delivery while maintaining traceability.

2.4.1.1.2 UserState

```
Fields:
data UserState = UserState
   {
     UserId :: BuiltinByteString  // Id of a user.
   , rewardBalance :: Integer  // Total available rewards for the user.
   }
```

Key Functionality: The UserState will function as the **redeemer** file in the smart contract. In the context of Plutus smart contracts, a **redeemer** is used to provide additional data that influences the behavior of the contract. Here's how it works in this case:

Input Handling:

- When a user initiates a redemption, their current state (represented by the UserState) is passed to the smart contract as a **redeemer**.
- The smart contract uses the UserState to:
    1. Verify the identity of the user (userStateUserId).
    2. Check the rewardBalance to ensure the user has sufficient rewards for the redemption request.

Validation:

1. **Identity Check**:
    - The userStateUserId is compared with the user ID in the Datum (e.g., in the RedeemRequest) to ensure that the correct user is making the redemption.
2. **Sufficient Balance**:
    - The rewardBalance is checked against the redeemReward field in the Datum.
    - Example:
        - When redeemReward = 50 and rewardBalance = 100, the transaction is valid.
        - When redeemReward = 150 and rewardBalance = 100, the transaction is invalid.

State Update:

The validation is successful:

- The contract will deduct the redeemed rewards (redeemReward) from the rewardBalance.
- A new UserState reflecting the updated balance will be recorded in the blockchain as part of the transaction.

Blockchain Interaction:

The UserState as a redeemer:

1. **Provides Inputs for Validation**:
    - The rewardBalance ensures that only valid redemptions are processed.
2. **Facilitates Accountability**:
    - By associating the userStateUserId with every redemption transaction, the system ensures traceability.

Explanation:

- The UserState as a redeemer allows the smart contract to validate reward balances in real-time during redemption.
- By linking userStateUserId with each transaction, unauthorized access and misuse are prevented.
- Storing UserState on the blockchain creates an immutable record of user reward balances and transactions.
- The contract processes only valid transactions, ensuring accuracy and minimizing errors.

2.4.1.2 Auditability Algorithm for Reward Redemption:

Here rewards Redemption depends on three factors and those are redeemRewards, rewardBalance and requiredReward.

redeemRewards are those rewards that we want to redeem.
rewardBalance are available rewards in the user account.
requiredReward are those rewards that will be redeemed for that service.

Logic:

```
// Logic for Reward redemption
        boolean sufficientBalance = (redeemRewards <= rewardBalance)
        boolean rewardMatch = (redeemRewards <= requiredReward)
```

Explanation: Users have an account and for each transaction, he/she will get some rewards or rewards. And that will be stored in **rewardBalance,** let’s say the available Reward is 50 in the account**.** Now he/she wants to buy some services or products and have some price, let's say 500 and there is an option of 470 + 30 rewards. This means a user buys that product using his/her 30 rewards. Here **rewardBalance** is 50, **requiredReward** is 30 and **redeemRewards** is also 30, or the user can define the amount. Now the logic says that if **redeemRewards** is less than or equal to **rewardBalance** then the user only redeems that reward, else user can not redeem. Logic will also check whether **redeemRewards** is less than or equal to **requiredReward,** if these two conditions are satisfied, then only a smart contract will execute.

```
// Update rewards Condition
        updateReward = (rewardBalance - redeemRewards) //Integer
        redeemRewards = updateReward
```

Explanation: After redeeming the rewards we are going to update the **rewardBalance.** For that we take an updateReward variable which calculates the remaining rewards after redeeming and that variable value is passed to the **redeemRewards**.

2.4.1.3 Storage (Onchain):

Here we are storing two fields. One is RedeemRequest and the other is UserState.
RedeemRequest Contains

```
data RedeemRequest = RedeemRequest
```

```
 {
  userId :: BuiltinByteString  // Id of an user.
  , redeemReward :: Integer // Reward requested for redemption.
  , referenceId :: BuiltinByteString  // Id of service being redeemed.
  , requiredReward :: Integer //Rewards required to redeem the services.
  , value :: Integer //Value of the services.
  , timestamp :: POSIXTime  // The time for redemption request.
  }
```

And UserState Contains

```
data UserState =  UserState
  {
    userId :: BuiltinByteString  // Id of an user.
   , rewardBalance :: Integer // Total available rewards for the user.
    }
```

2.4.1.4 Key Functions:

Function	Purpose	Explanation
validateRedemption	Validates the redemption process of rewards.	Ensures that the user's balance has enough rewards for redemption and updates the user's rewards correctly.
mkValidatorScript	Creates the validator script for the Plutus smart contract.	Combines the compiled validation logic into a validator script that can be used on the blockchain.
writePlutusScript	Writes the compiled Plutus script to a file.	Serializes and writes the Plutus script to a file at a given path.
writeScript	Handles the writing of the Plutus script file.	Calls writePlutusScript to write the script to the file system, ensuring necessary directories exist.
unsafeFromBuiltinData	Converts BuiltinData to Haskell data type, which is used in validation functions.	Helps to safely convert Plutus data (from blockchain transactions) into Haskell data types for validation and processing.
PlutusTx.unstableMakeIsData	Creates directories if they do not exist.	Ensures necessary directories are created before writing the Plutus script.
writeFileTextEnvelope	Writes the Plutus script text to an encrypted file with Cardano API.	Serializes and stores the validator script in a specific Cardano-compatible format.

displayError	Displays any error that occurs during the script-writing process.	Outputs human-readable error messages in case something goes wrong during script creation.

2.4.1.5 Smart Contract Implementation steps

1. **Write the Smart Contract**: Haskell code for validating and storing ratings.
2. **Deploy the Smart Contract**: Compile the contract using Plutus tools. Deploy the compiled script (.plutus file) to the blockchain.
3. **Submit Data**: Format the User ID, Redeem Rewards, Reference ID, Required Reward into a datum file (datum.json):

```
{
   "constructor": 0,
   "fields": [
      { "userId": "dXNlcl9pZA==" },
     {"redeemReward": 100},
      {"referenceId": "c2VyaWVfZGF0YQ==" },
     { "requiredReward": 150},
     {"timestamp": 1690575600},
   ]
}
```

4. Submit a transaction interacting with the smart contract:

```
cardano-cli transaction build \
  --tx-in <UTXO> \
```

5. **Query and Verify**: Use blockchain APIs and logs to retrieve and verify data.
6. **Data Structures of Smart Contract:**

   The Review data structure includes:

   - userId: A unique ID user.
   - redeemReward: Reward requested for redemption.
   - referenceId: Id of service being redeemed.
   - requiredReward: Rewards required to redeem the services.
   - timestamp: The time for a redemption request.

```
data RedeemRequest = RedeemRequest
   {
     userId :: BuiltinByteString  // Id of an user.
   , redeemReward :: Integer // Reward requested for redemption.
   , referenceId :: BuiltinByteString  // Id of service being redeemed.
   , requiredReward :: Integer //Rewards required to redeem the services.
   , timestamp :: POSIXTime  // The time for redemption request.
   }
```

2.4.2 Storing the Reward Generation, rewardId, rewardValue, and updatedValue:

Purpose

To record and store all loyalty-related transactions (reward generation, redemption) immutably on the blockchain. This ensures transparency, trust, and accountability in the Loyalty Reward System.

