RabbitMQ Notes

 Durable exchange and queues (subscription|...)

 Manual message acknowledgement

 Auto NACK if client disconnects

 Safe Flux streaming using FluxSink.OverflowStrategy.BUFFER to handle backpressure safely.

 Structured, extensible code for future headers or dynamic queues

* Retry with TTL & DLQ strategy?

**Best Practices Applied**

| **Aspect** | **Description** |
| --- | --- |
| **Manual ACK/NACK** | Ensures message is never lost due to premature acknowledgment. |
| **Disconnection-safe** | Uses emitter.isCancelled() to detect broken client connections. |
| **Thread-safe requeue** | Applies basicNack to safely requeue unprocessed messages. |
| **Encapsulated routing key** | Only takes queueSuffix, builds full `subscription |
| **Backpressure-safe** | Uses FluxSink.OverflowStrategy.BUFFER to handle slow subscribers. |

**❗️Issues & Red Flags**

| **Area** | **Issue** | **Fix** |
| --- | --- | --- |
| ❌ **Containers Map (Map<String, AbstractMessageListenerContainer>)** | Possible memory leak if containers aren't removed properly or are recreated unnecessarily. | Use computeIfAbsent + .stop() and cleanup more tightly. |
| ❌ **Queue non-durable (new Queue(name, false, ...))** | Risk of message loss if service restarts. | Use Queue(name, true, ...). |
| ❌ **Mixing Flux types (FluxSink<T> downcasted to FluxSink<NotificationResponse>)** | This is unsafe and brittle. | Use FluxSink<NotificationResponse> properly via typed emitter. |
| ⚠ **Multiple MessageListenerContainers for same queue** | Can lead to thread waste or duplicate delivery processing (if not careful). | Prefer a single shared consumer model (Flux.publish().refCount()). |
| ❌ **No centralized backpressure or replay** | Missed message delivery if client joins late. | Use ReplayProcessor or .replay(1) to retain last message. |
| ❌ **No message acknowledgment mode** | Default mode might be auto-ack (dangerous in long-polling). | Use AcknowledgeMode.MANUAL. |

**✅ Best Pattern to Use in QueueService**

Let's now implement your **production-ready QueueService**, based on these principles:

* Dynamic shared listener container per queue
* Manual ack
* One shared Flux<Notification> per queue using Flux.create() + FluxSink
* Cached per deviceId (or queue suffix)
* Clean lifecycle with stop on unsubscribe
* Safe handling of 10k concurrent listeners

**✅ Final Refactored QueueService**

java

CopyEdit

package com.example.service;

import com.example.config.RabbitMQConfig;

import com.rabbitmq.client.Channel;

import org.springframework.amqp.core.AcknowledgeMode;

import org.springframework.amqp.core.Queue;

import org.springframework.amqp.rabbit.connection.ConnectionFactory;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;

import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;

import org.springframework.stereotype.Service;

import reactor.core.publisher.Flux;

import reactor.core.publisher.FluxSink;

import java.nio.charset.StandardCharsets;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

@Service

public class QueueService {

private final RabbitTemplate rabbitTemplate;

private final ConnectionFactory connectionFactory;

private final Map<String, Flux<String>> activeFluxMap = new ConcurrentHashMap<>();

private final Map<String, SimpleMessageListenerContainer> listenerContainers = new ConcurrentHashMap<>();

public QueueService(RabbitTemplate rabbitTemplate, ConnectionFactory connectionFactory) {

this.rabbitTemplate = rabbitTemplate;

this.connectionFactory = connectionFactory;

this.rabbitTemplate.setMessageConverter(new Jackson2JsonMessageConverter());

}

/\*\*

\* Sends message to routing key based on dynamic queue name.

\*/

public void sendToQueue(String queueSuffix, String payload) {

String routingKey = "subscription|" + queueSuffix;

rabbitTemplate.convertAndSend(RabbitMQConfig.EXCHANGE, routingKey, payload);

}

/\*\*

\* Returns a shared reactive stream for a queue suffix.

\* Only one RabbitMQ consumer per suffix is created and shared among subscribers.

