**RewardPointCalculator Project Documentation**

**Overview**

The **RewardPointCalculator** application is a Spring Boot-based backend service that calculates reward points for customers based on their purchase transactions. The service provides RESTful endpoints for customer registration, login/logout, managing customer transactions, and retrieving reward point reports.

**Key Features:**

1. **Reward Points Calculation**: Customers earn reward points for each transaction based on the following logic:
   * 2 points for every dollar spent above $100.
   * 1 point for every dollar spent between $50 and $100.

Example:

* + A $120 purchase = 2 x $20 + 1 x $50 = 90 points.

1. **Customer and Transaction Management**: Customers can be registered, and their transactions (get, add, edit, delete) can be managed.
2. **Reward Points Report**: Reward points earned by a customer for each month and total points can be viewed via APIs.
3. **Asynchronous Task Execution**: The system uses asynchronous methods to update reward points, ensuring the user experience is not hindered by long-running tasks.
4. **Security and Authentication**: The application uses Spring Security for user authentication, and APIs are protected by authentication filters.

**1. High-Level Design**

The application follows a modular design with the following key components:

* **Controller Layer**: Exposes RESTful endpoints to handle HTTP requests.
* **Service Layer**: Contains business logic, including the calculation of reward points and the interaction with repositories.
* **Repository Layer**: Interfaces with the database using Spring Data JPA.
* **Database**: PostgreSQL database used for persistence.
* **Security Layer**: Authentication and authorization using Spring Security.

**2. Database Design**

The database consists of the following entities:

**Customer Table**

CREATE TABLE customer (

id SERIAL PRIMARY KEY,

first\_name VARCHAR(255) NOT NULL,

last\_name VARCHAR(255) NOT NULL,

email VARCHAR(255) NOT NULL UNIQUE,

password VARCHAR(255) NOT NULL);

**CustomerTransaction Table**

CREATE TABLE customer\_transaction (

id SERIAL PRIMARY KEY,

customer\_id BIGINT NOT NULL,

amount DOUBLE PRECISION NOT NULL,

spent\_details TEXT,

date DATE NOT NULL,

CONSTRAINT fk\_customer FOREIGN KEY (customer\_id) REFERENCES customer(id) ON DELETE CASCADE);

**RewardPoints Table**

CREATE TABLE reward\_points (

id SERIAL PRIMARY KEY,

customer\_id BIGINT NOT NULL,

points INT NOT NULL,

month INT NOT NULL CHECK (month >= 1 AND month <= 12),

year INT NOT NULL CHECK (year >= 1900 AND year <= 9999),

CONSTRAINT fk\_customer\_reward FOREIGN KEY (customer\_id) REFERENCES customer(id) ON DELETE CASCADE);

**BlacklistedToken Table**

CREATE TABLE blacklisted\_token (

id SERIAL PRIMARY KEY,

token VARCHAR(255) NOT NULL,

username VARCHAR(255) NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**3. Key Business Logic**

The core business logic revolves around calculating reward points for a customer based on their transactions.

**Reward Points Calculation Logic**

* **If a customer spends over $100**:
  + 2 points for every dollar spent over $100.
* **If a customer spends between $50 and $100**:
  + 1 point for every dollar spent between $50 and $100.

java

Copy

public class RewardPointCalculator {

public static int calculatePoints(double amount) {

int points = 0;

if (amount > 100) {

points += (amount - 100) \* 2; // 2 points for every dollar over $100

amount = 100; // Cap amount at $100

}

if (amount > 50) {

points += (amount - 50); // 1 point for every dollar between $50 and $100

}

return points;

}

}

**4. API Endpoints**

**Customer APIs**

1. **POST /customers/register** – Registers a new customer.

**Auth Apis**

1. **POST /customers/login** – Logs in a customer and provides authentication.
2. **POST /customers/logout-** Logs out the current customer

**Customer Transaction APIs**

1. **GET /transactions/{customerId}** – Retrieves all transactions for a specific customer.
2. **POST /transactions/{customerId}** – Adds a new transaction for a customer.
3. **PUT /transactions/{customerId}/{transactionId}** – Edits an existing transaction.
4. **DELETE /transactions/{customerId}/{transactionId}** – Deletes a transaction.

