[Technical Documentation: Spring Boot REST API with MongoDB as Datastore 1](#_Toc1831283816)

[1. Introduction 1](#_Toc2043119376)

[2. High-Level Architecture 1](#_Toc1417452542)

[Key Components 1](#_Toc1205692409)

[3. System Design Flow Diagram 1](#_Toc1324980635)

[Flow Diagram 1](#_Toc911297061)

[4. Endpoints Overview 2](#_Toc2015222122)

[Add Reward Points to Customer 2](#_Toc677658367)

[Retrieve Customer Rewards Points Total for the selected month 2](#_Toc1338745958)

[Retrieve Customer Rewards Points Total for last 3 months 2](#_Toc1055453594)

[5. Key Modules 2](#_Toc708778170)

[Controller Layer 2](#_Toc1710971167)

[6. Service Layer 4](#_Toc1553753921)

[7. Repository Layer 5](#_Toc424367464)

[8. MongoDB Configuration 6](#_Toc2014252331)

[9. Logging Configuration 6](#_Toc1050540779)

[10. Error Handling 6](#_Toc1520267184)

[11. Data Model 8](#_Toc2099636656)

[12. Testing 8](#_Toc1813696742)

[13. Deployment 10](#_Toc444479047)

[14. Future Enhancements 10](#_Toc269569040)

### Technical Documentation: Spring Boot REST API with MongoDB as Datastore

## Introduction

This documentation outlines the architecture, functionality, and flow of a Spring Boot REST API that uses MongoDB as its datastore. The API supports POST and GET operations and integrates various layers for efficient data management and interaction.

## **High-Level Architecture**

### **Key Components**

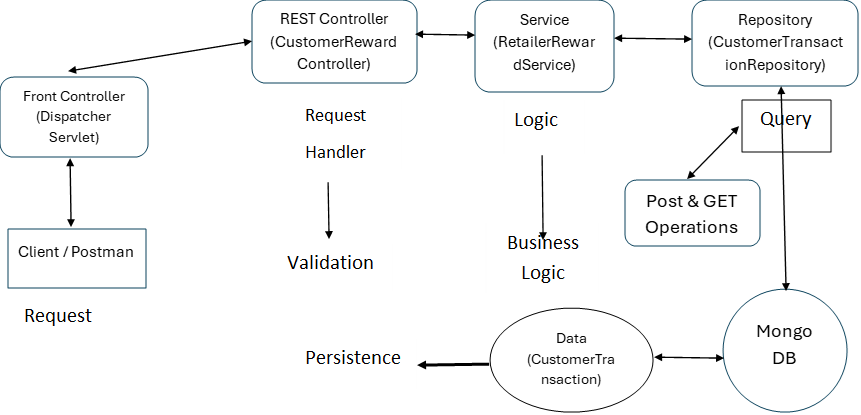
1. **Controller Layer**: Handles incoming HTTP requests and maps them to the appropriate service methods.
2. **Service Layer**: Contains business logic and interacts with the repository layer.
3. **Repository Layer**: Interfaces with MongoDB to perform data operations.
4. **MongoDB Database**: Stores the application’s data in a NoSQL format.
5. **Exception Handling**: Ensures proper error responses for invalid requests or failures.

## **System Design Flow Diagram**

Below is a high-level flow diagram illustrating how the system processes a request:

1. **Client (Frontend/Consumer)** sends an HTTP request.
2. The **REST** **Controller** processes the request and validates it.
3. The **Service Layer** executes business logic.
4. The **Repository Layer** interacts with the **MongoDB database**.
5. Then **Response** is returned back to the client.

### Flow Diagram



## **Endpoints Overview**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Request Body Example** |
| POST | /customerRewards/addPoints | Add Reward Points to Customer | {  "customerId" : "testCustomer1",  "amountSpent" : 110, } |
| GET | /customerRewards/{customerId }/monthly/{month}/{year} | Retrieve Customer Rewards Points Total for the selected month | Please refer to the Readme.md file for sample URL. |
| GET | /customerRewards / {customerId}/total | Retrieve Customer Rewards Points Total for last 3 months | Please refer to the Readme.md file for sample URL. |

## **Key Modules**

### **Controller Layer**

* **Responsibility**: Maps HTTP requests to methods and sends responses.
* **Implementation Example**:

@AllArgsConstructor

@RestController

@RequestMapping("/customerRewards")

**public** **class** CustomerRewardController {

**private** **final** RetailerRewardService retailerRewardService;

**private** **final** CustomerRequestUtil customerRequestUtil;

@PostMapping("/addPoints")

**public** ResponseEntity<String> addRewardPoints(@RequestBody CustomerRewardRequest customerRewardRequest) {

ResponseEntity<String> responseEntity;

**if** (customerRequestUtil.validAddRewardPointsRequest(customerRewardRequest)) {

retailerRewardService.postRewardPoints(customerRewardRequest);

responseEntity = ResponseEntity.*status*(HttpStatus.***CREATED***)

.body(RewardPointConstants.***REWARD\_POINTS\_SUCCESSFULLY\_ADDED\_RESPONSE***.getValue());

} **else** {

responseEntity = ResponseEntity.*badRequest*()

.body(RewardPointConstants.***INVALID\_CUSTOMER\_REQUEST\_RESPONSE***.getValue());

}

**return** responseEntity;

}

@GetMapping("/{customerId}/monthly/{month}/{year}")

**public** ResponseEntity<Object> getMonthlyRewardsTotal(@PathVariable String customerId, @PathVariable **int** month,

@PathVariable **int** year) {

ResponseEntity<Object> responseEntity;

**if** (customerRequestUtil.validMonthlyRewardsRequest(customerId, month, year)) {

**double** monthlyRewards = retailerRewardService.getMonthlyRewardsTotal(customerId, month, year);

responseEntity = ResponseEntity.*ok*(monthlyRewards);

} **else** {

responseEntity = ResponseEntity.*badRequest*()

.body(RewardPointConstants.***INVALID\_CUSTOMER\_REQUEST\_RESPONSE***.getValue());

}

**return** responseEntity;

}

@GetMapping("/{customerId}/total")

**public** ResponseEntity<Object> getRewardsTotal(@PathVariable String customerId) {

ResponseEntity<Object> responseEntity;

**if** (customerRequestUtil.validRewardsTotalRequest(customerId)) {

responseEntity = ResponseEntity.*ok*(retailerRewardService.getRewardsTotal(customerId));

} **else** {

responseEntity = ResponseEntity.*badRequest*()

.body(RewardPointConstants.***THREE\_MONTHLY\_REWARDS\_ERROR\_MSG***.getValue());

}

**return** responseEntity;

}

}

## Service Layer

* **Responsibility**: Contains the business logic.
* **Implementation Example**:

@AllArgsConstructor

@Service

**public** **class** RetailerRewardService {

**private** **final** CustomerTransactionRepository customerTransactionRepository;

**public** CustomerTransaction postRewardPoints(CustomerRewardRequest customerRewardRequest) {

CustomerTransaction customerTransaction = **new** CustomerTransaction(customerRewardRequest.getCustomerId(),

customerRewardRequest.getAmountSpent());

**return** customerTransactionRepository.save(customerTransaction);

}

**public** **double** getMonthlyRewardsTotal(String customerId, **int** month, **int** year) {

LocalDateTime startDate = LocalDateTime.*of*(year, month, 1, 0, 0);

LocalDateTime endDate = startDate.plus(1, ChronoUnit.***MONTHS***);

List<CustomerTransaction> transactions = customerTransactionRepository

.findTransactionsByCustomerIdAndDateBetween(customerId, startDate, endDate);

**double** rewardPointsTotal = 0;

**if** (Objects.*nonNull*(transactions) && !transactions.isEmpty()) {

rewardPointsTotal = calculatePoints(

(transactions.stream().mapToDouble(transaction -> transaction.getAmountSpent()).sum()));

}

**return** rewardPointsTotal;

}

**public** **int** getRewardsTotal(String customerId) {

LocalDateTime now = LocalDateTime.*now*();

LocalDateTime threeMonthsAgo = now.minus(3, ChronoUnit.***MONTHS***);

**int** rewardsTotal = 0;

List<CustomerTransaction> transactions = customerTransactionRepository

.findTransactionsByCustomerIdAndDate(customerId, threeMonthsAgo);

**if** (Objects.*nonNull*(transactions) && !transactions.isEmpty()) {

Map<Integer, Double> groupedByMonth = transactions.stream().collect(Collectors.*groupingBy*(t -> t.getTransactionDate().getMonthValue(), // Group by Month