\*/

public Flux<String> getPushEvent(String queueSuffix) {

return activeFluxMap.computeIfAbsent(queueSuffix, this::createPushFlux);

}

private Flux<String> createPushFlux(String queueSuffix) {

String fullQueueName = "subscription|" + queueSuffix;

return Flux.<String>create(emitter -> {

SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();

container.setConnectionFactory(connectionFactory);

container.setQueueNames(fullQueueName);

container.setAcknowledgeMode(AcknowledgeMode.MANUAL); // ensure ack happens only after delivery

container.setMessageListener((message, channel) -> {

try {

String body = new String(message.getBody(), StandardCharsets.UTF\_8);

emitter.next(body);

if (!emitter.isCancelled()) {

channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);

} else {

channel.basicNack(message.getMessageProperties().getDeliveryTag(), false, true);

}

} catch (Exception e) {

channel.basicNack(message.getMessageProperties().getDeliveryTag(), false, true);

}

});

// Stop container when last subscriber unsubscribes

emitter.onDispose(() -> {

container.stop();

listenerContainers.remove(queueSuffix);

activeFluxMap.remove(queueSuffix);

});

listenerContainers.put(queueSuffix, container);

container.start();

}).publish().refCount(); // share the stream with multiple subscribers

}

}

package com.example.service;

import com.example.config.RabbitMQConfig;

import com.rabbitmq.client.Channel;

import org.springframework.amqp.core.AcknowledgeMode;

import org.springframework.amqp.rabbit.connection.ConnectionFactory;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;

import org.springframework.stereotype.Service;

import reactor.core.publisher.Flux;

import reactor.core.publisher.FluxSink;

import java.nio.charset.StandardCharsets;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

@Service

public class QueueService {

private final RabbitTemplate rabbitTemplate;

private final ConnectionFactory connectionFactory;

// Cache Flux and listener containers per queue suffix

private final Map<String, Flux<String>> activeFluxMap = new ConcurrentHashMap<>();

private final Map<String, SimpleMessageListenerContainer> listenerContainers = new ConcurrentHashMap<>();

public QueueService(RabbitTemplate rabbitTemplate, ConnectionFactory connectionFactory) {

this.rabbitTemplate = rabbitTemplate;

this.connectionFactory = connectionFactory;

}

/\*\*

\* Sends message to durable queue using routing key.

\*/

public void sendToQueue(String queueSuffix, String payload) {

String routingKey = "subscription|" + queueSuffix;

rabbitTemplate.convertAndSend(RabbitMQConfig.EXCHANGE, routingKey, payload);

}

/\*\*

\* Returns shared Flux for a given queue suffix.

\* If already created, reuses existing listener + Flux.

\*/

public Flux<String> getPushEvent(String queueSuffix) {

return activeFluxMap.computeIfAbsent(queueSuffix, this::createSharedFluxForQueue);

}

/\*\*

\* Creates one shared listener container and stream per suffix.

\* Handles ack/nack, onDispose, and safe reconnect logic.

\*/

private Flux<String> createSharedFluxForQueue(String queueSuffix) {

String queueName = "subscription|" + queueSuffix;

return Flux.<String>create(emitter -> {

SimpleMessageListenerContainer container;

// Reuse existing container or create new

container = listenerContainers.computeIfAbsent(queueSuffix, key -> {

SimpleMessageListenerContainer newContainer = new SimpleMessageListenerContainer();

newContainer.setConnectionFactory(connectionFactory);

newContainer.setQueueNames(queueName);

newContainer.setAcknowledgeMode(AcknowledgeMode.MANUAL);

return newContainer;

});

container.setMessageListener((message, channel) -> {

long deliveryTag = message.getMessageProperties().getDeliveryTag();

try {

String body = new String(message.getBody(), StandardCharsets.UTF\_8);

if (emitter.isCancelled()) {

channel.basicNack(deliveryTag, false, true);

return;

}

emitter.next(body);

channel.basicAck(deliveryTag, false);

} catch (Exception e) {

channel.basicNack(deliveryTag, false, true);

