✅ Software Requirements Specification (SRS)

🛎️ Notification Service (Spring Boot + RabbitMQ)

---

1. Introduction

1.1 Purpose

This document defines the requirements for building a Notification Service that sends email, SMS, and in-app messages using REST APIs. It uses RabbitMQ to asynchronously process notifications and includes retry logic for failure handling.

1.2 Scope

The Notification Service:

- Exposes APIs for sending and retrieving notifications.
- Sends notifications via Email, SMS, and In-App.
- Utilizes **RabbitMQ** for decoupling and asynchronous processing.
- Persists notification metadata and delivery status.
- Handles automatic retries with exponential backoff.

---

2. Overall Description

System Architecture: - Producer: REST Controller receives notification requests. - Consumer: Listens to RabbitMQ queue and processes notifications. - Broker: RabbitMQ for async decoupling. - Storage: Metadata persisted with retry support.

Users: API Clients - Backend services triggering requests End Users - Recipients of email, SMS, in-app

User Class	Description
API Clients	Backend services triggering notifications
End Users	Receivers of notifications (email/SMS/in-app)

---

3. Specific Requirements

3.1 Functional Requirements

3.1.1 Send Notification

`POST /notifications`
Sends a notification by placing it into a message queue.

3.1.2 Get User Notifications

`GET /users/{id}/notifications`
Retrieves all sent notifications for a user.

3.2 Non-Functional Requirements

- High availability (99.9%)
- Asynchronous processing
- Secure APIs (token-based)
- Scalable and retry-enabled

---

4. Dependencies

Dependency	Purpose
Spring Boot	Core application framework for creating standalone, production-grade Spring-based applications.
Spring Web	Enables the development of REST APIs and HTTP services. *(Version used: 4.22.0)*
JavaFX Controls	Provides UI components (e.g., buttons, tables) for building rich desktop interfaces using JavaFX.
JavaFX FXML	Supports the use of FXML files for building UIs declaratively in JavaFX applications.
JUnit Jupiter (API)	Provides APIs for writing unit tests using the JUnit 5 framework.

```
<dependencies>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-controls</artifactId>
        <version>18.0.2</version>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-fxml</artifactId>
        <version>18.0.2</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>4.22.0</version>
    </dependency>

    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>${junit.version}</version>
```

5. Database Design (ER Diagram)

ER Diagram

Unset

---

6. System Design

🔄 6.1 Message Queue Monitoring (RabbitMQ Console)

The Notification Service uses RabbitMQ for decoupled and asynchronous message handling. Below is a view of the queues from the RabbitMQ management console:

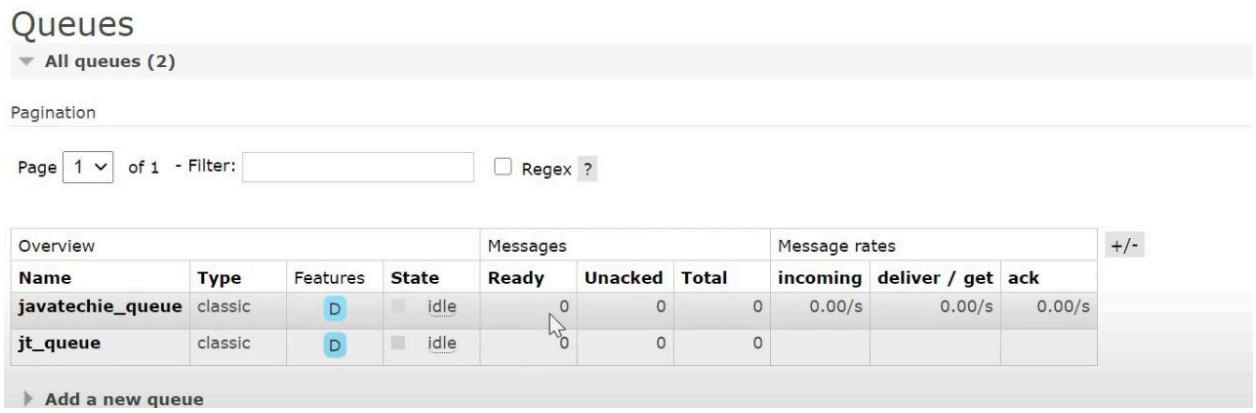
Queues Observed:

- `javatechie_queue`
- `jt_queue`

Key Metrics Displayed:

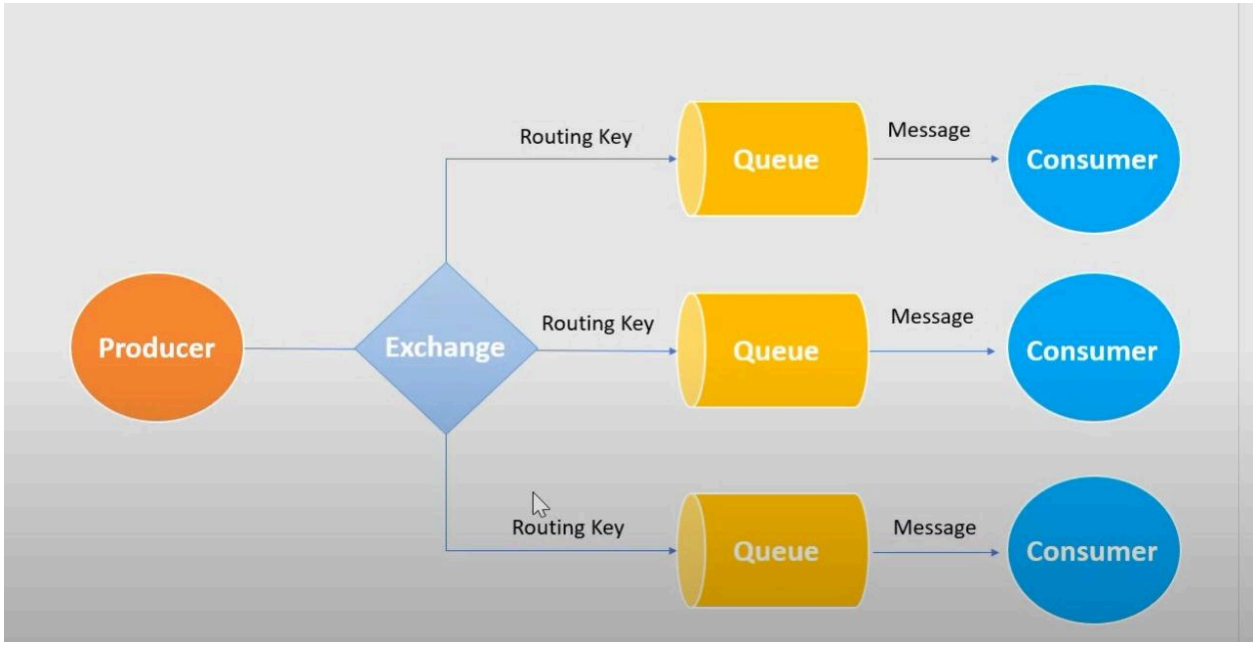
- **State**: Indicates whether the queue is active or idle.
- **Messages Ready/Unacked/Total**: Tracks message lifecycle within the queue.
- **Message Rates**: Shows throughput for incoming, delivery, and acknowledgment.

This visibility helps in:



- Monitoring real-time queue health.
- Debugging stalled or failed message delivery.
- Scaling consumers based on queue backlog.

🔧 6.2 High-Level Architecture



✅ Key Components:

- **Producer**: REST Controller that publishes jobs.

```java
@PostMapping("/{companyName}")
public String bookOrder(@RequestBody Order order, @PathVariable String companyName) {
    order.setOrderId(UUID.randomUUID().toString());
    //restaurantservice
    //payment service
    OrderStatus orderStatus = new OrderStatus(order, "PROCESS", "You have a notification from  " + companyName);
    template.convertAndSend(MessagingConfig.EXCHANGE, MessagingConfig.ROUTING_KEY, orderStatus);
    return "Success !!";
}
```

- `@PostMapping("/{companyName}")`: Exposes a REST endpoint for clients to submit orders, where `companyName` is passed as a path variable.
- `UUID.randomUUID()`: Assigns a unique ID to the order.
- `OrderStatus`: A message object containing order and status info.
- `template.convertAndSend(...)`: Publishes the message to RabbitMQ using the defined exchange and routing key.
- The system is designed to decouple message creation and processing, aligning with the microservice and asynchronous messaging architecture.

