1) Test Case for CustomerControllerTest

**1. testSaveCustomer**

**Purpose:**

* + To verify that the CustomerController correctly handles a POST request to save a customer and returns the appropriate Customer object in the response.

**Setup:**

* + Mock CustomerService to return a Customer object when save() is called.
  + Prepare a Customer object to be used in the test.

**Expected Behavior:**

* + The CustomerController should return a ResponseEntity with a status of OK (HTTP 200).
  + The response body should contain the Customer object that was saved, with the ID and name matching the mock data.

**Assertions:**

* + Verify that the HTTP status code is 200 OK.
  + Verify that the JSON response contains the correct customer\_id and customer\_name fields.

**Example Input:**

* + Request body: {"customer\_id":1,"customer\_name":"John Doe","listTransaction":[]}

**Example Output:**

* + ResponseEntity<Customer> with HTTP status 200 OK and body containing { "customer\_id": 1, "customer\_name": "John Doe" }

**2. testGetAllCustomer**

**Purpose:**

* + To verify that the CustomerController correctly handles a GET request to fetch all customers and returns the appropriate list of Customer objects.

**Setup:**

* + Mock CustomerService to return a list of Customer objects when getAllCustomer() is called.

**Expected Behavior:**

* + The CustomerController should return a ResponseEntity with a status of OK (HTTP 200).
  + The response body should contain a list of Customer objects, each with the correct ID and name.

**Assertions:**

* + Verify that the HTTP status code is 200 OK.
  + Verify that the JSON response contains the correct fields for each Customer object in the list.

**Example Input:**

* + No request body.

**Example Output:**

* + ResponseEntity<List<Customer>> with HTTP status 200 OK and body containing:

response body

[

{

"customerId": 1,

"customerName": "Shubham",

"listtransaction": [

{

"transactionId": 501,

"customerId": 1,

"transactionDate": "2024-05-20T00:00:00.000+00:00",

"transactionAmount": 120.0

},

{

"transactionId": 502,

"customerId": 1,

"transactionDate": "2024-07-20T00:00:00.000+00:00",

"transactionAmount": 55.0

},

{

"transactionId": 503,

"customerId": 1,

"transactionDate": "2024-08-07T00:00:00.000+00:00",

"transactionAmount": 120.0

}

]

},

{

"customerId": 2,

"customerName": "John",

"listtransaction": [

{

"transactionId": 504,

"customerId": 2,

"transactionDate": "2024-05-10T00:00:00.000+00:00",

"transactionAmount": 120.0

},

{

"transactionId": 505,

"customerId": 2,

"transactionDate": "2024-06-20T00:00:00.000+00:00",

"transactionAmount": 55.0

},

{

"transactionId": 506,

"customerId": 2,

"transactionDate": "2024-07-07T00:00:00.000+00:00",

"transactionAmount": 120.0

},

{

"transactionId": 507,

"customerId": 2,

"transactionDate": "2024-07-25T00:00:00.000+00:00",

"transactionAmount": 350.0

}

]

}

]

**3. testGetCustomerById**

* **Purpose:**
  + To verify that the CustomerController correctly handles a GET request to fetch a customer by ID and returns the appropriate Customer object.
* **Setup:**
  + Mock CustomerService to return a Customer object when findById() is called with a specific ID.
* **Expected Behavior:**
  + The CustomerController should return a ResponseEntity with a status of OK (HTTP 200).
  + The response body should contain the Customer object with the correct ID and name.
* **Assertions:**
  + Verify that the HTTP status code is 200 OK.
  + Verify that the JSON response contains the correct customer\_id and customer\_name.
* **Example Input:**
  + customerId: 1
* **Example Output:**
  + ResponseEntity<Customer> with HTTP status 200 OK and body containing { "customer\_id": 1, "customer\_name": "John Doe" , "listtransaction": [

{

"transactionId": 501,

"customerId": 1,

"transactionDate": "2024-05-20T00:00:00.000+00:00",

"transactionAmount": 120.0

},

}

**4. testGetCustomerByIdNotFound**

**Purpose:**

* + To verify that the CustomerController correctly handles a GET request for a non-existent customer ID by returning a 404 Not Found status.

**Setup:**

* + Mock CustomerService to return null when findById() is called with a non-existent ID.

**Expected Behavior:**

* + The CustomerController should return a ResponseEntity with a status of Not Found (HTTP 404).

**Assertions:**

* + Verify that the HTTP status code is 404 Not Found.

