Review and Reputation System

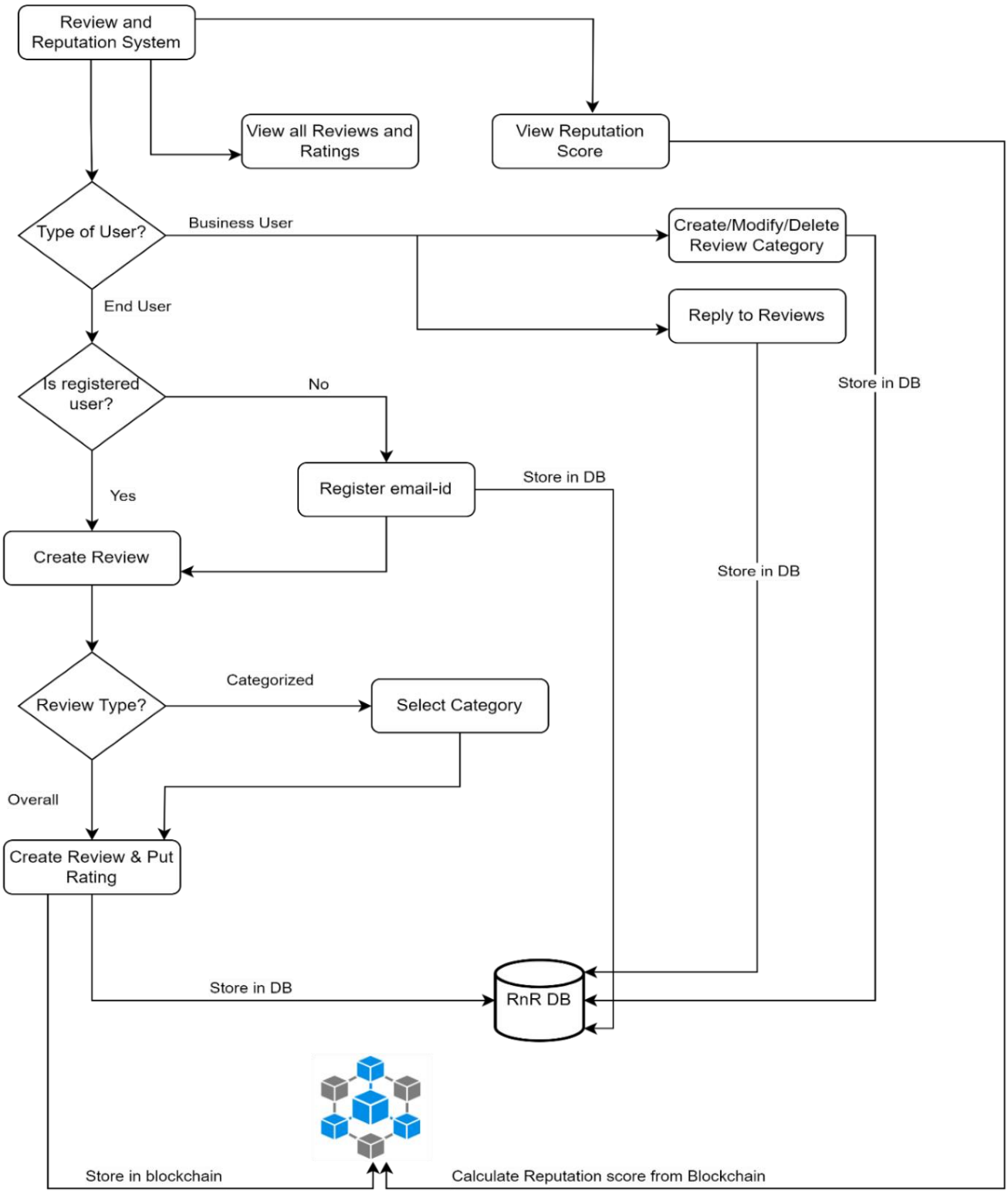
Table Of Contents

Milestone 1 -
UI Design and Tech Design
Review & Reputation 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/FACv8nNkArAruZd9gsdvpR/RnR?node-id=18-320&t=3BpMSOHQf3Sv40fG-1

2. Technical Design

2.1. Data Flow Diagram:



2.2. System Architecture Design:

2.2.1. Overview

The Review and Reputation System is designed to handle user reviews, ratings, and reputation scores, 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

- **Review Management Service**: Manages reviews and ratings.
- **Category Management Service**: Manages review categories.
- **Reputation Calculation Service**: Calculates overall reputation scores.
- **Reply Management Service**: Manages replies to reviews.
- **User Profile Management Service**: Manages user profiles.
- **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 Review & Reputation System for the Integration.**