2.4.2.1 Data Structure:

2.4.2.1.1 TransactionRequest

```
data LoyaltyTransaction = LoyaltyTransaction
    { txType      :: TransactionType
    , user        :: PubKeyHash
    , amount      :: Integer
    , timestamp  :: POSIXTime
    }
```

Key Functionality:
Input Handling:

- The contract receives a LoyaltyTransaction as part of the TransactionRequest input.
- Users initiate transactions (e.g., earning rewards or redeeming them), and the corresponding data is formatted into this structure.

Validation:

1. Transaction Type Check:
    - Ensures txType is valid (e.g., Reward, Redemption).
2. User Verification:
    - Confirms the user (via their public key hash) is authorized to perform the transaction.
3. Amount Validation:
    - Verifies the amount is appropriate for the transaction type (e.g., sufficient reward for redemption).
4. Timestamp Validation:
    - Ensures the transaction's timestamp is within an acceptable time range.

Storing the Transaction:

- After successful validation, the LoyaltyTransaction is stored immutably on the blockchain.
- This creates a permanent and transparent record of the transaction.

Explanation:

- Blockchain-based recording ensures that all loyalty transactions are visible and verifiable.
- Immutable records prevent tampering and ensure the authenticity of transactions.
- Every transaction is linked to a specific user and time, promoting responsibility.
- Using cryptographic techniques ensures that transactions are secure from unauthorized access.
- Smart contracts handle transaction validation and storage automatically, reducing manual intervention.

2.4.2.2 Algorithm:

2.4.2.2.1 Validate Transaction Function:

```
validateTransaction :: LoyaltyTransaction -> ScriptContext -> Bool
validateTransaction tx ctx =
   case txType tx of
      RewardGeneration -> validateRewardGeneration tx ctx
```

2.4.2.2.2 Reward Generation:

```
storeRewardDetails :: LoyaltyTransaction -> ScriptContext -> Bool
storeRewardDetails ctx = do
   let rewardId = sha2_256 $ serialize tx  -- Generate a unique reward ID
      rewardValue = amount tx
      updatedValue = rewardValue + existingBalance  -- Assuming `existingBalance` is
retrieved from state
   pure (rewardId, updatedValue)
```

2.4.2.2.3 Validate reward generation:

```
validateRewardGeneration :: LoyaltyTransaction -> ScriptContext -> Bool
validateRewardGeneration tx ctx =
   let info = scriptContextTxInfo ctx
   in  traceIfFalse "User not authorized" (txSignedBy info (user tx)) &&
      traceIfFalse "Amount must be non-negative" (amount tx >= 0)
```

The validateRewardGeneration function and its associated task of storing rewardId, rewardValue, and updatedValue focus on validating Reward generation transactions and updating the blockchain state accordingly. Here's how it works:

Key Functionality:

- Input:
  - tx: Represents the loyalty transaction with fields like txType, user, amount, and timestamp.
  - _: Represents the script context, which can provide additional information if needed.
- Validation:
  - Ensures that the amount (reward reward generated) is non-negative (amount tx >= 0).
  - When negative rewards are generated, the function fails with a trace message.

Storing rewardId, rewardValue, and updatedValue:

After successful validation of the Reward generation, the following fields are stored in the blockchain:

- rewardId: A unique identifier for each reward generation event. This could be a hash or sequential ID generated based on the transaction.
- rewardValue: Represents the initial number of reward reward generated.
- updatedValue: Tracks the updated balance after the transaction (i.e., after Reward generation).

Storage Process:

1. Reward Generation:
   - Upon successful validation, a rewardId is created (e.g., through hashing or sequence).
   - rewardValue is recorded as the initial reward generated.
   - updatedValue is computed by adding rewardValue to the existing balance for that user (stored in a separate state).

Functionality Flow:

1. Reward Generation Transaction:
   - A user requests reward reward (amount tx), which is validated to ensure it's a non-negative value.

2. Validation:
    - validateRewardGeneration checks if the generated reward is positive. When valid, the transaction proceeds.
3. Storing Details:
    - After validation, the following data is stored on the blockchain:
        - rewardId: A unique identifier.
        - rewardValue: The initial number of rewards generated.
        - updatedValue: The new balance after adding the generated reward.

Explanation:

- Ensures only positive rewards are generated to maintain transaction validity.
- rewardId uniquely identifies each reward generation event to prevent duplication.
- rewardValue records the initial generated reward for future reference.
- updatedValue reflects the user's new reward balance after rewards generation.
- Stores immutable details on the blockchain for transparent, traceable transactions.

2.4.2.3 Key Functions:

Function Name	Purpose	Explanation
validateTransaction	Validates the transaction type and delegates to specific validation logic.	It checks the txType of the transaction and calls the corresponding validation function: validateRewardGeneration.
storeRewardDetails	Store rewardId, rewardValue, and updatedValue	This function generates a unique rewardId, records the initial reward generated, and calculates the updated reward balance by adding the new rewards to the existing balance. This ensures that reward generation transactions are tracked and updated correctly on the blockchain.
getUserrewards	Retrieves the user's current rewards balance.	A placeholder function that always returns 100. This must be replaced with logic to fetch the user's rewards from UTxO on the blockchain.
validateRewardGeneration	Validates the RewardGeneration transaction type.	Ensures that the transaction is authorized and has a valid, non-negative reward amount before storing it on the blockchain.
validator	Defines the on-chain validation script.	Compiles the validateTransaction logic into a Validator script that is deployed to the Cardano blockchain for transaction validation.

2.4.2.4 Interaction of Smart Contract with Blockchain:

1. Submission:

1. Frontend collects data like redeemRewards, rewardBalance, requiredReward, RewardGeneration.
2. The system sends this data to **Plutus Smart Contract**.
3. The smart contract acts as a **validator** to ensure the integrity of the submitted data.

