**PROBLEM STATEMENT:**

A retailer offers a rewards program to its customers, awarding points based on each recorded purchase.

A customer receives 2 points for every dollar spent over $100 in each transaction, plus 1 point for every dollar spent between $50 and $100 in each transaction.

(e.g. a $120 purchase = 2x$20 + 1x$50 = 90 points).

Given a record of every transaction during a three month period, calculate the reward points earned for each customer per month and total.

• Solve using Spring Boot

• Create a RESTful endpoint

• Make up a data set to best demonstrate your solution

• Check solution into GitHub

**ADDITIONAL DETAILS:**

1. Unit and integration tests present in code.

2. Exception and negative test scenarios needs to be included.

3. Add readme file explaining the project and it’s structure and more details about the implementation.

4. Do not check in target folder.

5. Add java docs at class and method levels that should explain the purpose.

**FLOWCHART:**

A diagram of a service class

Description automatically generated  
  
**Client (Browser, Mobile App, etc.) HTTP Request** ------> **DispatcherServlet (Routing and Request Mapping)** --------> **Controller (RestController) (Mapping URLs & Handling HTTP Methods)** ------> **Service Layer (Optional) (Business Logic, Operations)** ------> **Repository Layer (Data Access/DB)** -------> **Response Mapping & HTTP Message Converter (JSON**)--------> **Client Receives HTTP Response (Data, Status)**

**TECHNICAL DESCRIPTION AND DETAILS:**

**PROJECT SETUP:**

1. Create Spring Boot Project using Spring Initializer.
2. Select particular Java version, give the Project Name, Add Maven Dependency and generate the zip file of the application.
3. Unzip the application file and open it in any IDE (Intelli J in my case). Build the project.

**POM.XML –** Use to save the dependencies which are required for the spring boot maven project. It create jar files of the added dependencies.

**APPLICATION.PROPERTIES:**  
It contains properties of H2 database which is used to connect application to the database.

**ENTITY CLASS:**

1. Consists of 2 entity/model classes:  
   **a) Customer:** It is created to save the user details and is associated with the transaction entity to save the transactions with respect to the particular customer.  
   **Attribute/Fields Names:**  
   🡪 customerId

🡪 name  
🡪 email  
 **b) Transaction:** It is created to save the transaction details such as amount and month wrt to the customer. It is also connected with the customer entity using customerId as the foreign Key where a particular customer can have multiple transactions.  
**Attribute/Fields Names:**  
🡪 customerId  
🡪 amount   
🡪 month

1. It also consists of multiple annotations which is used to avoid the boiler plate code using lombok dependency and annotations which are used to connect entity class to H2 database to create tables of those above-mentioned specific entity classes using JPA.

**ANNOTATIONS USED:**  
@Entity  
@Table(name="Customer")  
@NoArgsConstructor  
@AllArgsConstructor  
@Setter  
@Getter  
@ToString  
@Id  
@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
@JsonProperty("customerId")  
@Column(name = "customerId")  
@ManyToOne  
@JoinColumn(name = "customerId", insertable = false, updatable = false)

**JARS USED:**

1. Jackson annotation
2. Persistence api
3. lombok

**REPOSITORY CLASS:**

1. It is created as an Interface. Consists of 2 repository classes.  
   **a)** Customer Repository  
   **b)** Transaction Repository
2. It extends JPARepository of spring framework which takes generics as argument type that has model class name and datatype of the primary key of the model class. JPARepository has preexisting CRUD methods which can be used by the service layer with logic to fetch and save data respectively.
3. This interface also contains other abstract methods that we can define explicitly according to our application to implement it in the service class.

**SERVICE CLASS:**

1. **Class Definition and Autowired Repository:  
   a) @Service:** This annotation marks the RewardPointsCalculator class as a Spring service, meaning it will be managed by the Spring container as a bean. The @Service annotation is typically used for service layer components in Spring-based applications.  
   **b) @Autowired:** This annotation is used to inject the TransactionRepository bean into the RewardPointsCalculator class. TransactionRepository is presumably a repository interface (likely extending JpaRepository or a similar interface) used to query the database for transaction data.
2. **Method: calculatePoints  
   a)** This method calculates reward points for a single transaction.  
   **b)** **Logic**: Points are awarded based on the transaction amount:

* If the transaction amount is greater than 100, it awards 2 points for each dollar over 100.
* If the transaction amount is greater than 50 (but less than or equal to 100), it awards 1 point for each dollar over 50.
* Points are accumulated in the points variable, and the final value is returned.

**c)** If the transaction is null, the method throws an IllegalArgumentException.

1. **Method: calculateMonthlyPoints**This method calculates the total points for a customer in a specific month.