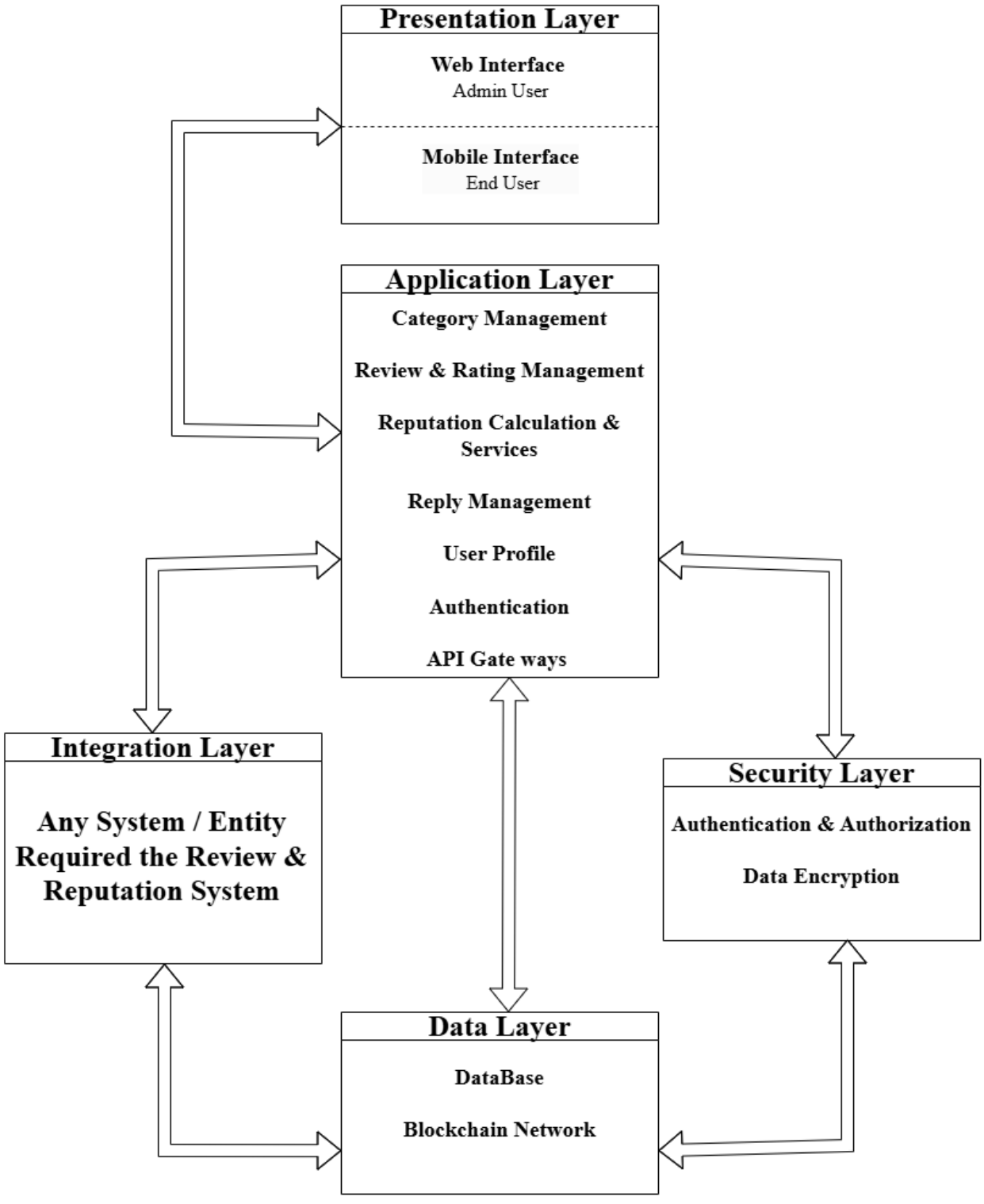
2.2.5. Security Layer

- **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, review content, metadata, and category information.
- **Blockchain Network**:
  - Cardano blockchain handles transaction data, storing hashes of reviewid and ratings and reputation scores 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):

https://github.com/AIQUANT-Tech/CardanoRnR/blob/main/DesignDocs/Table_Design_Generic.xlsx

2.3.6. API Design (Generic):

https://github.com/AIQUANT-Tech/CardanoRnR/blob/main/DesignDocs/API_Design_Generic.xlsx

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 reviews and reputation scores on the Cardano blockchain. It ensures transparency, immutability, and trust by implementing the following functionalities:

storeReview(reviewId, reviewReferenceId, rating, timestamp)

Functionality

- **Validation**:
  - Ensures that reviewId is unique.
  - Confirms that reviewReferenceId (e.g., Booking ID or Order ID) is optional.
  - Validates that the rating is an integer between 1 to 5.
- **On-Chain Storage**: If the above validations are successful, the review is stored on-chain as a **datum** associated with a transaction.

Explanation:

- The storeReview function takes a Review and a ScriptContext as inputs.
- It checks if the overallRating of the review is between 1 to 5.
- It ensures that the reviewReferenceId should meet the criteria Null or Not Null.

- If above conditions are met, the review is stored on-chain; otherwise, an error message is traced.

validateData(reviewId, reviewReferenceId, rating)

Ensures that the review data meets integrity constraints.

Functionality:

- Validates the uniqueness of reviewId (no duplicate reviews).
- Ensures reviewReferenceId matches a maybe (Booking/Order ID/etc).
- Confirms rating is an integer between 1 to 5.

Explanation:

- The validateData function checks the validity of the review data.
- It ensures the overallRating is between 1 to 5.
- It checks that the reviewReferenceId is Null / Not Null.
- If above conditions are met, the review data is considered valid.

retrieveReview(reviewId)

Retrieves a review from the blockchain based on its unique reviewId.

Functionality:

- Allows querying of review details for verification.
- Returns all the fields (e.g., reviewReferenceId, rating, timestamp) of a review.

