How To Build a Full Featured, Event Driven, Notification Service With Spring Boot and Redis

Introduction

An event can be defined as an action or status change, that can be observed and further trigger subsequent actions to complete a given task. At the very base of event driven system is the concept of producing events and listening for said event to make an informed decision on what to do next. One way of achieving/implementing event driven architectures is by using PUB/SUB event buses/or protocols to communicate between services. Event driven systems are highly scalable as they reduce the load of having to get immediate responses for each request.

What You'll Learn

- Basic Implementation of PUB/SUB queues
- Redis Basic Configuration
- Spring Boot Integration Redis

Prerequisites

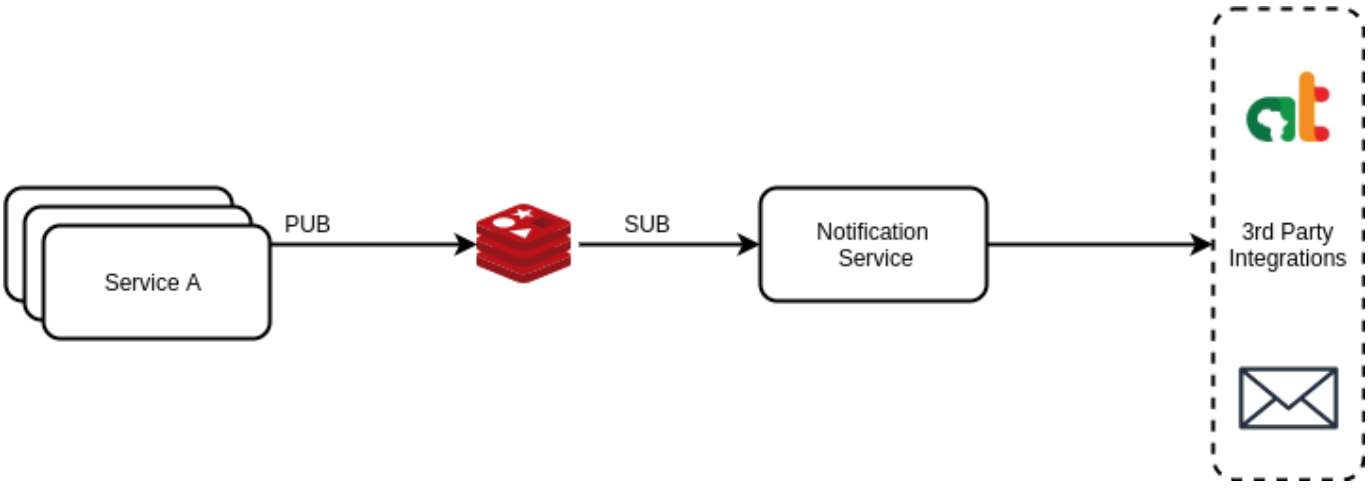
To follow this tutorial, you should have basic knowledge of working with:

- Spring Boot
- Docker
- Git

Key Tools and Components

- Redis
- Pub / Sub

The workflow



The figure above shows components involved: Service A (Generates events), Notifications Service, Redis, 3rd Party Integrations. The event processor (Notification Service) is what reads the events from the log and

processes i.e sending the messages over email or SMS.

Environment Set Up

Redis

Spring Boot

Ensure you have JDK and Maven installed. Head over to Spring Initializr and create your SpringBoot project with the following options.

Step 1 Gradle

```
plugins {
    id 'org.springframework.boot' version '2.3.3.RELEASE'
    id 'io.spring.dependency-management' version '1.0.10.RELEASE'
    id 'com.github.johnrengelman.processes' version '0.5.0'
    id 'org.springdoc.openapi-gradle-plugin' version '1.3.0' // springdoc API
documentation
    id 'java'
}

group = 'com.decoded'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
    gradlePluginPortal()
    maven {
        url 'http://dl.bintray.com/africastalking/java'
    }
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-redis'
    implementation 'org.springframework.boot:spring-boot-starter-data-rest'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'com.fasterxml.jackson.core:jackson-databind'
    implementation 'redis.clients:jedis'
    implementation 'org.springdoc:springdoc-openapi-ui:1.4.6'
    implementation 'org.springdoc:springdoc-openapi-data-rest:1.4.6'
    implementation 'com.africastalking:core:3.4.2'

    compileOnly 'org.projectlombok:lombok'
```

```
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    runtimeOnly 'mysql:mysql-connector-java'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }
    testImplementation 'io.projectreactor:reactor-test'
}

test {
    useJUnitPlatform()
}
```