2. Smart Contract Validation:

1. Smart Contract validates the incoming data whether it satisfies the condition or not.
2. When validation passes the data is included in a transaction to be recorded on blockchain.

3. Blockchain Storage:

1. Once validated the transaction is **signed** and **submitted** to the cardano blockchain.
2. The blockchain stores **Reward Redemption**, **Reward Generation**.

4. Confirmation:

After successfully storing, the blockchain generates a transactionId which helps to verify the data and helps to retrieve the data from the blockchain

2.4.2.5 Data Flow:

1. User Interaction

- **Input Data**: The user provides details necessary for redeeming their rewards:
  rewardBalance (Current Rewards available for redemption)
  redeemReward (Rewards to be redeemed)
  requiredReward(Rewards required to redeem the services)
  userId (Unique identifier for the user)
  referenceId

2. Source of Data

- **Data Collection**:
  1. The **userId** and **rewardBalance** are retrieved from the blockchain's state for the user.
  2. The **redeemReward** and **referenceId** are provided by the user via the front-end interface.

3. How the Data is Processed

- **Gathering Inputs**:

  Data is collected as follows: {
  userId: "USR123"
  rewardBalance: 500
  redeemReward: 100
  requiredReward: 100
  referenceId: "ORDER56789"
  }

- **Formatting**:

  The collected data is formatted into a **datum** structure to send to the smart contract.

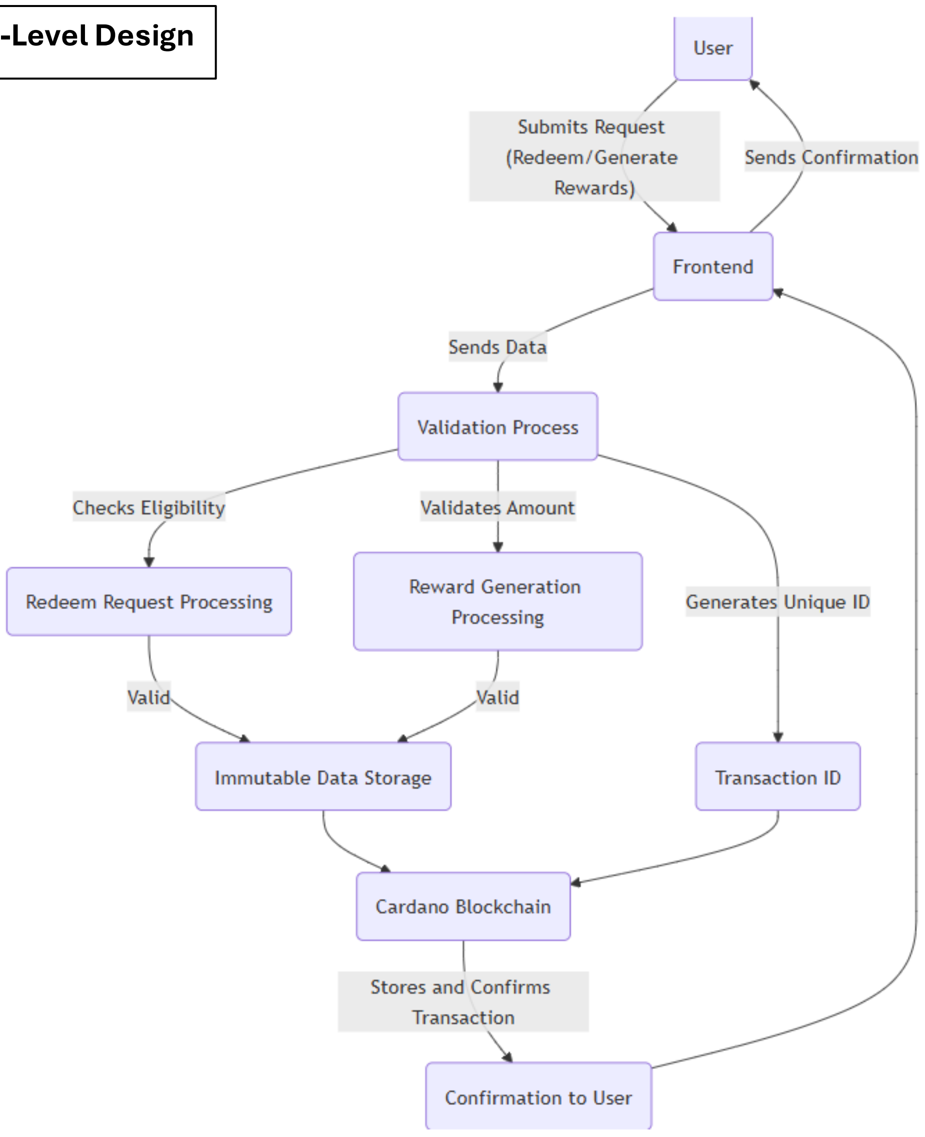
4. Validation and Execution by Smart Contract

- **Validation Logic**:

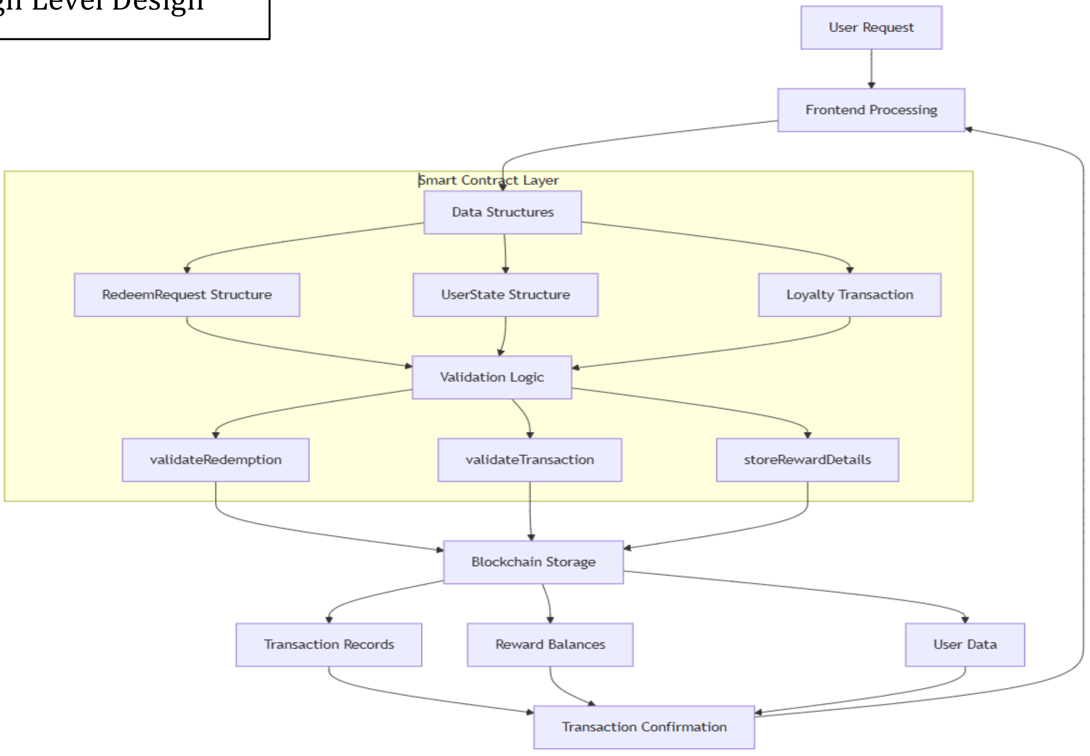
  The smart contract performs the following checks:
  Verifies **redeemReward <= rewardBalance**.
  Ensure **requiredReward <= redeemReward**