Explanation:

- The retrieveReview function is used to fetch review details from the blockchain.
- It retrieves the reviewReferenceId, rating, and timestamp for a given reviewId.
- This operation is performed off-chain using blockchain APIs.
- Using Blockfrost
  curl -X GET "https://cardano-mainnet.blockfrost.io/api/v0/assets/<asset_id>" \ -H "project_id: <your_project_id>"
- **Retrieve Metadata**:
  Use the following API endpoint to retrieve metadata for a specific transaction:curl -H "project_id: <your_api_key>" \https://cardano-mainnet.blockfrost.io/api/v0/metadata/txs/labels/<label>

calculateReputationScore(entityId, maxPossibleRatings)

Calculates and updates the reputation score for an entity based on submitted reviews.

Functionality:

- Retrieves all reviews for the entity from on-chain data.
- Computes the reputation score using the formula.

Explanation:

- The calculateReputationScore function calculates the reputation score for an entity.
- It uses the average rating and the normalized count of reviews.
- The formula combines these factors to compute the final reputation score.

updateReputation(entityId, review)

Updates the reputation score of an entity when a new review is submitted.

Functionality

- Adds the new review's rating to the entity's totalScore.
- Increments the ratingCount by 1.
- Recalculates the reputation score using the updated data.

Explanation:

- The updateReputation function updates the entity's reputation score when a new review is added.
- It increases the totalScore by the overallRating of the new review.
- It increases the ratingCount by 1.
- The updated totalScore and ratingCount are used to recalculate the reputation score.

ensureAuthorizedUser(pubKeyHash)

Verifies that the user submitting the review is authorized.

Functionality

- Checks the PubKeyHash of the reviewer against a list of authorized users.
- Prevents unauthorized users from submitting fraudulent reviews.

Explanation:

- The ensureAuthorizedUser function checks if the reviewer is authorized to submit a review.
- It verifies the PubKeyHash of the reviewer against the transaction information.
- If the reviewer is not authorized, an error message is traced.

validateReviewReference(reviewReferenceId)

Ensures that the reviewReferenceId (Booking/Order ID) is optional (Null / Not Null).

Functionality

- Checks that the reviewReferenceId is optional.
- Validates the reference against predefined patterns or datasets.

Explanation:

- Validates the uniqueness of reviewId by ensuring it does not exist in previously stored reviews. Confirms that reviewReferenceId (Booking/Order ID) is optional. If reviewReferenceId is provided (non-null), it must not be an empty string and should meet predefined criteria (if any). If reviewReferenceId is null, the review is still valid. Checks that rating is within the range of 1–5.

rejectInvalidReviews(review)

Rejects invalid reviews (e.g., those with missing data, invalid ratings, or duplicate IDs).

Functionality:

- Rejects reviews that fail any validation checks (e.g., rating out of bounds, Review ID, etc.).

Explanation:

- The rejectInvalidReviews function rejects reviews that do not pass validation.

- It uses the validateData function to check the review data.
- Reviews with invalid data are rejected to maintain data integrity

Functions Summary

Function	Purpose
`storeReview`	Validates and stores the review on-chain.
`validateData`	Ensures review data is valid (rating range, unique ID, valid reference).
`retrieveReview`	Allows fetching a review based on its unique reviewId.
`calculateReputationScore`	Calculates the reputation score for an entity.
`updateReputation`	Updates the reputation score when a new review is added.
`ensureAuthorizedUser`	Verifies that the user submitting the review is authorized.
`validateReviewReference`	Ensures the reference ID (Booking/Order ID) is Null / Not Null.
`rejectInvalidReviews`	Rejects invalid reviews.

2.4.1**. Reputation Score Calculation and Storage: -**

2.4.1.1. Scope:

To ensure a trustworthy reputation system, the Cardano blockchain shall be used to calculate and store reputation scores. Storing the score directly on the blockchain makes it transparent and verifiable, which builds user trust.

2.4.1.2. Implementation:

On-Chain Score Updates: Develop a Plutus smart contract that calculates reputation scores based on predefined metrics that are the number review ratings, and user interactions (Number of ratings submitted).

Storing Reputation Scores: Organization wise reputation score is stored as an on-chain variable. When actions that affect reputation (**A new rating**) are recorded, the score is updated through the smart contract.

2.4.1.3. Data Structures:

Review: Stores details of each rating, including ratings and timestamps.

```
data Rating = Rating
  {
 reviewer  :: PubKeyHash // This field stores the public key hash of the reviewer. It uniquely identifies the user who submitted the review.
, ratings      :: [Integer] // This field is integer representing the rating given by the reviewer.
, timestamp:: POSIXTime // This field stores the timestamp of when the
```

```
rating was submitted.
  }
```

Entity: Stores the reputation score for an entity.

```
data Entity = Entity
  {
totalScore  :: Integer // This field stores the cumulative sum of all ratings
received by the entity.
  }
```

2.4.1.4. Designing a Reputation Score Algorithm

A reputation score typically reflects user behavior, engagement, and the value of contributions over time. Key factors and approaches for designing the algorithm are:

Factors Influencing Reputation:

- **Number of Review Frequency**: The number of reviews submitted over a specific time.
- **Value of Ratings**: Based on metric is rating provided by reviewer.

Algorithm Approach: (**On-Chain)**

Here's a reputation score algorithm where the **key factors** are **ratings (1 to 5)** and **the number of ratings given by the user**, with a weight of **0.5(configurable)** assigned to both components:

Reputation Score Formula

Reputation Score = (Wr * Average Rating) + (Wn * Normalized Count)

Explanation of Terms

Average Rating:

Average Rating = Sum of Ratings / Number of Ratings

This measures the average score of the user's ratings.