Step 2 Properties

```
spring.datasource.url=jdbc:mysql://${DB_HOST:127.0.0.1}:${DB_PORT:3306}/${MYSQL_DA
TABASE:messaging.decoded.application}

spring.datasource.username=${MYSQL_USER:null}
spring.datasource.password=${MYSQL_USER_PASSWORD:null}

spring.redis.host=${REDIS_HOST:127.0.0.1}
spring.redis.port=6379
spring.redis.password=${REDIS_PASSWORD:null}


decoded.africastalking.username=${AT_USER_NAME:null}
decoded.africastalking.api-key=${AT_API_KEY:null}
decoded.africastalking.default-from=${AT_DEFAULT_SENDER:AFRICASTKNG}

server.port=${SERVER_PORT:5768}

springdoc.api-docs.enabled=true
springdoc.swagger-ui.path=/swagger-ui.html

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Step 3 Configuration Let's start by adding the redis configuration.

First Well Define a ***MessageListenerAdapter***. this delegates the handling of messages to target listener methods through reflection, with flexible message type conversion. This bean defines the subscriber in the pub-sub messaging model, currently the handler function is defined as "onMessage" you can customize this to something else.

```
@Bean
MessageListenerAdapter messageListener(RedisMessageSubscriber subscriber) {
    return new MessageListenerAdapter(subscriber, "onMessage");
}
```

Second we'll define the ***RedisMessageListenerContainer*** which provides asynchronous behaviour for Redis message Listeners, It handles the low level details of listening, converting and message dispatching

```
@Bean
RedisMessageListenerContainer redisContainer(MessageListenerAdapter
listenerAdapter) {
    final RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
    container.setConnectionFactory(jedisConnectionFactory());
    container.addMessageListener(listenerAdapter, topic());
    return container;
}
```

Next we'll set up a topic to which the publisher should send messages, and to which the subscriber will listen to.

```
@Bean
ChannelTopic topic() {
    return new ChannelTopic("pubsub:queue");
}
```

We'll then move on to create a bean using the custom ***MessagePublisher*** interface. This will allow us to have a generic message-publishing API, and have the Redis implementation get a redisTemplate and topic as its constructor arguments

```
@Bean
MessagePublisher redisPublisher() {
    return new RedisMessagePublisher(redisTemplate(), topic());
}
```

Next is to define Redis template to allow key/value data handlling this will also allow us define the serialization scheme. The below configuration will Serialize the values to Json before publishing, this will also be used to deserialize at the subscriber instance.

```
@Bean
public RedisTemplate<String, Object> redisTemplate() {
    final RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();
    redisTemplate.setConnectionFactory(jedisConnectionFactory());
    redisTemplate.setKeySerializer(new StringRedisSerializer());
    redisTemplate.setValueSerializer(new GenericJackson2JsonRedisSerializer());
    return redisTemplate;
}
```

Finally to close of the confuguration we'll define the ***JedisConnectionFactory*** Bean to allow connection to the Redis instance based on predefined credentials.

```
@Value("${spring.redis.host}")
private String hostName;

@Value("${spring.redis.port}")
private int port;

@Value("${spring.redis.password}")
private String password;

@Bean
JedisConnectionFactory jedisConnectionFactory() {
    RedisStandaloneConfiguration redisStandaloneConfiguration = new
RedisStandaloneConfiguration(this.hostName, this.port);
    redisStandaloneConfiguration.setPassword(RedisPassword.of(this.password));
    return new JedisConnectionFactory(redisStandaloneConfiguration);
}
```

Step 4 Entity and Redis Dto Lets define our Message entity as well as its DAO(Repository)

```
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;

import lombok.AccessLevel;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@Entity
@Getter(value = AccessLevel.PUBLIC)
@Setter(value = AccessLevel.PUBLIC)
@ToString
public class Message {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
```

```
    private String recepient;

    private MessageStatus status;

    private MessageType type;

    private Integer retries = 0;

    @Column(nullable = true)
    private String subject;

    @Column(columnDefinition = "text")
    private String content;

    @CreatedDate
    @Temporal(TemporalType.TIMESTAMP)
    protected Date creationDate;

    @LastModifiedDate
    @Temporal(TemporalType.TIMESTAMP)
    protected Date lastModifiedDate;
}

public enum MessageStatus {
    PENDING, SENT, FAILED
}

public enum MessageType {
    SMS, MAIL
}
```

The Redis Dto

```
import java.io.Serializable;

import {your-packaging}.MessageStatus;
import {your-packaging}.MessageType;

import org.springframework.data.redis.core.RedisHash;

import lombok.AccessLevel;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@RedisHash("messages")
@Getter(value = AccessLevel.PUBLIC)
@Setter(value = AccessLevel.PUBLIC)
@ToString
public class MessageRedisDto implements Serializable {
    private String id;
```

```
    private static final long serialVersionUID = 1L;

    private String recepient;

    private MessageStatus status;

    private MessageType type;

    private Integer retries;

    private String subject;

    private String content;
}
```

Since we have spring data rest within the installed packages well expose some methods directly from the repository.