- **State Update:**
  1. When validation passes, the smart contract deducts **redeemReward** from the user's **rewardBalance** and stores the new balance on the blockchain.
  2. A record of the redemption is logged in the transaction.

Low-Level Design



High Level Design



Integration

- **APIs**: Develop APIs to facilitate communication between the front-end application and the smart contract.
- **Middleware**: Use middleware to handle data processing and ensure smooth integration on-chain.

2.5. Test Plan:

Test Plan for Loyalty Reward System

2.5.1. Introduction

The purpose of this test plan is to outline the testing strategy for the Loyalty Reward System (LRS). This system has different user types: Business Users, End Users, and Automated Services. The goal is to ensure that all features are functioning as expected and meet the business requirements.

2.5.2. Test Objectives

- Verify that all functionality for Business Users, End Users, and Automated Services works as intended.
- Validate that the system adheres to security, performance, and usability requirements.
- Ensure all tier and offer management processes are robust, including mapping tiers to offers and ensuring the correct application of percentage rules.
- Test wallet functionalities, including transactions and transfers, for End Users.

2.5.3. Scope

In-Scope:

- Business User functions:
  - Login and logout.
  - Create, modify, and delete offers.
  - Create, modify, and delete tiers.
  - Map tiers with offers.
  - Set up percentage-based rules and associate them with tiers.
  - View and edit End User information, including tier assignments.
- End User functions:
  - Login and logout.
  - View own tier, offers, and Cardano wallet.
  - Transfer currency between wallets.
  - Use or trade with wallet currency.
- Automated Service:
  - Fetch end user and tier details.
  - Match end users' tier with rules setup.
  - Upgrade, downgrade, or assign tiers to end users based on rules.

Out-of-Scope:

- External systems integrations (e.g., third-party wallets, third-party APIs for currency exchange).
- Security and penetration testing (unless explicitly requested).

2.5.4. Test Environment

- **Business Dashboard**: Business access with valid credentials.
- **End User Portal**: End User access with valid credentials.
- **Automated Service**: Mock automated service environment to simulate tier assignments.
- **Backend Database**: Verify the data consistency between the database and user interface.

2.5.5. Test Scenarios and Cases

Business User Test Cases:

Test Case ID	Description	Steps	Expected Result
TC-01	Login as Business User	1. Navigate to the Business login page. 2. Enter valid business credentials. 3. Click "Login".	Business successfully logs into the system.
TC-02	Create Offer	1. Login as Business. 2. Navigate to the "Offers" section. 3. Click "Create New Offer". 4. Enter valid offer details. 5. Save.	New offer is created and listed in the offer section.
TC-03	Modify Offer	1. Login as Business. 2. Select an existing offer. 3. Modify details. 4. Save changes.	Offer is updated successfully with the new details.
TC-04	Delete Offer	1. Login as Business. 2. Select an existing offer. 3. Click "Delete". 4. Confirm deletion.	Offer is deleted from the system and no longer visible.
TC-05	Create Tier	1. Login as Business. 2. Navigate to "Tier Management". 3. Click "Create New Tier". 4. Enter valid tier details. 5. Save.	New tier is created and listed in the tier section.
TC-06	Modify Tier	1. Login as Business. 2. Select an existing tier. 3. Modify tier details. 4. Save changes.	Tier is updated successfully with new details.
TC-07	Delete Tier	1. Login as Business. 2. Select an existing tier. 3. Click "Delete". 4. Confirm deletion.	Tier is deleted from the system. If it is mapped to offers, the system should provide a warning.
TC-08	Map Tier with Offer	1. Login as Business. 2. Navigate to "Map Tier to Offer". 3. Select tier and offer. 4. Map and Save.	Tier is successfully mapped with the offer, and visible in mapping section.
TC-09	Set Percentage Rule and Map with Tier	1. Login as Business. 2. Navigate to "Rules". 3. Set up percentage-based rules (e.g., 10% off). 4. Map rule to tier.	Percentage-based rule is successfully created and mapped to the tier.
TC-10	View End User Info	1. Login as Business. 2. Navigate to "End User Management". 3. View details of end users.	End User details are displayed correctly, including assigned tiers, offers, and wallet balance.
TC-11	Edit Tier of End User	1. Login as Business. 2. Navigate to "End User Management".	End user's tier is updated as expected.

Test Case ID	Description	Steps	Expected Result
		3. Select an end user. 4. Edit tier.	

End User Test Cases:

Test Case ID	Description	Steps	Expected Result
TC-12	Login as End User	1. Navigate to the End User login page. 2. Enter valid user credentials. 3. Click "Login".	End user successfully logs into the system.
TC-13	View Own Tier and Offers	1. Login as End User. 2. Navigate to "My Profile". 3. View the current tier and available offers.	End user successfully views own tier and offers.
TC-14	View Own Cardano Wallet	1. Login as End User. 2. Navigate to "Wallet". 3. View wallet balance and details.	End user successfully views own Cardano wallet information, including balance.
TC-15	Transfer Currency from Wallet	1. Login as End User. 2. Navigate to "Wallet". 3. Select "Transfer Currency". 4. Enter recipient details and transfer amount.	Currency is transferred successfully to another wallet.
TC-16	Use or Trade with Wallet Currency	1. Login as End User. 2. Navigate to "Wallet". 3. Choose an option to use or trade currency.	End user successfully uses or trades wallet currency as expected.