}

});

container.start();

// On cancel, stop container if no other subscribers

emitter.onDispose(() -> {

container.stop();

listenerContainers.remove(queueSuffix);

activeFluxMap.remove(queueSuffix);

});

}).publish().refCount(); // Shared stream that auto-starts container on demand

}

}

**Requirements Recap**

* You consume from RabbitMQ queue (e.g., subscription|ack|device123)
* Instead of returning **one message per long-poll response**, collect up to:
  + max 10 messages or
  + within 3 seconds
* Send those as a **single batched response** to the client
* Maintain **manual ACK**, durability, and long-poll behavior

public Flux<List<String>> getAckEvents(String queueSuffix) {

String queueName = "subscription|" + queueSuffix;

return Flux.<String>create(emitter -> {

SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();

container.setConnectionFactory(connectionFactory);

container.setQueueNames(queueName);

container.setAcknowledgeMode(AcknowledgeMode.MANUAL);

container.setMessageListener((message, channel) -> {

long deliveryTag = message.getMessageProperties().getDeliveryTag();

try {

String body = new String(message.getBody(), StandardCharsets.UTF\_8);

if (emitter.isCancelled()) {

channel.basicNack(deliveryTag, false, true);

return;

}

emitter.next(body); // buffer later

channel.basicAck(deliveryTag, false);

} catch (Exception e) {

channel.basicNack(deliveryTag, false, true);

}

});

emitter.onDispose(() -> container.stop());

container.start();

})

// 🧠 Batching logic:

.bufferTimeout(10, Duration.ofSeconds(3)) // ⏳ collect 10 items OR after 3 sec

.filter(batch -> !batch.isEmpty()) // 🔍 avoid sending empty batches

.publish()

.refCount();

}

**Usage in Netty (or REST, or Reactive Controller)**

If you're using Netty/HTTP long polling:

java

CopyEdit

getAckEvents("ack|device123|domain")

.next() // wait for one batch (1-10 messages)

.subscribe(batch -> {

// Convert to JSON or any response format

ctx.writeAndFlush(batch);

});

**Why This Works**

| **Behavior** | **Reason** |
| --- | --- |
| .bufferTimeout(n, d) | Collects items into a list (batch) of up to n, or after d duration |
| .publish().refCount() | Makes the Flux shareable and auto-start on first subscriber |
| Manual ack | Ensures messages are not lost if emitter dies |
| Low CPU & network | Long polling now responds in fewer requests |

**Optional Enhancements**

* Add .limitRate() if needed for backpressure
* Add deviceId metadata in each message if batching across multiple queues
* Allow batch size and timeout as configurable via properties/env

❌ **Your current design supports only one active subscriber per subscription ID. It does not fan out to all waiting long-polling clients.**

**Optional: Use Flux.publish().refCount() for Fan-Out**

You're essentially creating **one consumer per queue**, and **each queue is device-specific** — so:

**This is *point-to-point communication*, not fan-out.**

It’s designed for:

* Per-device message delivery (isolated),
* No multiple consumers per message,
* Highly controlled.

**2. Improved QueueService Using publish().refCount() — Shared Stream Fan-Out**

**Code:**

java

CopyEdit

return Flux.<String>create(emitter -> {

// create RabbitMQ listener

}).publish().refCount();

**What this does:**

* Creates a **shared Flux stream** from RabbitMQ for a **specific queue**.
* Any subscriber to this Flux will **receive the same data stream.**
* RabbitMQ messages are only consumed once, but **streamed to all live subscribers.**

**Use Case:**

* **Multiple listeners** connected to same queue
* Want **same data pushed to all of them**
* Works great for:
  + Broadcasting
  + Chat rooms
  + Newsfeeds
  + Device groups

**Which is better?**

| **Scenario** | **Use** |
| --- | --- |
| Device-level push | FCMCentralQueueService ✅ |
| Real-time group updates | publish().refCount() model ✅ |
| Multiple UI tabs per user | publish().refCount() recommended |
| Chat with multiple viewers | Shared Flux with fan-out ✅ |

**✅ Final Verdict for Your Code**

✅ You're doing **per-device push**, so FCMCentralQueueService is correctly designed.