Normalized Rating Count:

Normalized Rating Count = Number of Ratings / Max Possible Ratings

Weights:

Wr =0.5 Wn =0.5

These weights balance the contributions of the average rating and the normalized rating count.

Implemented in Haskell:

```
calculateReputationScore :: Entity -> Integer -> Integer
calculateReputationScore entity maxPossibleRatings =
   let wr = 50 -- Weight for average rating (Wr = 0.5 as per 100 - based scale)
       wn = 50 -- Weight for normalized count (Wn = 0.5 as per 100 - based
scale)
       avgRating = (totalScore entity) / (ratingCount entity)
       normalizedCount = (ratingCount entity) / (maxPossibleRatings)
   in (wr * avgRating + wn * normalizedCount) / 100
```

2.4.1.5. Storage:

Blockchain Storage: Use Cardano to store hashes and proofs of reputation score.

Develop the Smart Contract

Write the Contract: Use Plutus and Haskell to code the smart contract, implement the logic for checking activity, calculating of Reputation Score by each Rating submission, and updating Score.

Reputation Score Integration:

User Interaction:

- Ratings submission.

Reputation Calculation:

- Each submission of Rating triggers the **Reputation Service** to recalculate the score in real-time.

Storage:

Updated scores are stored in the Cardano Blockchain.

Blockchain Logging:

The **Blockchain Service** generates a hash of scores and uploads it to Cardano.

Access Control

Implement robust authentication and authorization mechanisms.

Ensure only authorized users can submit reviews, ratings and access reputation scores.

2.4.2. Storage of Reviews ID, reviewReference ID and Ratings in Blockchain

2.4.2.1. Scope:

Ensure that review ID, reviewReference ID and review ratings cannot be altered or deleted once submitted, providing transparency and trust.

2.4.2.2. Implementation:

Smart Contracts: Use Plutus smart contracts to store review IDs, reviewReference ID (optional) and review ratings on Cardano blockchain.

Data Flow: When a user submits a review, the review ID and review Rating data are sent to a smart contract, which records it on the blockchain.

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 Review Data**: Format the Review ID, reviewReference ID and Rating into a datum file (datum.json):

```
{
  "constructor": 0,
  "fields": [
     { "bytes": "REV12345" },
    { "constructor": 1, "fields": [] }, // reviewReferenceId is null
     { "int": 5 }
  ]
}
```

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 reviews.
6. **Data Structures of Smart Contract:**

The Review data structure includes:

- reviewId: A unique ID for the review with respect to reviewReferenceId.
- reviewReferenceId: A unique id (booking/order ID/etc) for which the reviews be taken.
- overallRating: The rating value (1–5).
- timestamp: Timestamp when the review was created.

```
data Review = Review
  {
    reviewId          :: BuiltinByteString  -- Unique Review ID
  , reviewReferenceId :: BuiltinByteString  -- Nullable Booking/Order ID
  , overallRating     :: Integer            -- Rating (1 to 5)
  , timestamp         :: POSIXTime          -- Time of review submission
  }
```

Smart Contract Details: -

- **Key Functions**:

  storeReview(reviewID, reviewReferenceId, rating): Stores the review ID, reviewReferenceId and rating after validation.

  validateData(reviewID, reviewReferenceId, rating): Ensures that the review ID is unique, and rating is within acceptable bounds.

  retrieveReview(reviewID, reviewReferenceId): Enables fetching of review ratings for verification purposes.

Component	Purpose
Review Data Type	Encapsulates review ID, reviewReferenceId and rating data.
Validation Logic	Ensures the integrity of inputs (valid rating range).
Validator Script	Applies validation logic and rejects invalid transactions.

Contract Address	Enables interaction with the smart contract.
Typed Validator	Provides type safety and binds the validation logic to the blockchain.
Helper Functions	Store and retrieve review data, linking on-chain and off-chain processes.

Validator Function:

- Accepts:
  - `Review` as input (Datum).
  - No redeemer is required.
- Validates:
  - Review rating is between 1–5.

```
typedValidator :: TypedValidator Review
typedValidator = mkTypedValidator @Review
  $$(PlutusTx.compile [|| validateReview ||])
  $$(PlutusTx.compile [|| wrap ||])
 where
  wrap = mkUntypedValidator
```

2.4.2.3. Smart Contract Interaction with Blockchain

1. **Submission to the Blockchain**
   a. The front-end collects review data: reviewID, reviewReferenceId (optional) and rating.
      If no Booking/Order ID exists, the reviewReferenceId is set to null.
      The user submits a review for a specific booking or order.
   b. The system sends this data to the **Plutus smart contract** via a transaction.
   c. The smart contract acts as a **validator** to ensure the integrity of the submitted data.
2. **Smart Contract Validation**
   a. The smart contract validates the incoming data:
      i. Validates the reviewReferenceId is either Null or valid.
      ii. Validates the overallRating is within the range (1–5).
   b. If validation passes, the data is included in a transaction to be recorded on the blockchain.
3. **Blockchain Storage**
   a. Once validated, the transaction is **signed and submitted** to the Cardano blockchain.
   b. The blockchain stores the review data as part of a transaction's metadata (in binary or JSON format).
4. **Confirmation**
   a. After successful storage, the blockchain generates a **transaction ID** (TxID), which shall be used for:
      i. Verifying that the data is on-chain.
      ii. Retrieving the data later using a query.

2.4.2.4. Data Flow Steps

1. **User Interaction**: User submits a review through the UI.
2. **Source of Data**:

a. The **Review ID,** Booking/Order ID (**ReviewReferenceId**). and **Rating** are collected from a front-end system.