Automated Service Test Cases:

Test Case ID	Description	Steps	Expected Result
TC-17	Fetch End User and Tier Details	1. Automated service triggers a fetch request. 2. Fetch end user data and their tier information.	End user data and tier details are fetched correctly from the database.
TC-18	Match End User Tier with Rule Setup	1. Automated service triggers rule matching logic. 2. Match user tier with the setup rules.	Automated service matches the correct rule with the end user's tier and updates the tier if necessary.
TC-19	Upgrade/Downgrade Tier Based on Rules	1. Automated service triggers tier upgrade or downgrade based on rules. 2. Update end user's tier.	Tier is updated based on the rule evaluation (upgrade or downgrade).

Test Case ID	Description	Steps	Expected Result
TC-20	Assign Tier to Users with No Tier Setup	1. Automated service identifies users without tier setup. 2. Assign an appropriate tier based on rules.	Users without a tier are assigned a tier correctly based on available rules.

2.5.6. Test Environment

- **Test Server**: Development/Testing server environment with the latest version of the application.
- **Database**: The test environment database should mirror productioLetn for realistic testing.
- **Browsers**: Test across modern browsers (Chrome, Firefox, Safari, Edge).
- **Devices**: Desktop, mobile (iOS and Android) testing.

2.5.7. Test Data

Test data will be created as follows:

- **Business Users**: Valid business credentials for login and access.
- **End Users**: Multiple end users with varying tier information, offers, and wallet details.
- **Offer Data**: A set of offers with associated percentages, discounts, and product mappings.
- **Tier Data**: Different tiers with varying criteria and rules.
- **Percentage Rules**: Various percentage-based rules linked to specific tiers.

2.5.8. Test Execution Strategy

Manual Testing:

- Test cases involving UI interactions like offer creation, tier management, and wallet activities will be executed manually to verify user flow, data consistency, and system behavior.

Automated Testing:

- Automated tests will

be written for the Automated Service interactions (fetching end users, matching rules, upgrading/downgrading tiers) using appropriate testing frameworks.

- Regression testing for business-critical flows will be automated where possible.

2.5.9. Risk and Mitigation Plan

Risk	Mitigation Strategy
Data inconsistency	Ensure that all database updates are atomic and validated during testing.
Performance degradation	Conduct performance testing on wallet transactions and automated services.
Security vulnerabilities	Review access control, authentication, and data encryption to ensure security policies are applied correctly.

2.6. Deployment Plan:

This deployment plan outlines the setup for a Loyalty Reward System along with Blockchain for decentralized loyalty management. Initially, the deployment will be on an on-premises server with containerization for efficient scalability and management. The system will have future scalability for migration to the cloud (e.g., AWS, Azure, Google Cloud). Additionally, the APIs will be open sourced for public use.

2.6.1. System Overview

The key components include:

- **Frontend**: React.js (for dynamic and responsive UI)
- **Backend**: Node.js with Express.js (for API services)
- **Database**: MongoDB (for non-relational data storage)
- **Blockchain**: Cardano Blockchain, Smart Contract – Haskell & Plutus, IPFS, MeshJS, Lucide.

2.6.2. System Requirements

Hardware:

- On-premises servers (or virtual machines) for hosting the system initially.
- Minimum specifications for the server:
    - CPU: 4+ cores (recommended)
    - RAM: 16GB or more
    - Disk Space: 100GB+ (SSD for fast data access)
    - Network: Stable connection for internal and external access.

Software:

- **Operating System**: Linux (Ubuntu or CentOS recommended) or Windows Server.
- **Docker**: For containerization of the application.
- **Blockchain**: Cabal, GHC, Cardano Node & Cardano CLI
- **CI/CD Tools**: Jenkins, Git, CI Actions for automation.
- **Monitoring**: Prometheus for system health and resource utilization.
- **Reverse Proxy**: Nginx for routing and load balancing (if scaling horizontally).

2.6.3. Containerization

- Docker will be used to containerize both the frontend and backend services, making the system portable and scalable.

  Steps:

1. **Dockerfile :**
    - A Dockerfile will be created to build a Docker image for the React frontend. This will include installing dependencies and serving the app.
    - A Dockerfile for the Node.js backend will be created for running the Express API and connecting to MongoDB and the blockchain network.
    - Docker Compose will be used to manage multi-container applications, such as the frontend, backend, and MongoDB.
2. **Docker Compose**:
    - Docker Compose will be used to manage multi-container applications, such as the frontend, backend, and MongoDB.
3. **Blockchain Node**:
    - Use an official or community-maintained Cardano node image.
    - For Smart Contract the Template required from the existing template to build the .plutus file and use the cBorHeX from the file to show output.

2.6.4. Deployment on On-Premises Server

1. **Set up Server:**
    - Install Docker and Docker Compose on the on-premises server.
    - Ensure the server has access to the internet (if public blockchain is used).
2. **Database Setup:**
    - Use MongoDB cluster or use a Dockerized version of MongoDB.

3. **Blockchain Setup:**
    - GHC – 8.10.7
    - Cabal – 3.8.1.0
    - Cardano Node (Conway era)
    - Deploy the Smart Contract
        - Blockchain: Cardano Blockchain that supports Plutus.
        - Deployment: Deploy the smart contract to the blockchain.
4. **Configure Environment Variables:**
    - Set environment variables for API keys, MongoDB connections, blockchain configurations, etc., in the backend.
5. **Build and Deploy:**
    - Use Docker Compose to spin up the containers:
    - Verify that the React frontend, Node.js backend, MongoDB, and blockchain are all running smoothly.

2.6.5. CI/CD Setup

- **Source Code Repository:**
    - o Host the code on GitHub.
    - o Make APIs open source by providing access to the repository.
- **CI/CD Pipeline:**
    - o Set up a CI/CD pipeline with GitHub Actions Jenkins for automatic build and deployment.
    - o Automated tests should be implemented for both the frontend and backend.
    - o Docker images should be built, tested, and deployed to staging and production environments.

2.6.6. Security and Monitoring

- **Blockchain Security:**
    - Ensure that smart contracts are thoroughly tested and audited for security vulnerabilities.
    - Use HTTPS and secure communication between services.
- **API Security:**
    - Use JWT (JSON Web Tokens) for secure authentication and authorization.
    - Rate-limit API calls to prevent abuse.
- **Monitoring:**
    - Use Prometheus to monitor container health, API response times, and blockchain interactions.
    - Set up alerts for system failures or downtime.