Logic:

* First, it checks if either the customerId or month is null, throwing an IllegalArgumentException if so.
* It then queries the database using the transactionRepository to fetch transactions for the given customerId and month. If a DataAccessException is thrown (indicating a database error), it wraps the exception in a RuntimeException.
* If no transactions are found, it throws a TransactionNotFoundException with a message indicating the absence of transactions for that customer in the specified month.
* If transactions are found, it calculates the total points by streaming over the list of transactions, applying the calculatePoints method to each transaction, and summing up the results.

1. **Method: calculateTotalPoints:**This method calculates the total points for a customer across all their transactions.

**Logic:**

* It first checks if the customerId is null, throwing an IllegalArgumentException if so.
* It then queries the database to fetch all transactions for the given customerId using transactionRepository. If a database error occurs, it wraps the DataAccessException in a RuntimeException.
* If no transactions are found, it throws a TransactionNotFoundException indicating that no transactions were found for the given customer.
* If transactions are found, it uses a stream to calculate and sum the points for all transactions using the calculatePoints method.

**CONTROLLER CLASS:**

1. Consists of 3 Rest Controller classes:  
   **a) Customer –** Contains Rest API and mapping to create and save customer in the database.  
   **b) Transaction -** Contains Rest API and mapping to create and save transaction details in the database.  
   **c) RewardPoints –** Contains Rest API and post mapping to calculate and save RewardPoints wrt customer and their transaction in the database. It gives the Reward Points in response as when we pass customerID and month as the request parameter in the url.

**CODE EXPLANATION:**

1. **Class Definition and Annotations**

* **@RestController:** This annotation combines @Controller and @ResponseBody. It indicates that the class is a Spring MVC controller, and the methods inside will return data directly to the HTTP response body (typically in JSON or XML format). It is specifically designed for RESTful web services.
* **@RequestMapping**("/api/rewardpoints"): This annotation defines the base URL path for all the endpoints in this controller. All the request mappings in this controller will be prefixed with /api/rewardpoints. For example:
* /api/rewardpoints/calculate
* /api/rewardpoints/calculate-month

This provides a consistent URL structure for the reward points API.

1. **Autowired Dependency Injection**

* @Autowired: This annotation is used to automatically inject the RewardPointsCalculator service into the controller. Spring's Dependency Injection (DI) mechanism takes care of creating the instance and wiring it to the controller, so you don't have to manually instantiate it.
* rewardCalculator: This is the injected instance of the RewardPointsCalculator service, which contains the business logic for calculating reward points.

1. **calculateTotalPoints Endpoint**

* @PostMapping("/calculate"): This annotation maps HTTP POST requests to the /api/rewardpoints/calculate URL to this method. This method is used to calculate the **total reward points** for a customer based on their transactions.
* @RequestParam String customerId: The customerId is passed as a query parameter in the POST request. This means that the client will need to send the customerId in the request body (in the case of a POST request) or as part of the URL parameters (in a query string, e.g., ?customerId=12345).
* rewardCalculator.calculateTotalPoints(customerId): This line calls the calculateTotalPoints method of the RewardPointsCalculator service, passing the customerId as a parameter. It returns the total reward points for the customer, which is then returned as the HTTP response.

1. **calculateMonthlyPoints Endpoint**

* @PostMapping("/calculate-month"): This annotation maps HTTP POST requests to the /api/rewardpoints/calculate-month URL to this method. This method calculates the **monthly reward points** for a customer in a given month.
* @RequestParam String customerId, @RequestParam String month: These annotations specify that the customerId and month should be passed as query parameters in the request (e.g., ?customerId=12345&month=2025-01). Both parameters are required for the calculation.
* rewardCalculator.calculateMonthlyPoints(customerId, month): This line calls the calculateMonthlyPoints method of the RewardPointsCalculator service, passing both customerId and month as parameters. It returns the total reward points accumulated by the customer in the specified month, which is then returned as the HTTP response.

**CUSTOM EXCEPTION CLASS:**

1. **Purpose of the TransactionNotFoundException Class:**

This custom exception class is useful in scenarios where you want to handle a specific case where a **transaction cannot be found** in your application. For example, when querying a database for a transaction, if the transaction does not exist, you can throw this exception to inform the caller about the missing transaction.

1. **extends RuntimeException**: This indicates that TransactionNotFoundException is a **runtime exception**. By extending RuntimeException, it means this exception is **unchecked**, meaning it doesn't have to be explicitly caught or declared in the method signature. Unchecked exceptions are typically used for errors that can happen during runtime (like trying to access a non-existent transaction in this case).
2. **Constructor: public TransactionNotFoundException(String message)**

* **TransactionNotFoundException(String message)**: This is the constructor for the custom exception. It takes a String message as a parameter. This message is typically used to provide more detailed information about the cause of the exception, such as why a transaction was not found.
* **super(message)**: The constructor calls the **superclass** (RuntimeException) constructor with the message parameter. This passes the message to the RuntimeException constructor, which stores the message in the exception. The super() method in this case ensures that the exception contains the message that can be retrieved later, typically for logging or providing information to the user.