You don't need publish().refCount() *unless* multiple clients need the **same message at the same time.**

**Summary: Per-Device Queues**

| **Design Element** | **Value** |
| --- | --- |
| Queue name per device | ✅ Yes |
| Routing key per device | ✅ Yes |
| Isolated delivery | ✅ Yes (only that device gets its message) |
| RabbitMQ fan-out exchange used? | ❌ No — you're using **direct exchange** per device |
| Shared queue between devices? | ❌ No — no overlap, hence no application-level fan-out needed |

**Benefits of This Design**

* 🧱 Strong **message isolation** between devices
* 🔐 Easy to **secure** and track delivery per device
* 🛡️ Easy to guarantee **exactly-once delivery** via manual ACK
* 🔁 Messages persist per queue if client is temporarily disconnected

**The Case:**

One request is doing **long-polling and listening** to queue:  
e.g., queue subscription|ack|00002|domainA

Meanwhile, **another request unsubscribes**, and your code:

* **unbinds the queue from the exchange**
* maybe even deletes the queue if it’s exclusive or marked for cleanup

| **Condition** | **Effect** |
| --- | --- |
| **If you only unbind** the queue from the exchange | 🔄 The queue still exists, but **no new messages will arrive**, because it’s no longer bound to any routing key. Existing messages (if any) remain. The consumer stays alive but becomes idle. |
| **If you delete the queue** | 🛑 The queue is destroyed. RabbitMQ sends a **Basic.Cancel** to the consumer. Spring’s listener container will detect this and **stop the consumer** automatically. |
| The Flux subscriber on client side | ⚠️ If no message is emitted and no timeout is set, the client may wait indefinitely (or until the request times out). |
| FluxSink.onNext(...) | Will **not be called again**, because queue is gone. |
| No error is thrown by default | Unless you catch it inside listener or Flux. |

**How to Handle Gracefully**

1. Add a check before unbinding:

Before unbinding a queue, make sure:

* No active listeners are attached (container.isRunning() or subscriber count)
* Or give subscribers a **"shutdown" message**

1. **Improve onDispose() in Flux:**
2. emitter.onDispose(() -> {
3. // Also stop container explicitly
4. container.stop();
5. });

**Add heartbeat or timeout fallback:**

Flux.create(...)

.timeout(Duration.ofSeconds(30))

.onErrorResume(TimeoutException.class, e -> Flux.empty());

**Send shutdown message before unbinding:**

Before you unbind:

rabbitTemplate.convertAndSend("exchange", "subscription|ack|00002|domainA", "\_\_UNSUBSCRIBE\_\_");

Then in listener:

if ("\_\_UNSUBSCRIBE\_\_".equals(body)) {

emitter.complete(); // Ends Flux

}

**Included Enhancements:**

| **Feature** | **Description** |
| --- | --- |
| 🧵 **Per-device queues** | Each device uses a unique `subscription |
| 🔁 **Batched delivery** | getAckEvents(...) returns up to 10 acks in one batch |
| ♻️ **Listener container reuse** | containerMap used to manage and reuse consumers |
| 🧠 **Replay buffer** | replayLastEvents(...) returns last 10 messages per queue |
| 🕒 **TTL queues** | Queues expire after 5 minutes of inactivity via x-expires |
| ⏹ **Stop consumer** | stopConsumer(queueSuffix) stops a listener and removes it |
| ⏸ **Pause consumer** | pauseListener(queueSuffix) pauses a listener without removing |
| 📡 **Monitoring** | getActiveListeners() returns a list of live queue listeners |

package com.example.pushserver.queue;

import com.fasterxml.jackson.databind.ObjectMapper;

import lombok.extern.slf4j.Slf4j;

import org.springframework.amqp.core.\*;

import org.springframework.amqp.rabbit.connection.ConnectionFactory;

import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;

import org.springframework.amqp.support.AmqpHeaders;

import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;

import org.springframework.messaging.handler.annotation.Header;

import org.springframework.stereotype.Service;

import reactor.core.publisher.Flux;

import reactor.core.publisher.FluxSink;

import reactor.core.scheduler.Schedulers;

import java.nio.charset.StandardCharsets;

import java.time.Duration;

import java.util.\*;

import java.util.concurrent.ConcurrentHashMap;