2.6.7. Future Scalability to Cloud

Once the system is successfully running on-premises, the following steps can be followed to migrate to a cloud environment:

- **Container Orchestration:**
    - o Use Kubernetes to manage containerized services, which will be beneficial when scaling the system to multiple nodes.
- **Cloud Database:**
    - o Move MongoDB to a cloud-managed service (e.g., MongoDB Atlas).
- **Cloud Blockchain Nodes:**
    - o Move the blockchain nodes to a cloud provider or leverage a cloud-based blockchain service.

- **CI/CD Pipeline:**
    - Set up pipelines for deployment on cloud environments (e.g., AWS, Google Cloud, or Azure).

2.6.8. Maintenance and Updates

- Regularly update the application with new features, bug fixes, and security patches.
- Ensure blockchain smart contracts are updated as required.
- Regular monitoring and scaling based on usage patterns and performance metrics.

\

3. Loyalty Reward System Integrated with Hotel Booking System

3.1. Introduction

The purpose of this integration plan is to outline the strategy for integrating the Loyalty Reward System (LRS) with an external Hotel Booking System (HBS). This integration will allow customers to earn loyalty rewards, redeem offers, and manage their loyalty tier while interacting with the hotel booking system. The integration will support seamless data exchange between the two systems to ensure a smooth user experience and efficient backend processes.

3.2. Objectives

The primary objectives of this integration are:

- **Loyalty Rewards Earning and Redemption**: Customers should earn loyalty rewards when making bookings through the hotel system and be able to redeem them for discounts or upgrades.
- **Tier Management**: Customers' loyalty tiers (e.g., Bronze, Silver, Gold) should be synchronized between the LRS and HBS, based on their activity and booking history.
- **Seamless User Experience**: Integration should ensure that the user experience remains consistent and seamless, with loyalty benefits visible and accessible during the booking process.
- **Real-Time Data Sync**: Both systems should communicate in real-time for important actions like booking creation, loyalty rewards accrual, and tier upgrades.
- **Offer Visibility and Application**: Ensure that loyalty offers (e.g., discounts, upgrades) tied to specific tiers are applied automatically during hotel bookings.

3.3. Functional Overview of Integrated System

3.3.1. Hotel Booking System (HBS) Features

The HBS serves as the platform for managing bookings, reservations, and guest details. Key functionalities include:

- Reservation Management (Bookings, Check-in, Check-out)
- Guest Profile Management (Name, Email, Stay Dates)
- Room Availability Management

3.3.2. Loyalty Reward System Features

The LRS allows for the guest tier and reward management with offers. Key features include:

- Guest data Collection
- Configuration of tier, offer and reward rule
- Tier assignment to guest as per loyalty rule setup

3.3.3. Integration Goals

The integration aims to connect the two systems seamlessly, enabling:

- Automatic sending of guests’ info after payment from the Hotel Booking System.

- Real-time synchronization of booking and guest information between both systems.
- The ability for hotel staff (Business User) to tier and reward configuration from within the Hotel Booking System interface.
- Automated service to upgrade/downgrade tiers as per guest data.

3.4. Integration Process

3.4.1. Data Flow Overview

The integration will enable data to flow between the Hotel Booking System and the Loyalty Reward System through APIs or a middleware layer.

Pre data flow setup:

- HBS will provide own reservation currency, blockchain currency, wallet address to LRS integration table HBS_MAST.
- HBS will provide all existing guests and business users to LRS using an API for initial integration.

1. **Payment Confirmation**: Once a guest completes a payment (or a set of guests at configurable schedule as per HBS) in the HBS, their booking information (name, email, total payment, currency) is sent to the LRS.
2. **Payment Verification**: LRS will verify the guest's currency is matching with hotel currency. If not, guest payment will be converted to hotel currency.
3. **Verification with Reward rule setup**: Verify assigned rule setup with guests' payment (current one + past payment within assigned tenure).
4. **Tier Assignment**: LRS will upgrade or degrade the tier of guest as per Reward rule setup.
5. **Business User Dashboard:** Hotel stuff can see upgraded or degraded tier setup of guest from their guest details view dashboard.

3.4.2. Integration Components

API Integration

- **Hotel Booking System API**: The hotel booking system will invoke an API to share guest details (e.g., guest email, revenue) with LRS.
- **LRS System API**: The LRS will expose an API for retrieving and guest details and assign tier and reward as per the revenue rule.
- **Middleware (if required)**: A middleware or connector can be used to facilitate communication between the systems if direct API integration isn't feasible.

Data Syncing

- **Automated Data Syncing**: HBS will send all guest data, tier info(if already integrated with any loyalty system).

3.5. Technical Requirements

3.5.1. Database Schema:

The database schema will need to integrate guest, revenue data between the two systems. Below are suggested table structures for both the Hotel Booking System (HBS) and Loyalty Reward System (LRS):

CardanoLRS/DesignDocs/Table_Design_Hotel_Usecase.xlsx at main · AIQUANT-Tech/CardanoLRS

3.5.2. API Specifications:

For seamless integration, both systems will communicate using RESTful APIs. Below is the API design:

CardanoLRS/DesignDocs/API_Design_Hotel_Usecase.pdf at main · AIQUANT-Tech/CardanoLRS

3.6. Data Flow

Data Flow and Interaction

3.6.1. User Registration / Authentication

- **User Actions**: A new user registers on the Hotel Booking System (HBS).
- **Integration Flow**:
  - The HBS sends the user data (name, email, etc.) to the LRS for creating a corresponding loyalty profile.
  - The LRS responds with the initial loyalty tier and reward percentage balance (e.g., 0 reward percentage, basic tier).

Outcome: A user can log into LRS system from HBS dashboard, and their loyalty profile is automatically synced.

3.6.2. Booking Process

- **User Actions**: A customer books a hotel room on the HBS.
- **Integration Flow**:
  1. HBS sends booking details (price, booking date, user details) to the LRS API to calculate loyalty reward percentage earned.
  2. LRS calculates the reward percentage based on predefined rules and returns the reward percentage to the HBS.
  3. HBS updates the user's booking details with loyalty reward percentage and tier information.

Outcome: The user earns loyalty reward percentage for their booking, and their tier may be updated based on the new reward percentage balance.