3. **How the Data is Processed**:

a. The application gathers the review data (reviewID = "REV5", "reviewReferenceId": "BOOK12345", rating = 5).

b. The data is formatted as a **datum** to be sent to the smart contract.

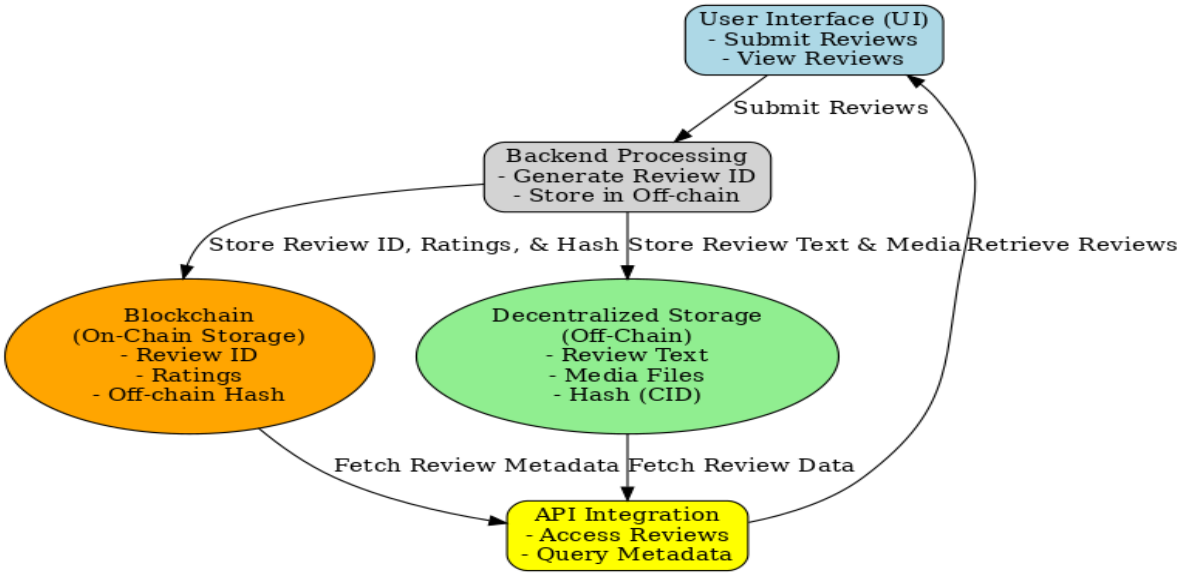
4. **Validation and Storage by Smart Contract**:

a. The smart contract validates the data (ensures the rating is between 1 - 5).

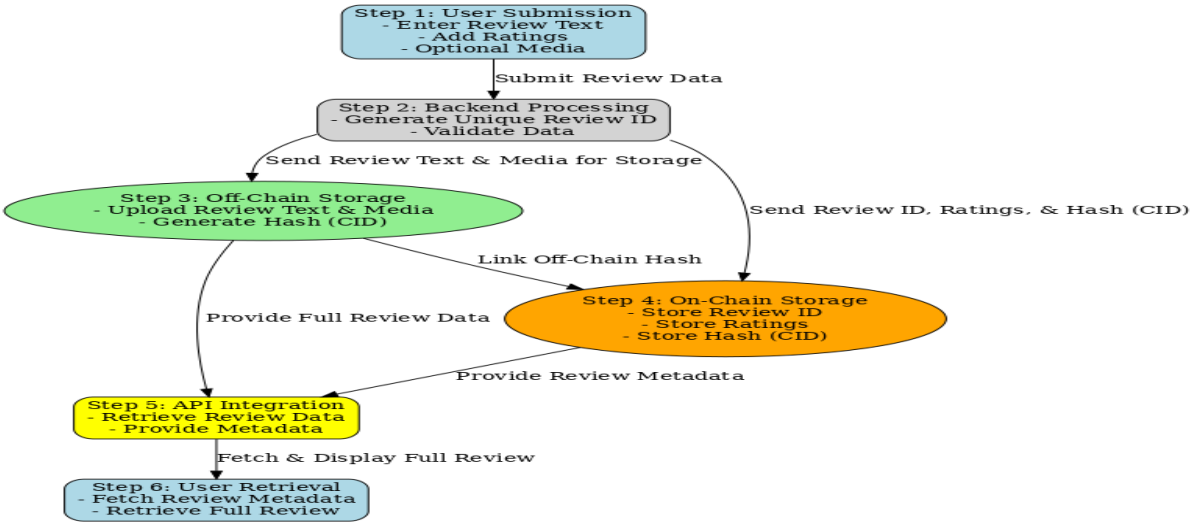
b. The data is stored as part of the blockchain transaction interacting with the contract.

c. The smart contract logic ensures that Review IDs are unique, and Ratings are within the acceptable range.

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 in on-chain.

2.5. Test Plan:

2.5.1. Summary

This test plan outlines the testing strategy for the **Review and Reputation System** feature. The purpose of this system is to allow users to submit reviews and ratings, view all the reviews and accumulate reputation scores based on ratings received from reviewers.

2.5.2. Objectives

- To verify that users can successfully submit reviews and ratings.

- To validate that reputation scores are calculated and updated accurately.
- To ensure proper access control, security, and data integrity in the system.
- To test the overall end user experience of submitting reviews, viewing reviews and reputation score.
- To test the overall business user experience of creating review categories, viewing reviews and replying on user reviews.

