**Documentation for Customer Management Service**

This documentation provides setup instructions and details of the code architecture for the Customer Management Service.

**Setup Instructions**

**Prerequisites**

1. **JDK 17-** Java development
2. **Database**: Use a MySQL database-8 version or a compatible relational database configured SSL for secure connection.
3. **Spring Boot Environment**:use spring initializer and eclipse IDE use pom.xml for maven dependencies.
4. **Application Configuration**: Provide the necessary database connection details in application.properties or application.yml.
5. **Redis**- version 3.2.100 runs on localhost: 6379 used redis-cli for cache logs by commands monitor. Flushall to delete all cache logs.

**Steps to Set Up**

1. **Clone Repository**: Clone the repository =git clone <repository-url>
2. **Build the Application**: mvn clean install
3. **Database Setup**:
   * Create a database named **test**.
   * Ensure the table **customerdetails** is created automatically by Hibernate during application startup.
4. **Run the Application**: mvn spring-boot:run
5. **Access the APIs**: The application runs on http://localhost:9000 by default. Use tools like Postman to test the endpoints.

**Code Architecture Details**

**1. Entity Layer**

The CustomerDetails entity class maps to the customerdetails table in the database.

**Code Highlights:**

* **Annotations**:
  + @Entity, @Table: Maps this class to a database table.
  + @Data: Generates boilerplate code like getters and setters.
  + @Id, @GeneratedValue: Marks the primary key and defines its generation strategy.
* **Field Validations**:
  + @NotBlank, @Email, @Size, @Min, @Max: Ensure data integrity.
* **Custom Converter**:
  + @Convert: Converts the countries field to and from JSON using JpaConverterJson.

**Example:**

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotBlank(message = "Name is mandatory")

private String name;

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

private String email;

**2. Controller Layer**

The CustomerControllerImpl class handles HTTP requests and responses for customer-related operations.

**API Endpoints:**

* **Create Customer**: POST /api/customers
* **Get Customer by ID**: GET /api/customers/{id}
* **Get All Customers**: GET /api/customers/all
* **Update Customer**: PUT /api/customers/{id}
* **Delete Customer**: DELETE /api/customers/{id}
* **Custom Queries**:
  + GET /api/customers/check-customer-experience/{Designation}
  + GET /api/customers/check-country-experience?country=value

**Example:**

@PostMapping

public ResponseEntity<CustomerDetails> createCustomer(@RequestBody CustomerDetails customer) {

CustomerDetails createdCustomer = customerservice.createCustomer(customer);

return new ResponseEntity<>(createdCustomer, HttpStatus.CREATED);

}

**3. Service Layer**

The CustomerServiceImpl class implements business logic and interacts with the repository layer.

**Features:**

* **Caching**: @Cacheable, @CachePut, @CacheEvict annotations for performance optimization.
* **Retry and Circuit Breaker**: Resilient mechanisms using @Retry and @CircuitBreaker annotations.
* **Asynchronous Operations**: @Async annotation for handling concurrent tasks.
* **Transaction Management**: @Transactional for atomic database operations.

**Example:**

@Override

@CachePut(value = "createCustomer", key = "#customer")

@Retry(name = "createCustomerRetry", fallbackMethod = "fallbackForCreateCustomer")

@CircuitBreaker(name = "createCustomerCircuitBreaker", fallbackMethod = "fallbackForCreateCustomer")

public CustomerDetails createCustomer(CustomerDetails customer) {

return customerdetailsrepo.save(customer);

}

**4. Repository Layer**

The CustomerDetailsRepo interface extends JpaRepository to perform CRUD operations.

**Example:**

@Repository

public interface CustomerDetailsRepo extends JpaRepository<CustomerDetails, Long> {

List<CustomerDetails> findCustomersWithExpLessThanFive(String designation);

List<CustomerDetails> findByExperienceyr(int years);

}

**5. Utility Features**

* **Validation**: Ensures data integrity at the input level using annotations like @NotBlank and @Email.
* **Custom Exception Handling**: CustomerNotFoundException for specific error scenarios.
* **Logging**: Provides detailed logs for monitoring and debugging.

**1. Layered Architecture (Architectural Pattern)**

My application is divided into the following layers:

* **Controller Layer**: Handles HTTP requests and delegates processing to the service layer.
* **Service Layer**: Contains business logic and interacts with the repository layer.
* **Repository Layer**: Manages direct interactions with the database.

**2. Singleton Design Pattern**

* **Where Used**:
  + Spring beans **(e.g., CustomerControllerImpl, CustomerServiceImpl, CustomerDetailsRepo**) are by default instantiated as singletons by the Spring IoC container.

**3. Factory Pattern (via Spring Data JPA)**

* **Where Used**:
  + The **JpaRepository** provided by Spring Data acts as a factory for creating repository implementations at runtime.

**4. Proxy Design Pattern**

* **Where Used**:
  + Spring's **AOP (Aspect-Oriented Programming)** mechanisms, like caching **(@Cacheable, @CacheEvict**) and retry logic **(@Retry**), internally use dynamic proxies to wrap and enhance method execution.

**5. Template Method Pattern**

* **Where Used**:
  + By extending **Spring’s JpaRepository**, we can leverage predefined methods (**save, findById, findAll**) that follow a fixed algorithm template for database operations. We can customize behavior by overriding methods if needed.

**6. Circuit Breaker Pattern**

* **Where Used**:
  + The use of **@CircuitBreaker** ensures that fallback methods are triggered when dependent services fail or time out.

**7. Builder Pattern (for JSON Conversion)**

* **Where Used**:
  + The **@Convert** annotation and custom JSON converters suggest the use of the Builder Pattern for converting complex objects (like countries) into JSON strings and vice versa.

**API Summary**

| **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| POST | /api/customers | Create a new customer |
| GET | /api/customers/{id} | Retrieve a customer by ID |
| GET | /api/customers/all | Retrieve all customers |
| PUT | /api/customers/{id} | Update a customer by ID |
| DELETE | /api/customers/{id} | Delete a customer by ID |
| GET | /api/customers/check-customer-experience/{Designation} | Check experience based on designation |
| GET | /api/customers/check-country-experience?country=value | Get customers based on country and experience |

**Logging and Monitoring**

* **SLF4J Logger**: Logs are generated for all major events, including API calls, database interactions, and fallback mechanisms.
* **Execution Time Logging**: Monitors method execution times for performance analysis.

**Performance Reports**: sequential vs parallel calls example code

| **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| GET | /performance/compare | Compare sequential vs parallel execution performance. |

**For Spring Boot Actuator endpoints:** Measure and log performance , metrics and health of my application

| **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| GET | /actuator/mappings | View all mapped endpoints and their corresponding handlers. |
| GET | /actuator/circuitbreakers | Monitor and manage circuit breaker states in the application. |
| GET | /actuator/health | Check the health status of the application and its dependencies. |
| GET | /actuator/beans | View all Spring beans registered in the application context. |
| GET | /actuator/loggers | Manage and view logging levels of the application dynamically. |

**Test Cases**

* **Added Unit and integration Tests**: Implement comprehensive Junit and integration tests for all layers.