**Reward Points APIs**

1. **GET /reward-points/{month}/{year}** – Retrieves reward points for a specific customer, month, and year.
2. **GET /reward-points/all**– Retrieves all reward points for a customer across multiple months and years.
3. **Exception Handling and Error Management**

The system includes detailed exception handling mechanisms:

**Global Exception Handling**

A centralized exception handler captures and handles all exceptions, providing meaningful error messages.

**Custom Exception Handling**

The application uses Spring's @ControllerAdvice to handle exceptions globally, providing detailed error messages for common scenarios like customer not found, invalid input, or database issues.

Example of a custom exception handler:

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResponseStatusException.class)

public ResponseEntity<String> handleResponseStatusException(ResponseStatusException ex) {

return new ResponseEntity<>(ex.getReason(), ex.getStatus());

}

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleGeneralException(Exception ex) {

return new ResponseEntity<>("An unexpected error occurred: " + ex.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**HTTP Status Codes**: Standard HTTP status codes are used to indicate the result of an API call:

* 200 OK: Successful request.
* 400 Bad Request: Invalid input or data.
* 404 Not Found: Resource does not exist.
* 500 Internal Server Error: General server error.

**6. Security**

The application implements **Spring Security** to protect endpoints and ensure that customers must be authenticated to perform specific operations.

* **Authentication**: JWT-based authentication is used.
* **Authorization**: Role-based access control (RBAC) is enforced, allowing only authorized users to check their own reward points.

**7. Performance Optimization**

**Database Indexing**

Indexes are created on frequently queried columns, such as customer\_id, month, and year in the reward\_points and customer\_transaction tables.

**Asynchronous Processing**

Some operations, such as updating reward points, are processed asynchronously using Spring’s @Async annotation to avoid blocking the main thread and improve performance.

@Async

public void updateRewardPointsAsync(Long customerId, Integer month, Integer year) {

updateRewardPoints(customerId, month, year);

}

**Database Query Optimization**

Efficient queries are used to fetch transaction data and calculate the reward points based on the provided date ranges.

**8. Code Quality and Documentation**

* The code follows **SOLID principles** to ensure high reusability and maintainability.
* Methods and classes are documented using Javadoc, explaining their purpose and functionality.
* Proper input validation is used for incoming requests to ensure data integrity.

Example:

public class CustomerDTO {

@NotEmpty(message = "First name is required")

private String firstName;

@NotEmpty(message = "Last name is required")

private String lastName;

@Email(message = "Email should be valid")

private String email;

@NotEmpty(message = "Password is required")

private String password;

}

**9. Test Coverage**

**Unit Tests**

JUnit 5 and Mockito are used for unit testing. Key service methods like calculatePoints() are unit-tested to ensure correct behavior.

Example unit test for reward points calculation:

@ExtendWith(MockitoExtension.class)

public class RewardPointCalculatorTest {

@Test

public void testCalculatePoints() {

assertEquals(90, RewardPointCalculator.calculatePoints(120));

assertEquals(50, RewardPointCalculator.calculatePoints(50));

assertEquals(0, RewardPointCalculator.calculatePoints(30));

}

}

**Integration Tests**

Integration tests ensure that the API endpoints function correctly and interact with the database as expected.

**10. Swagger / OpenAPI Documentation**

Swagger (OpenAPI) documentation is integrated into the project using the springdoc library. It provides a UI to explore all available APIs, request parameters, and response formats.

To enable Swagger, the following dependency is added:

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.8.3</version>

</dependency>

Swagger UI can be accessed at /swagger-ui.html

**11. GitHub Repository**

The source code, along with all configurations, test cases, and documentation, is hosted on **GitHub**.

* **GitHub Repository** : https://github.com/RifatBano/RewardPointsCalci

**Conclusion**

The **RewardPointCalculator** service is a robust, well-designed Spring Boot application that follows best practices in exception handling, security, and performance optimization. It efficiently calculates and manages reward points for customers, while providing a user-friendly API interface.