2.5.3. Scope

The testing will cover the following components of the Review and Reputation System:

- End user interface for submitting and viewing reviews.
- Back-end calculations for reputation scores.
- Business user interface for creating categories, viewing reviews and reply to reviews.
- System performance and scalability under load.

2.5.4. Test Strategy

- **Test Type**: Manual and automated tests
- **Test Levels**: Unit testing, Integration testing, Functional testing, Performance testing, and User acceptance testing (UAT).
- **Tools**: Selenium, Postman (for API testing), JUnit (for unit tests), LoadRunner (for performance testing).

2.5.5. Test Scenarios

2.5.5.1. Review Submission

- **Test Case 1**: User can successfully submit a review with valid data (rating, text).
    - Steps:
        1. Log in as a registered user or provide email id to register yourself
        2. Navigate to the product or service.
        3. Submit a review with a rating and comment.
        4. Verify that the review appears in the review list.
        5. Ensure the review displays the correct timestamp and rating.
- **Test Case 2**: User cannot submit a review without a rating or text.
    - Steps:
        1. Try to submit a review with no text or rating.
        2. Verify that a validation message is displayed.

2.5.5.2. Review Display and Interaction

- **Test Case 3**: Reviews are correctly displayed on the product/service page.
    - Steps:

1. Navigate to the review listing page.
2. Verify that all reviews are listed in descending order of submission date.
3. Ensure that the average rating is calculated correctly.

- **Test Case 4**: Review shall be reported by users.
    - Steps:
        1. Report an inappropriate review.
        2. Verify that the review is flagged for admin review.

2.5.5.3. Reputation Score Calculation

- **Test Case 5**: Reputation score is calculated based on the number and ratings.

  Steps:
    1. Submit a higher rating and ensure the user's reputation score increases.
    2. Submit a lower rating and check if the reputation score is penalized.
- **Test Case 6**: Reputation score updates immediately after submitting a new review.

  Steps:
    1. Submit a review.
    2. Verify that the reputation score is updated immediately.
    3. Ensure the reputation score persists across sessions.

2.5.5.4. User Permissions

- **Test Case 8**: End users cannot edit or delete reviews.

  Steps:
    1. Log in as an End user.
    2. Try to edit or delete another user's review.
    3. Verify that no option is there.
- **Test Case 9**: Business user can edit or delete any review category.

  Steps:
    1. Log in as a business user.
    2. Delete any review categories
    3. Edit user review/ review categories.
    4. Reply to any review.
    5. Ensure the changes are reflected properly.
- **Test Case 10**: Unregistered user can create review by registering.

  Steps:
    1. View Reviews.
    2. Try to create own review

3. Prompt for email id to proceed.
4. Submit email id.
5. Create a review.

2.5.5.5. Performance Testing

- **Test Case 11**: System can handle a large number of simultaneous reviews.

  Steps:

  1. Use a load testing tool (e.g., LoadRunner) to simulate multiple users submitting reviews.
  2. Measure response times and system stability under load.

2.5.5.6. Security Testing

- **Test Case 12**: Reviews are sanitized to prevent XSS attacks.

  Steps:

  1. Verify that the input is sanitized, and the review is stored safely.

2.5.5.7. Prototype Testing on Testnet

- **Test Case 13**: Verify that prototype should successfully connect to the blockchain testnet**.**

  Steps**:**

  1. Ensure the testnet environment is configured.
  2. Verify that the application connects successfully to the testnet.

- **Test Case 14**: Verify that ratings cannot be tampered with once posted (immutability).

  Steps:

  1. Attempt to modify the submitted rating through the user interface.
  2. Check the blockchain to ensure no change is reflected.

- **Test Case 15** Ensure synchronization between front-end review interface and blockchain.

  Steps:

  1. Ratings are posted in the blockchain.
  2. The review is recorded as a transaction on the testnet blockchain, and the transaction metadata matches the review data (reviewId, reviewReferenceId, rating, timestamp).
  3. Refresh the front-end review interface.
  4. Confirm the review is reflected on the UI.

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 production 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 Accounts**: Create multiple test accounts with different user roles (admin, regular user, etc.).
- **Test Reviews**: Prepare test cases with varied review contents (positive, neutral, negative) and ratings (1-5 stars).

2.5.8. Entry and Exit Criteria

- **Entry Criteria**:
  - Development is complete for the Review and Reputation System.
  - Test environments are set up and ready.
- **Exit Criteria**:
  - All planned test cases have been executed.
  - Critical and high-severity bugs are fixed.
  - All test cases have passed or have been re-tested after fixes.

2.6. Deployment Plan:

This deployment plan outlines the setup for a Review and Reputation System leveraging the MERN stack (MongoDB, Express.js, React.js, Node.js) along with Blockchain for decentralized reputation 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 Review and Reputation System will allow users to leave reviews for products or services, and reputation will be calculated through blockchain. 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.

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:**
    - Host the code on GitHub.
    - Make APIs open source by providing access to the repository.
- **CI/CD Pipeline:**
    - Set up a CI/CD pipeline with GitHub Actions Jenkins for automatic build and deployment.
    - Automated tests should be implemented for both the frontend and backend.
    - 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:**
  - Use Kubernetes to manage containerized services, which will be beneficial when scaling the system to multiple nodes.
- **Cloud Database:**
  - Move MongoDB to a cloud-managed service (e.g., MongoDB Atlas).
- **Cloud Blockchain Nodes:**
  - 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. Review and Reputation System Integrated with Hotel Booking System

3.1. Introduction

The part of this document outlines the process for integrating the Review and Reputation System (RnR) with the Hotel Booking System (HBS). By combining these systems, hotels can better manage guest review, enhance their online reputation, and improve guest satisfaction. The integration will automate the flow of information between the booking system and the reputation management platform, ensuring seamless feedback collection, monitoring, and response.