@Slf4j

@Service

public class QueueService {

private final AmqpAdmin amqpAdmin;

private final ConnectionFactory connectionFactory;

private final ObjectMapper objectMapper = new ObjectMapper();

private final Exchange exchange;

// Active containers per queue

private final Map<String, SimpleMessageListenerContainer> containerMap = new ConcurrentHashMap<>();

// TTL tracking

private final Map<String, Long> subscriptionTimestamps = new ConcurrentHashMap<>();

// Replay cache

private final Map<String, List<String>> lastEvents = new ConcurrentHashMap<>();

public QueueService(AmqpAdmin amqpAdmin, ConnectionFactory connectionFactory) {

this.amqpAdmin = amqpAdmin;

this.connectionFactory = connectionFactory;

this.exchange = ExchangeBuilder.directExchange("push-events").durable(true).build();

amqpAdmin.declareExchange(this.exchange);

}

public Flux<List<String>> getAckEvents(String queueSuffix) {

String queueName = getQueueName(queueSuffix);

declareQueue(queueName, queueSuffix);

return Flux.<String>create(emitter -> {

SimpleMessageListenerContainer container = containerMap.computeIfAbsent(queueName, key -> {

SimpleMessageListenerContainer c = new SimpleMessageListenerContainer();

c.setConnectionFactory(connectionFactory);

c.setQueueNames(queueName);

c.setAcknowledgeMode(AcknowledgeMode.MANUAL);

c.setMessageListener(message -> {

long tag = message.getMessageProperties().getDeliveryTag();

try {

String payload = new String(message.getBody(), StandardCharsets.UTF\_8);

emitter.next(payload);

// Cache for replay (bounded to 10)

lastEvents.computeIfAbsent(queueName, k -> new LinkedList<>());

List<String> cached = lastEvents.get(queueName);

if (cached.size() >= 10) cached.remove(0);

cached.add(payload);

message.getMessageProperties().getChannel().basicAck(tag, false);

} catch (Exception e) {

log.error("Ack processing failed", e);

message.getMessageProperties().getChannel().basicNack(tag, false, true);

}

});

c.start();

return c;

});

subscriptionTimestamps.put(queueName, System.currentTimeMillis());

emitter.onDispose(() -> {

log.info("Emitter disposed for queue: {}", queueName);

});

})

.bufferTimeout(10, Duration.ofSeconds(3))

.filter(batch -> !batch.isEmpty())

.publish()

.refCount()

.doOnCancel(() -> log.info("Flux cancelled for queue: {}", queueName))

.subscribeOn(Schedulers.boundedElastic());

}

public void stopConsumer(String queueSuffix) {

String queueName = getQueueName(queueSuffix);

Optional.ofNullable(containerMap.remove(queueName)).ifPresent(container -> {

container.stop();

log.info("Stopped consumer for: {}", queueName);

});

}

public void pauseListener(String queueSuffix) {

String queueName = getQueueName(queueSuffix);

Optional.ofNullable(containerMap.get(queueName)).ifPresent(container -> {

container.stop();

log.info("Paused listener for: {}", queueName);

});

}

public List<String> getActiveListeners() {

return new ArrayList<>(containerMap.keySet());

}

public List<String> replayLastEvents(String queueSuffix) {

String queueName = getQueueName(queueSuffix);

return lastEvents.getOrDefault(queueName, Collections.emptyList());

}

private void declareQueue(String queueName, String routingKey) {

Queue queue = QueueBuilder.durable(queueName).withArgument("x-expires", 300000).build(); // TTL 5 mins

amqpAdmin.declareQueue(queue);

Binding binding = BindingBuilder.bind(queue).to(exchange).with(routingKey).noargs();

amqpAdmin.declareBinding(binding);