```
import {your-packaging}.Message;
import {your-packaging}.MessageStatus;
import {your-packaging}.MessageType;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.rest.core.annotation.Description;
import org.springframework.data.rest.core.annotation.RestResource;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
@Transactional
public interface MessageRepository extends CrudRepository<Message, Long> {

    @Override
    @RestResource(exported = false)
    <S extends Message> S save(S entity); // hide PUT & POST methods from the REST
API

    @Override
    @RestResource(exported = false)
    void delete(Message message);// hide DELETE methods from the REST API

    @RestResource(path = "filter/status", description = @Description(value = "Find
By Status"))
    Page<Message> findByStatus(MessageStatus status, Pageable pageable);

    @RestResource(path = "eligible/to/send", description = @Description(value =
""))
    Page<Message> findByStatusAndRetriesLessThan(MessageStatus status, Integer
retries, Pageable pageable);
```

```
    @RestResource(path = "filter/type/status", description = @Description(value =
""))
    Page<Message> findByTypeAndStatus(MessageType type, MessageStatus status,
Pageable pageable);

}
```

Step 5 Message Handlers

Define the Event Processor

The Event Processor performs tasks triggered by events in this case out task will be actually sending the notification using 3rd Party integrations.

```
import java.io.IOException;
import java.util.List;

import com.africastalking.AfricasTalking;
import com.africastalking.SmsService;
import com.africastalking.sms.Recipient;
import {your-packaging}.RedisMessagePublisher;
import {your-packaging}.MessageStatus;
import {your-packaging}.MessageType;
import {your-packaging}.MessageRedisDto;
import {your-packaging}.MessageRepository;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

@Service
public class SmsMessagingService {

    @Autowired
    private MessageRepository messageRepository;

    @Value("${decoded.africastalking.username}")
    private String africasTalkingUserName;

    @Value("${decoded.africastalking.api-key}")
    private String africasTalkingApiKey;

    @Value("${decoded.africastalking.default-from}")
    private String africasTalkingDefaultFrom;

    @Autowired
    private RedisMessagePublisher redisMessagePublisher;
```

```
    /**
     *
     * @param messages
     * @throws JsonProcessingException
     * @throws JsonMappingException
     */
    public void sendMessages(MessageRedisDto message) throws JsonMappingException,
JsonProcessingException {
        // Initialize AT SDK
        AfricasTalking.initialize(africasTalkingUserName, africasTalkingApiKey);

        SmsService sms = AfricasTalking.getService(AfricasTalking.SERVICE_SMS);
        try {
            if (message.getType() == MessageType.SMS) {
                List<Recipient> response = sms.send(message.getContent(),
africasTalkingDefaultFrom,
                        new String[] { "+" + message.getRecepient() }, true);
                if (!response.isEmpty()) {
                    message.setStatus(MessageStatus.SENT);
                }
            }
            // HANDLERS FOR OTHER 3rd PARTIES
        } catch (IOException e) {
            Integer retries = message.getRetries() + 1;
            message.setRetries(retries);
        }
        ObjectMapper mapper = new ObjectMapper();
        {your-packaging}.Message ms =
mapper.readValue(mapper.writeValueAsString(message),
        {your-packaging}.Message.class);
        messageRepository.save(ms);
    }
}
```

Define the Subscriber RedisMessageSubscriber implements the Redis-provided MessageListener interface

```
import {your-packaging}.MessageRedisDto;
import com.fasterxml.jackson.core.JsonProcessingException;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;

public class RedisMessageSubscriber {

    @Autowired
    private RedisTemplate<?, ?> redisTemplate;

    public void onMessage(String message) throws JsonProcessingException {
        MessageRedisDto ms = (MessageRedisDto)
this.redisTemplate.getValueSerializer().deserialize(message.getBytes());
```

```
        System.out.println(ms);
    }

}
```

Define the Publisher

We'll define a custom ***MessagePublisher*** interface and a ***RedisMessagePublisher*** implementation for it.

```
public interface MessagePublisher {
    void publish(final Object message);
}
```

```
import {your-packaging}.MessagePublisher; // your custom MessagePublisher
interface

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.listener.ChannelTopic;
import org.springframework.stereotype.Service;

@Service
public class RedisMessagePublisher implements MessagePublisher {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    @Autowired
    private ChannelTopic topic;

    public RedisMessagePublisher() {
    }

    public RedisMessagePublisher(
      RedisTemplate<String, Object> redisTemplate, ChannelTopic topic) {
      this.redisTemplate = redisTemplate;
      this.topic = topic;
    }

    public void publish(Object message) {
        redisTemplate.convertAndSend(topic.getTopic(), message);
    }

}
```

MySQL