3.6.3. Applying Loyalty Reward percentage

- **User Actions**: A customer opts to use loyalty reward percentage for wallet credit
- **Integration Flow**:
  1. HBS sends the redemption request with reward percentage details (e.g., how much reward percentage the user wants to redeem) to the LRS.
  2. LRS validates the request (sufficient reward percentage, eligible redemption options).
  3. LRS sends back a response (successful redemption or error) and the updated loyalty reward percentage balance.
  4. HBS applies the discount to the booking and updates the final price.

Outcome: The user's booking total is adjusted based on the loyalty reward percentage redeemed, and the remaining reward percentage balance is updated in both systems.

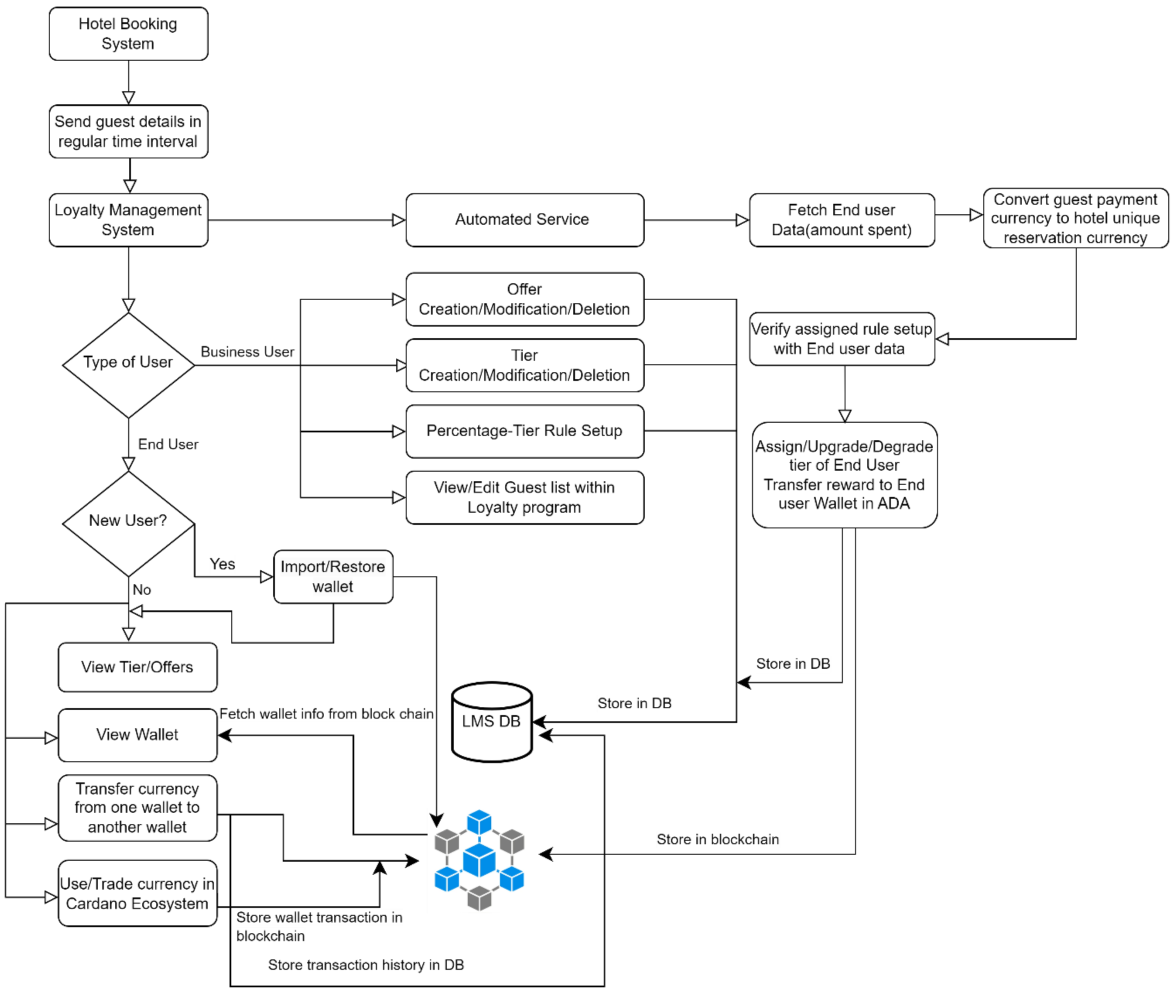
3.6.4. Tier Upgrade / Downgrade

- **User Actions**: A customer makes multiple bookings over time, earning enough reward percentage to qualify for a higher loyalty tier.
- **Integration Flow**:
  1. After each booking, HBS sends booking details to the LRS for rewards accumulation.
  2. LRS evaluates whether the accumulated reward percentage is sufficient for a tier upgrade (or downgrade).

3. If the tier changes, the LRS sends the updated tier information back to HBS.
4. HBS reflects the updated tier status on the user's profile.

Outcome: The user's loyalty tier is updated in both systems, ensuring that the user has access to the correct offers and benefits based on their tier.

3.6.6. Integration Data Flow:


Hotel Booking System
Send guest details in regular time interval
Loyalty Management System
Automated Service
Fetch End user Data(amount spent)
Convert guest payment currency to hotel unique reservation currency
Offer Creation/Modification/Deletion
Tier Creation/Modification/Deletion
Percentage-Tier Rule Setup
View/Edit Guest list within Loyalty program
Verify assigned rule setup with End user data
Assign/Upgrade/Degrade tier of End User Transfer reward to End user Wallet in ADA
Type of User
Business User
End User
New User?
Yes
No
Import/Restore wallet
View Tier/Offers
View Wallet
Fetch wallet info from block chain
LMS DB
Store in DB
Store in DB
Transfer currency from one wallet to another wallet
Use/Trade currency in Cardano Ecosystem
Store in blockchain
Store wallet transaction in blockchain
Store transaction history in DB

3.7. Security Considerations

- **Authentication**: All API endRewards should be secured with OAuth 2.0 or another secure method of authentication.
- **Data Validation**: Both systems must validate data inputs to prevent SQL injection, cross-site scripting (XSS), and other vulnerabilities.
- **Encryption**: All sensitive data, especially guest information, should be encrypted at rest and in transit (e.g., using HTTPS and AES encryption).
- **Compliance**: Ensure all data handling complies with regulations such as GDPR or any other relevant privacy laws.

3.8. Component

To implement this integration, a Scheduler component is required:

- **Purpose**: The scheduler will run periodically at defined intervals to collect guest booking data and send it to LRS.
- **Trigger Conditions**:

- **On Payment Completion**: Whenever a payment is completed for a booking, the scheduler captures this and immediately sends the relevant guest data to the LRS.
- **Time-based Interval**: On a regular interval (e.g., daily), the scheduler will send all the guest info (or a specific batch) to the LRS.