**Example Input:**

* + customerId: 999

**Example Output:**

* + ResponseEntity with HTTP status 404 Not Found

2) Test Cases for RewardControllerTest

#### **1. testGetRewardsByCustomerId\_Success**

* **Purpose:** To verify that the RewardController correctly handles a GET request for rewards by customer ID and returns the appropriate Rewards object in the response.
* **Setup:**
  + Mock the RewardService to return a Rewards object for a given customer ID.
  + Mock the CustomerRepository to return a Customer object when searching by customer ID.
  + Prepare a Rewards object with nested CustomerInfo and RewardDetails to simulate realistic data.
* **Expected Behavior:**
  + The RewardController should return a ResponseEntity with a status of OK (HTTP 200).
  + The response body should contain a JSON object with the correct customer, totalPoints, and detailed rewards for the customer.
* **Assertions:**
  + Verify that the HTTP status code is 200 OK.
  + Verify that the JSON response contains the correct customer.id, customer.name, totalPoints, and detailed rewards for the customer.
  + Validate the JSON response includes the correct transaction details such as transactionId, transactionAmount, points, and transactionDate.
* **Example Input:**
  + Request URL: /api/v1/{customerId}/rewards
  + Request Method: GET
  + Path Variable: customerId = 1
* **Example Output:**

json

Response body

{  
 "customer": {  
 "id": 1,  
 "name": "John Doe"  
 },  
 "totalPoints": 100,  
 "rewards": [  
 {  
 "transactionId": "12345",  
 "transactionAmount": 120.0,  
 "points": 90,  
 "transactionDate": "2024-08-03T08:49:41.859753"  
 }  
 ]  
}

#### **2. testGetAllCustomerRewards\_Success**

* **Purpose:** To verify that the RewardController correctly handles a GET request to fetch rewards for all customers and returns the appropriate list of Rewards objects in the response.
* **Setup:**
  + Mock the RewardService to return a list of Rewards objects.
  + Prepare multiple Rewards objects with nested CustomerInfo and RewardDetails to simulate realistic data for multiple customers.
* **Expected Behavior:**
  + The RewardController should return a ResponseEntity with a status of OK (HTTP 200).
  + The response body should contain a JSON array with each element including customer, totalPoints, and rewards fields.
  + The JSON response should match the provided format, including details for each reward transaction.
* **Assertions:**
  + Verify that the HTTP status code is 200 OK.
  + Verify that the JSON response contains the correct customer.id, customer.name, totalPoints, and detailed rewards for each customer.
  + Validate that the JSON response includes the correct transaction details for all rewards.
* **Example Input:**
  + Request URL: /api/v1/getAllCustomer
  + Request Method: GET
* **Example Output:**

json

Response body

[  
 {  
 "customer": {  
 "id": 1,  
 "name": "John"  
 },  
 "totalPoints": 300,  
 "rewards": [  
 {  
 "transactionId": "501",  
 "transactionAmount": 200.0,  
 "points": 250,  
 "transactionDate": "2024-08-11T17:47:00.722000"  
 },  
 {  
 "transactionId": "502",  
 "transactionAmount": 100.0,  
 "points": 50,  
 "transactionDate": "2024-07-11T17:47:00.722000"  
 },  
 {  
 "transactionId": "503",  
 "transactionAmount": 45.0,  
 "points": 0,  
 "transactionDate": "2024-07-26T17:47:00.722000"  
 }  
 ]  
 },  
 {  
 "customer": {  
 "id": 2,  
 "name": "Shubham"  
 },  
 "totalPoints": 300,  
 "rewards": [  
 {  
 "transactionId": "504",  
 "transactionAmount": 200.0,  
 "points": 250,  
 "transactionDate": "2024-08-05T17:47:00.722000"  
 },  
 {  
 "transactionId": "505",  
 "transactionAmount": 100.0,  
 "points": 50,  
 "transactionDate": "2024-07-01T17:47:00.722000"  
 }  
 ]  
 },  
 {  
 "customer": {  
 "id": 3,  
 "name": "Joseph"  
 },  
 "totalPoints": 196,  
 "rewards": [  
 {  
 "transactionId": "511",  
 "transactionAmount": 89.0,  
 "points": 39,  
 "transactionDate": "2024-08-11T18:11:04.063000"  
 },  
 {  
 "transactionId": "512",  
 "transactionAmount": 129.0,  
 "points": 108,  
 "transactionDate": "2024-07-11T18:11:04.063000"  
 },  
 {  
 "transactionId": "513",  
 "transactionAmount": 99.0,  
 "points": 49,  
 "transactionDate": "2024-08-01T18:11:04.063000"  
 }  
 ]  
 }  
]  
**3. testGetAllCustomerRewards\_EmptyList**