3.2. Objectives

The primary objectives of this integration are:

- To collect, manage, and analyse guest reviews in real-time.
- To automate the process of sending review requests after a guest's stay.
- To display guest reviews on the hotel's website.
- To allow hotel management to track and respond to reviews efficiently.
- To leverage reviews for marketing, promotional, and improvement purposes.

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. Review and Reputation System Features

The RnR allows for the collection, monitoring, and response to guest reviews. Key features include:

- Review Collection (Automated requests after stay)
- Reputation Analysis (Identify positive, negative, or neutral rating)
- Reporting & Analytics (Trends, Scores, Guest Insights)
- Review Response Management (Direct reply to reviews)

3.3.3. Integration Goals

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

- Automatic sending of review requests to guests after their stay based on booking data from the Hotel Booking System.
- Real-time synchronization of booking and guest information between both systems.
- The ability for hotel staff (Business User) to respond to reviews from within the Hotel Booking System interface.
- Real-time sentiment analysis and reporting of review data alongside booking and performance analytics.

3.4. Integration Process

3.4.1. Data Flow Overview

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

1. **Booking Confirmation**: Once a guest completes a booking and the reservation is confirmed in the HBS, their booking information (name, email, stay dates) is sent to the RnR.
2. **Stay Completion**: RnR scheduler will check the stay completion status.
3. **Review Request Trigger**: Upon guest checkout, the RnR sends an automated email requesting the guest to leave a review. This request is personalized based on guest data.

4. **Review Collection**: The guest submits a review through the RnR. The review is then displayed on the RnR through hotel's website or aggregated on a central dashboard.
5. **Reputation Analysis**: The RnR analyses the ratings, calculating the reputation score.
6. **Review Management**: The hotel staff (Business User) can view, respond reviews using RnR system through hotel's website or on a central dashboard.

3.4.2. Integration Components

API Integration

- **Hotel Booking System API**: The hotel booking system will invoke an API to share booking details (e.g., guest email, stay dates) with RnR.
- **RnR System API**: The review system will expose an API for retrieving and sending review requests, collecting reviews, and review responses.
- **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 amenities and room categories to RnR and RnR system will store these data as review categories.

3.5. Technical Requirements

3.5.1. Database Schema:

The database schema will need to integrate guest, booking, and review data between the two systems. Below are suggested table structures for both the Hotel Booking System (HBS) and Review and Reputation System (RnR):
https://github.com/AIQUANT-Tech/CardanoRnR/blob/main/DesignDocs/Table_Design_Hotel_Usecase.xlsx

3.5.2. API Specifications:

For seamless integration, both systems will communicate using RESTful APIs. Below is the API design:
https://github.com/AIQUANT-Tech/CardanoRnR/blob/main/DesignDocs/API_Design_Hotel_Usecase.xlsx

3.6. Data Flow

3.6.1. Booking to Review Request:

1. **Event**: Guest completes booking (confirmed).
2. **Action**: RnR scheduler component pulls guest and booking details in regular interval(configurable).
3. **Outcome**: RnR sends a review request to the guest via email after checkout. This email includes a hyperlink that directs the guest to the RnR system, which is integrated into the hotel dashboard.

3.6.2. Review Submission:

4. **Event**: Guest submits a review via the RnR.
5. **Action**: RnR stores review data in the reviews table and triggers Reputation score generation.
6. **Outcome**: Review and generated Reputation score is visible on the RnR system through hotel's dashboard.