}

private String getQueueName(String suffix) {

return "subscription|" + suffix;

}

}

**Your Project Code: listeners.put(subscriptionId, eventEmitter::emit)**

**How it works:**

* You maintain a **ConcurrentHashMap<String, Consumer<T>>** called listeners
* Each subscriptionId is tied to one eventEmitter
* On message arrival, you do:

java

CopyEdit

listeners.forEach((key, listener) -> listener.accept(message));

* The listener pushes the message to the client via the sink

**✅ Pros:**

| **Point** | **Why it's good** |
| --- | --- |
| 🔄 Dynamic | You can add/remove subscribers at runtime easily |
| 👤 Per-subscription | One emitter per client/device = clean isolation |
| 🔧 Simple mapping | Easy to understand and trace per-client state |

**❌ Cons:**

| **Point** | **Limitation** |
| --- | --- |
| ❌ Manual fan-out | You must broadcast to all relevant emitters yourself |
| ❌ Not reactive-native | You bypass backpressure — pushing to sinks directly |
| ❌ Memory risk | If listeners map grows large or isn't cleaned up, may cause leaks |
| ❌ One emitter per subscription | No real shared stream, no reuse, no hot observable semantics |

**🔁 QueueService Code: Using Flux.create().publish().refCount()**

**How it works:**

* RabbitMQ listener wraps a FluxSink<T>
* Flux.create(...) is turned into a **shared stream** using .publish().refCount()
* If multiple subscribers connect, they all share the **same message stream**
* You can buffer, transform, replay, etc., using the Flux pipeline

**✅ Pros:**

| **Point** | **Why it's better** |
| --- | --- |
| ✅ Reactive-compliant | Fully supports backpressure, cancellation, timeouts |
| ✅ Shared stream | If needed, multiple subscribers can share the same stream |
| ✅ Batching built-in | You can buffer messages using .bufferTimeout() |
| ✅ Replay support | Add .replay(...) if needed per stream |
| ✅ Cleaner lifecycle | You can cleanly dispose via onDispose() |

**❌ Cons:**

| **Point** | **Limitation** |
| --- | --- |
| 🔄 Less manual control | You don’t have a manual map of listeners per se |
| 🧠 Slightly more complex | You need to understand how hot vs cold publishers behave |
| ❓ No built-in mapping to subscriptionId | You handle routing via queue names instead |

**🧠 Which Is Best? (Verdict)**

| **Scenario** | **Recommended Approach** |
| --- | --- |
| You need per-device isolation | ✅ Your emitter-per-subscription is simple and fine |
| You want buffering, retry, replay, timeout, etc. | ✅ QueueService with Flux.create().publish().refCount() is better |
| You need multiple subscribers to the same data (e.g. dashboard tabs) | ✅ QueueService is best |
| You want more control and simpler debugging | ✅ Your emitter-map approach is more transparent |
| You care about memory cleanup, scalability, and reactive control | ✅ QueueService wins here |

**✅ My Suggestion (Hybrid)**

If you're happy with per-device emitters but want more power:

* Use QueueService’s Flux logic
* But add your Map<String, Flux<...>> for per-subscription management
* Avoid listeners.put(...) and instead store Flux objects

java

CopyEdit

private final Map<String, Flux<List<String>>> subscriptionStreams = new ConcurrentHashMap<>();

Let Spring AMQP + Reactor handle:

* Backpressure
* Stream closure
* Acknowledgement
* Buffering

**📌 Summary**

| **Feature** | **Your Code** | **QueueService** |
| --- | --- | --- |
| Per-device mapping | ✅ Yes | ✅ Yes (via queue name) |
| Reactive lifecycle | ❌ Manual | ✅ Built-in |
| Stream reuse | ❌ One sink per | ✅ Shareable |
| Message batching | ❌ Manual | ✅ Easy with .bufferTimeout() |
| Replay buffer | ❌ You build it | ✅ Already integrated |
| Best for | Simplicity, isolation | Scaling, control, reuse |