* **Purpose:** To verify that the RewardController correctly handles a GET request for all customer rewards when no rewards are available and returns an empty list.
* **Setup:**
  + Mock the RewardService to return an empty list of Rewards objects.
* **Expected Behavior:**
  + The RewardController should return a ResponseEntity with a status of OK (HTTP 200).
  + The response body should be an empty JSON array.
* **Assertions:**
  + Verify that the HTTP status code is 200 OK.
  + Verify that the JSON response is an empty array.
* **Example Input:**
  + Request URL: /api/v1/getAllCustomer
  + Request Method: GET
* **Example Output:**

json

Response body

[]

#### **4. testGetRewardsByCustomerId\_CustomerNotFound**

* **Purpose:** To verify that the RewardController correctly handles a GET request for rewards by customer ID when the customer is not found and returns an appropriate response.
* **Setup:**
  + Mock the CustomerRepository to return an empty Customer object when searching by customer ID.
* **Expected Behavior:**
  + The RewardController should return a ResponseEntity with a status of OK (HTTP 200).
  + The response body should not contain any rewards data.
* **Assertions:**
  + Verify that the HTTP status code is 200 OK.
  + Verify that the JSON response does not contain any rewards data.
* **Example Input:**
  + Request URL: /api/v1/{customerId}/rewards
  + Request Method: GET
  + Path Variable: customerId = 5
* **Example Output:**

json

Response body

{}

3) Test Cases for RewardsServiceImplTest

**1. testGetRewardsByCustomerId**

* **Purpose:**
  + To test the getRewardsByCustomerId method in the RewardsServiceImpl class. This test verifies that the service correctly calculates reward points for a customer based on their transaction history over the past three months.
* **Setup:**
  + Mock objects for TransactionRepository and CustomerRepository are used.
  + Sample transaction data is provided for the last three months to simulate the customer's transaction history.
  + ById returns the mock customer.
  + findfindAllByCustomerIdAndTransactionDateBetween returns the list of mock transactions.
* **Execution:**
  + The getRewardsByCustomerId method is called with a sample customer ID (customerId = 1).
  + The service calculates rewards based on the mock transactions.
* **Assertions:**
  + Verify that the returned Rewards object contains:
  + The correct customerId (1).
  + The correct customer name ("John Doe").
  + The total reward points (382).
  + The correct number of transactions in the rewards list (4).
* **Example Input:**
  + Customer ID: 1
  + Mock Transactions:

120.0

170.0

100.0

101.0

* **Example Output:**

**Total Rewards**: 382

**Number of Transactions**: 4

### **2. testGetRewardsByCustomerId\_NoTransactions**

**Purpose**:

To test the getRewardsByCustomerId method when there are no transactions for the specified customer. This ensures the service handles such cases correctly.

**Setup**:

* **Mock Objects**: Mock TransactionRepository and CustomerRepository are used.
* **Sample Data**:
  + A mock Customer object with ID 1 and name "John Doe".
  + An empty list of transactions.

The when method is used to simulate responses from repositories:

* + findById returns the mock customer.
  + findAllByCustomerIdAndTransactionDateBetween returns an empty list.

**Execution**:

* Call the getRewardsByCustomerId method with customer ID 1.
* The service calculates rewards based on the absence of transactions.

**Assertions**:

* Verify that the returned Rewards object contains:
  + The correct customerId (1).
  + The correct customer name ("John Doe").
  + Total reward points (0).
  + An empty rewards list.

**Example Input**:

* **Customer ID**: 1
* **Mock Transactions**: (None)

**Example Output**:

* **Total Rewards**: 0
* **Number of Transactions**: 0

### **3. testCalculateRewardForTransaction\_BelowThreshold**

**Purpose**:

To test the calculateRewardForTransaction method to ensure it correctly returns 0 points for transactions below the minimum reward threshold.

**Setup**:

* **Mock Object**: A mock Transaction object with an amount below the threshold.
* **Sample Data**:
  + A Transaction object with an amount of 40.0 USD.

**Execution**:

* Call the calculateRewardForTransaction method with the low-value transaction.
* The service calculates the reward points for the transaction.

**Assertions**:

* Verify that the returned points are 0.

**Example Input**:

* **Transaction Amount**: 40.0

**Example Output**:

* **Reward Points**: 0

### **4. testCalculateRewardForTransaction\_AboveSecondLimit**

**Purpose**:

To test the calculateRewardForTransaction method to ensure it calculates reward points correctly for transactions above the second reward limit.

**Setup**:

* **Mock Object**: A mock Transaction object with an amount above the second reward limit.
* **Sample Data**:
  + A Transaction object with an amount of 150.0 USD.

**Execution**:

* Call the calculateRewardForTransaction method with the high-value transaction.
* The service calculates the reward points for the transaction.

**Assertions**:

* Verify that the points calculated match the expected result based on the transaction amount and reward calculation logic.

**Example Input**:

* **Transaction Amount**: 150.0

**Example Output**:

* **Reward Points**: 150 - Constant.secondRewardLimit + (50 \* 2)

150