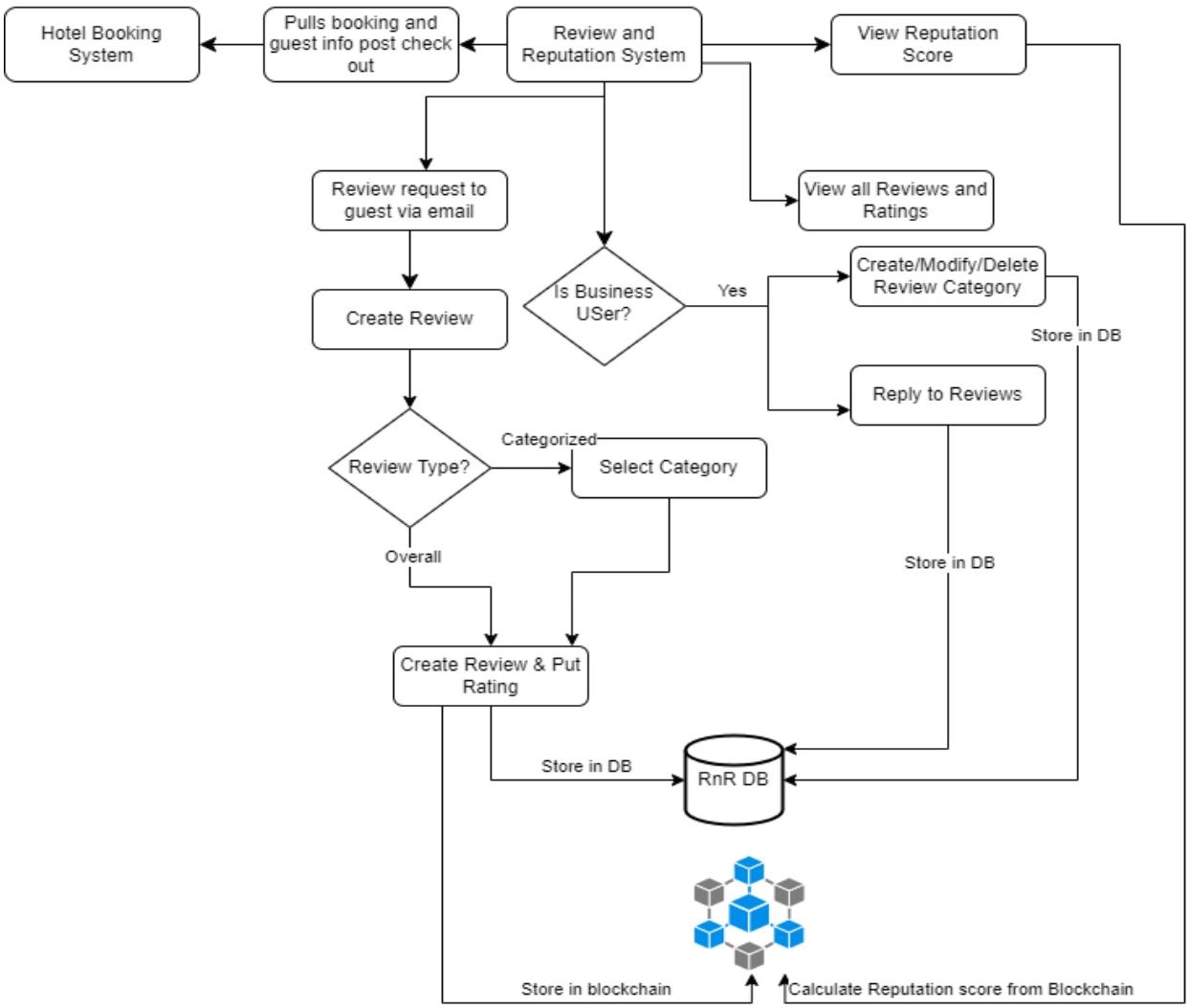
3.6.3. Review Response:

7. **Event**: Hotel staff responds to a review via the RnR.
8. **Action**: RnR stores the response in the table.
9. **Outcome**: Hotel staff can view and manage responses within RnR thorough the hotel’s dashboard.

3.6.4. Create Review Category:

10. **Event**: Hotel staff create a review category via RnR.
11. **Action**: RnR stores the category in the table.
12. **Outcome**: Guest can give review and rating by selecting a category. Hotel staff can view reviews by categories.

3.6.5. Integration Data Flow:



3.7. Security Considerations

- **Authentication**: All API endpoints 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

A scheduler component needs to be developed within RnR application which will pull HBS guest and booking details which are in post check out state, but Review request email has not been sent yet.

- Scheduler will pull (in regular interval) guest_id,first_name,last_name,email,booking_id from

Guest_Info,Booking_Info for which check_out_date is past and is_rnr_notified is false.

- Email will be triggered to each guest email and is_rnr_notified will be set to true and updated_at will be modifed.

3.9. Test Plan:

Summary: This Test Plan outlines the strategy, objectives, scope, methodology, and schedule for testing the integration between the RnR system and the HBS. The primary goal is to ensure the proper functioning, reliability, and synchronization of the integration and to validate that data flows seamlessly between the two systems.

3.9.1. Test Strategy:

3.9.1.1. Scheduler Testing

- **Objective:** Ensure that all APIs between the RNR system and hotel booking system are functioning as expected.
- **Tests:**
  - Verify that the scheduler component of RnR pulls correct guest and booking details which are still not pulled.
  - Already pulled data will not be fecthed again.
  - Ensure that error handling works properly (e.g., invalid reservation IDs, missing fields).
  - Validate the response times and status codes.

3.9.1.2. Data Synchronization Testing

- **Objective: Ensure data consistency and accurate synchronization between systems.**
- **Tests:**
  - Ensure that guest details (id, name, email id) are correctly mapped and updated.
  - Check HBS's guest ids are properly mapped with RnR's user ids.
  - Verify all amenities of HBS are synchronized correctly with Categories of RnR system.
  - Verify the time zone and date-time handling for bookings.

3.9.1.3. Security Testing

- **Objective: Ensure that the integration is secure.**
- **Tests:**
  - Verify that all API communication is encrypted (HTTPS).
  - Ensure that only authorized systems can pull or push data.
  - Test for potential security vulnerabilities (e.g., injection attacks, man-in-the-middle attacks).

3.9.1.4. Prototype Testing on Testnet

- **Objective: Ensure that after checkout, the user can submit a rating and store in blockchain as immutable data.**
- **Tests:**

- Verify that a user submits rating to blockchain after checkout.
- A transaction is recorded on the blockchain testnet with Booking id as reviewReferenceId.
- The review is recorded as a transaction on the testnet blockchain, and the transaction metadata matches the review data (reviewId, reviewReferenceId, rating, timestamp).
- Verify that review data like, reviewReferenceId, rating cannot be modified once submitted in the testnet blockchain.
- The Ratings displayed in the Hotel Booking System should match exactly with the data stored on the testnet blockchain.
- Verify that no review can be deleted once it has been posted on the testnet blockchain.