**TEST CLASSES:**  
Created 3 Test classes:

1. **CustomerRewardPointsApplicationTests** – This is a set of unit and integration tests for the RewardPointsCalculator service in a Spring Boot application
2. **RewardPointsControllerTest** – This is a set of integration tests for the RewardPointsController class, written using **JUnit 5**, **Spring Boot Test**, and **MockWebServer** for simulating external interactions. The tests are designed to verify the HTTP request-response cycle of the RewardPointsController's endpoints.
3. **RewardPointsNegativeTests** – It consists of failure test scenarios for the reward points controller class

**CustomerRewardPointsApplicationTests**

1. **Test Class Annotations and Setup**

* **@SpringBootTest:** This annotation is used to indicate that the tests should run with Spring Boot's testing support. It ensures that the context of the Spring Boot application is loaded for integration tests.
* **@Mock:** This annotation is used to create a mock instance of the TransactionRepository. Mocking is essential for unit tests because it allows you to simulate database interactions without actually hitting a database.
* **@InjectMocks:** This annotation is used to inject the mocked dependencies (e.g., transactionRepository) into the RewardPointsCalculator class. Mockito automatically injects the mocked TransactionRepository into the RewardPointsCalculator instance.
* **@BeforeEach:** The setup() method annotated with @BeforeEach ensures that before each test is executed, MockitoAnnotations.openMocks(this) is called to initialize the mock objects.

1. **Test Methods:  
   🡪** It consists of both Success and failure scenarios Test cases based on the conditions of Rewards Points Logic defined in Service Class.

**RewardPointsControllerTest and RewardPointsNegativeTests:**

1. **Class-Level Annotations and Setup**

* **@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT):** This annotation is used to launch the entire Spring Boot application context for integration testing. The webEnvironment = RANDOM\_PORT ensures that the application will run with a randomly assigned port, which is used for testing web interactions.
* **@AutoConfigureWebTestClient:** This annotation enables the automatic configuration of WebTestClient, a reactive client that allows you to perform HTTP requests and assert responses. It helps to test the controller's endpoints effectively.
* **@Autowired**: This injects the WebTestClient into the test class. WebTestClient is used for making HTTP requests in integration tests.
* **@Mock**: This creates a mock of the RewardPointsCalculator service, so you can simulate its behavior without actually invoking the real service logic. This is useful for controlling the response in the tests.
* **@InjectMocks**: This annotation tells Mockito to inject the mocked RewardPointsCalculator into the RewardPointsController to test the controller's functionality. By using @InjectMocks, Mockito automatically injects the mock into the controller.
* **MockWebServer**: MockWebServer is used to simulate an external HTTP service. It's a useful tool for simulating RESTful APIs that the application might interact with. In this case, it's used to simulate the interaction with external services.
* **@BeforeEach**: The setup() method annotated with @BeforeEach is executed before each test method. It initializes the MockWebServer instance to simulate external interactions during the tests.

1. **Test Methods:**It consists of both Success and failure scenarios Test cases for the Controller Class.

**SUCCESS EXAMPLE:**  
**Purpose:** Test to verify the calculateTotalPoints endpoint of the RewardPointsController.

**Steps:**

**Setup:**

* A mock response is created for the RewardPointsCalculator service. The when(rewardPointsCalculator.calculateTotalPoints(customerId)).thenReturn(expectedPoints) simulates the behavior of the calculateTotalPoints method returning 150 points for the given customerId.
* A response is also queued up in the MockWebServer to simulate an external service that returns {"points": 150} with a 200 OK response code.

**MockWebServer:**

* The mock server is started, and a URL is generated for the endpoint /api/rewardpoints/calculate?customerId=12345.

**WebTestClient Request:**

* A POST request is sent to the mock server URL using WebTestClient, and the response is validated.
* The test checks that the response status is 200 OK using .expectStatus().isOk(), and it asserts that the points field in the JSON response equals 150 using .jsonPath("$.points").isEqualTo(expectedPoints).

**Shutdown:**

* The mock server is shut down to clean up after the test.

**FAILURE EXAMPLE:**

**Purpose:** This test simulates a situation where the customer ID is invalid. It ensures that the controller returns a 400 Bad Request response with the message "Invalid customerId".

**Steps:**

* Enqueue a mock response with a 400-status code, indicating an invalid customerId.
* Start the mock server, make the HTTP POST request with the invalid customerId, and assert the response status is 400 and the body contains "Invalid customerId".

**JAVA DOCS:**

1. [**https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html**](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html)
2. [**https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/package-summary.html**](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/package-summary.html)
3. [**https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/package-summary.html**](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/package-summary.html)
4. [**https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html**](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html)
5. [**https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Exception.html**](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Exception.html)
6. [**https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runtime.html**](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runtime.html)