Scheduler Characteristics:

- **Interval Configuration**: The scheduler should be configurable to run at a specific time interval, such as hourly, daily, or weekly.
- **Data Collection**: The scheduler will collect relevant guest information for each reservation that has a completed payment or has occurred since the last scheduled run.

Types of Information Sent:

- **Guest Information**: Guest's name, email, loyalty ID, and contact details.
- **Reservation Details**: Booking date, check-in/out dates, price paid, etc.
- **Payment Information**: Total payment amount, payment method, and payment status.

3.9. Test Plan:

Summary: The purpose of the integration test plan is to ensure the successful communication and data flow between the Hotel Booking System (HBS) and the Loyalty Reward System (LRS). This test plan will focus on validating the integration Rewards, ensuring that data is transmitted correctly, and ensuring both systems behave as expected when interacting with each other.

3.9.1. Test Strategy:

3.9.1.1. Scenario based test plans:

Test Scenario	Test Case	Preconditions	Steps	Expected Results
Scenario 1: Send Payment Data from HBS to LRS	**Test Case 1.1: Valid Payment Completed**	A valid booking with an active payment gateway.	1. Make a payment for a booking. 2. Check if payment data is sent to LRS. 3. Validate loyalty Rewards update.	Payment details, guest ID, booking ID, payment amount are correctly sent to LRS.
	Test Case 1.2: Payment Data Missing/Incomplete	Payment with missing or incomplete details.	1. Simulate a payment failure or incomplete payment. 2. Observe API handling.	Error logged. No incorrect data sent to LRS. No update in LRS. Business user notified of the error.
Scenario 2: Send Batch Data on Regular Interval	**Test Case 2.1: Scheduler Sends Batch Data**	Guest reservations completed within the interval period.	1. Trigger the scheduler to send batch data (e.g., daily). 2. Verify if guest data is sent to LRS.	LRS receives all booking data for the period. Loyalty rewards are correctly updated.
Scenario 3: Tier Updates	**Test Case 3.1: Tier Upgrade Based on Loyalty Rewards**	User has accumulated enough Rewards for an upgrade.	1. Send booking and payment data to LRS. 2. Verify tier upgrade.	User’s tier is upgraded based on Rewards earned (e.g., Silver to Gold).
	Test Case 3.2: Tier Downgrade Due to	User has insufficient Rewards for current tier.	1. Send data with rewards deduction to LRS.	User’s tier is downgraded based on Rewards balance (e.g., Gold to Silver).

Test Scenario	Test Case	Preconditions	Steps	Expected Results
	Insufficient Rewards		2. Verify tier downgrade.	
Scenario 4: Offer and Redemption Integration	**Test Case 4.1: Available Offers Based on Tier & Rewards**	User has enough Rewards for offer eligibility.	1. Query LRS for available offers. 2. Verify if offers match the user’s tier and Rewards balance.	Correct offers are sent to HBS based on the user's Rewards and tier.
	Test Case 4.2: Redeem Loyalty Rewards	User has enough Rewards for redemption.	1. Redeem loyalty rewards during booking. 2. Verify if Rewards are deducted.	Correct amount of rewards is deducted, and redemption is applied to booking.
Scenario 5: Error Handling and Logging	**Test Case 5.1: API Failure During Data Transmission**	API failure scenario (e.g., timeout, invalid credentials).	1. Simulate an API failure. 2. Observe how the system handles the failure (retry mechanism or logging).	System retries the failed API call or logs the error. Admin receives an alert notification.
	Test Case 5.2: Missing Guest or Booking Information	Missing guest ID or booking data in the API request.	1. Trigger payment event with missing data. 2. Observe how the system handles it.	Error is logged, and no incorrect data is sent to LRS. A notification is sent to admins.
Scenario 6: Security Testing	**Test Case 6.1: API Authentication Failure**	Invalid API credentials (token or key).	1. Attempt to send data with invalid credentials. 2. Verify the response.	The request is rejected with an "Unauthorized" message.
	Test Case 6.2: Data Encryption and Security	Secure HTTPS connection enabled.	1. Verify that data is transmitted securely between HBS and LRS.	All data is encrypted using SSL/TLS encryption during transmission.

3.9.1.2. Prototype Testing on Testnet

Objective: Ensure that after checkout, the user can claim loyalty Rewards, and these Rewards are recorded on the blockchain as immutable data.

Tests:

1. **Verify Rewards Submission to Blockchain:**
   - ○ Ensure that users can claim loyalty Rewards after checkout.
   - ○ Verify that a transaction is recorded on the blockchain testnet with the booking ID as rewardReferenceId.
   - ○ Confirm that the transaction metadata matches the loyalty reward data (UserId, referenceId, redeemReward,requiredReward and timestamp).
2. **Data Integrity Validation:**

    - Verify that loyalty reward data such as referenceId, redeemReward, requiredReward cannot be modified once submitted to the blockchain testnet.
3. **Loyalty Rewards Display:**
    - Ensure that the loyalty Rewards displayed in the **Hotel Booking System** match the data stored on the testnet blockchain.
4. **Immutable Transactions:**
    - Verify that no loyalty Rewards transaction can be deleted once it has been recorded on the testnet Cardano blockchain.

3.9.1.3. User Acceptance Testing (UAT)

Objective: The goal of this UAT is to validate the functionality, usability, and performance of the **Loyalty Reward System** within real-world scenarios. The testing will be conducted across different hotels/villas for each, focusing on loyalty Rewards allocation, redemption, and balance tracking.

Tests:

Scope:

- Test scenarios will involve the hotels/villas with guests each, across different Spends.
- Ensure a comprehensive validation of the Loyalty Reward System, including Rewards allocation for each hotel after payments are completed.

Loyalty Rewards Allocation

- Ensure loyalty Rewards are awarded to guests-based Purchase.
- Validate that Rewards are calculated accurately based on predefined rules (e.g., Rewards per spent).

Rewards Redemption

- Test the redemption process by allowing guests to redeem Rewards for specific rewards during their stay.
- Verify that redeemed Rewards are deducted accurately from the guest's balance on the blockchain.

Balance Display

- Ensure that the guest's loyalty Rewards balance is displayed accurately across the booking system and matches blockchain data.

Reward Validation

- Verify that redeemed rewards correspond to the claimed items and are immutable on the blockchain testnet.