Collectors.*summingDouble*(CustomerTransaction::getAmountSpent) // Sum the amounts by Month

));

**double** rewardsTotalGrouped = groupedByMonth.entrySet().stream().map(Map.Entry::getValue)

.collect(Collectors.*summingDouble*(Double::*valueOf*));

rewardsTotal = calculatePoints(rewardsTotalGrouped);

}

**return** rewardsTotal;

}

**private** **int** calculatePoints(**double** amount) {

**if** (amount > 100) {

**return** (**int**) ((amount - 100) \* 2 + 50); // 2 points for every dollar over 100 + 1 point for

// 50-100

} **else** **if** (amount > 50) {

**return** (**int**) ((amount - 50)); // 1 point for every dollar over 50

} **else** {

**return** 0; // No points

}

}

}

### Repository Layer

* **Responsibility**: Interfaces with MongoDB for data operations.
* **Implementation Example**:

**public** **interface** CustomerTransactionRepository **extends** MongoRepository<CustomerTransaction, String> {

**void** deleteByCustomerId(String customerId);

@Query("{'customerId': ?0, 'transactionDate': { $gte: ?1 } }")

List<CustomerTransaction> findTransactionsByCustomerIdAndDate(String customerId, LocalDateTime fromDate);

@Query("{'customerId': ?0, 'transactionDate' : { $gte: ?1, $lte: ?2 } }")

List<CustomerTransaction> findTransactionsByCustomerIdAndDateBetween(String customerId, LocalDateTime startDate,

LocalDateTime endDate);

}

1. MongoDB Configuration

* **Responsibility**: Connects the Spring Boot application to MongoDB.
* **Configuration in application.properties**:

spring.application.name=customer-rewards-springboot-application

spring.data.mongodb.database=retailerDatabase

spring.data.mongodb.uri=mongodb://admin:admin@localhost:27017/

1. Logging Configuration

* **Configuration in application.properties**:

logging.level.root=INFO

logging.level.com.sandusky.retailer=DEBUG

logging.file.name=logs/application.log

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n

logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n

## **Error Handling**

* **Global Exception Handling**: @RestControllerAdvice is used for centralized error management.
* **Example**:

@RestControllerAdvice

@Slf4j

**public** **class** GlobalExceptionHandler {

// Handle validation errors

@ExceptionHandler(MethodArgumentNotValidException.**class**)

**public** ResponseEntity<Map<String, String>> handleValidationExceptions(MethodArgumentNotValidException ex) {

Map<String, String> errors = **new** HashMap<>();

ex.getBindingResult().getFieldErrors()

.forEach(error -> errors.put(error.getField(), error.getDefaultMessage()));

***log***.error(RewardPointConstants.***METHOD\_ARGUMENT\_NOT\_VALID\_EXCEPTION***.getValue(), ex);

**return** **new** ResponseEntity<>(errors, HttpStatus.***BAD\_REQUEST***);

}

// Handle MongoDB Duplicate Key exceptions

@ExceptionHandler(DuplicateKeyException.**class**)

**public** ResponseEntity<String> handleDuplicateKeyException(DuplicateKeyException ex) {

**return** **new** ResponseEntity<>(RewardPointConstants.***DUPLICATE\_KEY\_ERROR***.getValue() + ex.getMessage(),

HttpStatus.***CONFLICT***);

}

// Handle MongoDB Query Execution exceptions

@ExceptionHandler(MongoQueryException.**class**)

**public** ResponseEntity<String> handleMongQueryException(MongoQueryException ex) {

**return** **new** ResponseEntity<>(RewardPointConstants.***MONGO\_QUERY\_EXCEPTION\_MSG***.getValue() + ex.getMessage(),

HttpStatus.***CONFLICT***);

}

// Handle resource not found (custom exception)

@ExceptionHandler(NoHandlerFoundException.**class**)

**public** ResponseEntity<String> handleResourceNotFoundException(NoHandlerFoundException ex) {

**return** **new** ResponseEntity<>(RewardPointConstants.***ENDPOINT\_NOT\_FOUND\_MSG***.getValue().concat(ex.getRequestURL()),

HttpStatus.***NOT\_FOUND***);