This controller acts as the **entry point** for other backend services to **initiate a notification flow**, effectively functioning as the **producer component** in the architecture.

- **Consumer**: Worker service that listens to RabbitMQ and processes the job.

---

7. Postman Collection

Collection: `NotificationService.postman_collection.json`

Unset

```
{
  "info": {
    "name": "NotificationService",
    "_postman_id": "abc123-def456",
    "description": "APIs for sending and retrieving
notifications.",
    "schema":
"https://schema.getpostman.com/json/collection/v2.1.0/collection.
json"
  },
  "item": [
    {
      "name": "Send Notification",
      "request": {
        "method": "POST",
        "header": [{ "key": "Content-Type", "value":
"application/json" }],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"userId\": \"123\",\n  \"type\":
\"EMAIL\",\n  \"title\": \"Welcome!\",\n  \"message\": \"Thanks
for signing up.\" \n}"
        },
        "url": {
          "raw": "http://localhost:8080/notifications",
          "protocol": "http",
          "host": ["localhost"],
          "port": "8080",
          "path": ["notifications"]
        }
      }
    },
    {
      "name": "Get Notifications by User",
      "request": {
        "method": "GET",
```

```
        "url": {
          "raw": "http://localhost:8080/users/123/notifications",
          "protocol": "http",
          "host": ["localhost"],
          "port": "8080",
          "path": ["users", "123", "notifications"]
        }
      }
    }
  ]
}
```

---

8. Retry Logic

- **Max Retries**: 3
- **Backoff Strategy**: Exponential
    - Attempt 1: 10s
    - Attempt 2: 30s
    - Attempt 3: 1min
- On max retry failure, mark notification as `FAILED`.

---

9. Future Enhancements

🔌 9.1 API Integration Testing using Postman

To verify the functionality of the RabbitMQ-based messaging system, Postman is used to simulate API requests to the Order Service.

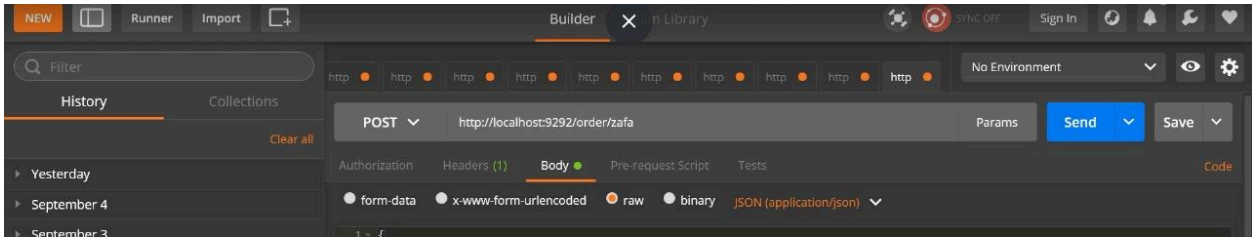
Request Example:

- **Method:** POST
- **URL:** `http://localhost:9292/order/zafa`
- **Content-Type:** `application/json`
- **Body:** Raw JSON payload sent to simulate order placement and message publication.

This request triggers the controller logic which publishes a message to RabbitMQ, sending it to a configured exchange and routing key.

This aids in:

- Verifying endpoint behavior
- Testing message production without UI
- Observing message flow to RabbitMQ queues



🔧 10. Configuration Properties (New Section)

This section defines system-level and runtime configurations for the Notification Service.

Property	Value	Purpose
`distributionUrl`	Maven Wrapper URL	Specifies the Maven distribution used by the wrapper.

`wrapperUrl`	Maven Wrapper JAR URL	Defines where the Maven Wrapper JAR is downloaded from.
`server.port`	`9292`	Custom port on which the Spring Boot application runs.

```
distributionUrl=https://repo.maven.apache.org/maven2/org/apache/maven/apac
wrapperUrl=https://repo.maven.apache.org/maven2/org/apache/maven/wrapper/m
server.port=9292
```