}

// Handle general server errors

@ExceptionHandler(Exception.**class**)

**public** ResponseStatusException handleGenericException(Exception ex) {

***log***.error(RewardPointConstants.***EXCEPTION\_DURING\_PROCESSING\_REQUEST***.getValue(), ex);

**return** **new** ResponseStatusException(HttpStatusCode.*valueOf*(HttpStatus.***INTERNAL\_SERVER\_ERROR***.value()),

ex.getLocalizedMessage(), ex);

}

}

## **Data Model**

* **Example Resource Entity**:

@Setter

@Getter

@Document(collection = "customerTransaction")

**public** **class** CustomerTransaction {

@Id

**private** String id;

**private** String customerId;

**private** **double** amountSpent;

**private** LocalDateTime transactionDate;

**public** CustomerTransaction(String customerId, **double** amountSpent) {

**this**.customerId = customerId;

**this**.amountSpent = amountSpent;

transactionDate = LocalDateTime.*now*();

}

}

## **Testing**

**Unit Testing**: Use JUnit 5 for testing service and repository layers.

**Integration Testing**: Use Spring Boot Test for end-to-end tests.

* **Example Unit Test**:

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.***RANDOM\_PORT***)

**class** CustomerRewardControllerIntegrationTest {

@LocalServerPort

**private** **int** port;

@Autowired

**private** TestRestTemplate restTemplate;

**private** CustomerRewardRequest customerRewardRequestOne;

**private** CustomerRewardRequest customerRewardRequestTwo;

**private** CustomerRewardRequest badCustomerRewardRequest;

@Autowired

**private** CustomerTransactionRepository customerTransactionRepository;

@BeforeEach

**void** setUp() **throws** Exception {

// Setup test data

customerRewardRequestOne = **new** CustomerRewardRequest(RewardPointConstants.***CUSTOMER\_ONE***.getValue(), 200d);

customerRewardRequestTwo = **new** CustomerRewardRequest(RewardPointConstants.***CUSTOMER\_TWO***.getValue(), 100d);

badCustomerRewardRequest = **new** CustomerRewardRequest(RewardPointConstants.***CUSTOMER\_THREE***.getValue(), **null**);

}

@AfterEach

**void** tearDown() **throws** Exception {

customerRewardRequestOne = **null**;

customerRewardRequestTwo = **null**;

customerTransactionRepository.deleteByCustomerId(RewardPointConstants.***CUSTOMER\_ONE***.getValue());

customerTransactionRepository.deleteByCustomerId(RewardPointConstants.***CUSTOMER\_TWO***.getValue());

}

@Test

**public** **void** testAddRewardPoints() {

String url = RewardPointConstants.***HTTP\_LOCALHOST***.getValue() + port

+ RewardPointConstants.***CUSTOMER\_REWARDS\_ADD\_POINTS\_URI***.getValue();

ResponseEntity<String> addRewardPointsResponse = restTemplate.postForEntity(url, customerRewardRequestTwo,

String.**class**);

*assertEquals*(HttpStatus.***CREATED***, addRewardPointsResponse.getStatusCode());

}

@Test

**public** **void** testAddRewardPointsBadRequest() {

String url = RewardPointConstants.***HTTP\_LOCALHOST***.getValue() + port

+ RewardPointConstants.***CUSTOMER\_REWARDS\_ADD\_POINTS\_URI***.getValue();

ResponseEntity<String> addRewardPointsResponse = restTemplate.postForEntity(url, badCustomerRewardRequest,

String.**class**);

*assertEquals*(HttpStatus.***BAD\_REQUEST***, addRewardPointsResponse.getStatusCode());

}

}

## **Deployment**

**Local Environment**:

* 1. Run the application using mvn spring-boot:run.

**Cloud Deployment**:

* 1. Containerize the application using Docker.
  2. Deploy to platforms like AWS, Azure, or Google Cloud with minimal changes.

## Future Enhancements

Implement caching with Redis for improved performance.

Add authentication using Spring Security.

Use a custom DTO layer for API responses.

This documentation provides a structured guide to understanding, setting up, and interacting with the Spring Boot REST API. For further queries, please refer to the [Spring Boot documentation](https://spring.io/projects/spring-boot) and [MongoDB documentation](https://www.mongodb.com/docs/).