Now full code with all features –

package com.example.pushserver.queue;

import com.fasterxml.jackson.databind.ObjectMapper;

import lombok.extern.slf4j.Slf4j;

import org.springframework.amqp.core.\*;

import org.springframework.amqp.rabbit.connection.ConnectionFactory;

import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;

import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;

import org.springframework.stereotype.Service;

import reactor.core.publisher.Flux;

import reactor.core.publisher.FluxSink;

import reactor.core.scheduler.Schedulers;

import java.nio.charset.StandardCharsets;

import java.time.Duration;

import java.util.\*;

import java.util.concurrent.ConcurrentHashMap;

@Slf4j

@Service

public class HybridQueueService {

private final AmqpAdmin amqpAdmin;

private final ConnectionFactory connectionFactory;

private final ObjectMapper objectMapper = new ObjectMapper();

private final Exchange exchange;

// Cache of active listener containers per queue

private final Map<String, SimpleMessageListenerContainer> containerMap = new ConcurrentHashMap<>();

// Per-subscription Flux stream cache

private final Map<String, Flux<List<String>>> streamMap = new ConcurrentHashMap<>();

// Replay buffer for reconnect scenarios

private final Map<String, List<String>> replayBuffer = new ConcurrentHashMap<>();

public HybridQueueService(AmqpAdmin amqpAdmin, ConnectionFactory connectionFactory) {

this.amqpAdmin = amqpAdmin;

this.connectionFactory = connectionFactory;

this.exchange = ExchangeBuilder.directExchange("push-events").durable(true).build();

amqpAdmin.declareExchange(exchange);

}

/\*\*

\* Returns a Flux stream of batched messages from a device-specific queue.

\* Caches the Flux so reconnects resume from the same stream.

\*/

public Flux<List<String>> getAckEvents(String queueSuffix) {

return streamMap.computeIfAbsent(queueSuffix, suffix -> {

String queueName = getQueueName(suffix);

declareQueue(queueName, suffix);

return Flux.<String>create(emitter -> {

SimpleMessageListenerContainer container = containerMap.computeIfAbsent(queueName, key -> {

SimpleMessageListenerContainer c = new SimpleMessageListenerContainer();

c.setConnectionFactory(connectionFactory);

c.setQueueNames(queueName);

c.setAcknowledgeMode(AcknowledgeMode.MANUAL);

c.setMessageListener(message -> {

long tag = message.getMessageProperties().getDeliveryTag();

try {

String payload = new String(message.getBody(), StandardCharsets.UTF\_8);

emitter.next(payload);

// Save last 10 events for replay

replayBuffer.computeIfAbsent(queueName, k -> new LinkedList<>());

List<String> cache = replayBuffer.get(queueName);

if (cache.size() >= 10) cache.remove(0);

cache.add(payload);

message.getMessageProperties().getChannel().basicAck(tag, false);

} catch (Exception e) {

log.error("Error processing message", e);

message.getMessageProperties().getChannel().basicNack(tag, false, true);

}

});

c.start();

return c;

});

emitter.onDispose(() -> log.info("Emitter disposed for: {}", queueName));

})

.bufferTimeout(10, Duration.ofSeconds(2))

.filter(batch -> !batch.isEmpty())

.publish()

.refCount()

.subscribeOn(Schedulers.boundedElastic());

});

}

/\*\*

\* Stops and removes the listener for a specific device queue.

\*/

public void stopConsumer(String queueSuffix) {

String queueName = getQueueName(queueSuffix);

Optional.ofNullable(containerMap.remove(queueName)).ifPresent(container -> {

container.stop();

log.info("Stopped container for: {}", queueName);

});

streamMap.remove(queueSuffix);

}

/\*\*

\* Pauses the listener but keeps it in the cache for possible resume.

\*/

public void pauseListener(String queueSuffix) {

String queueName = getQueueName(queueSuffix);

Optional.ofNullable(containerMap.get(queueName)).ifPresent(container -> {

container.stop();

log.info("Paused listener for: {}", queueName);

});

}

/\*\*

\* Returns active queue names currently being listened to.

\*/

public List<String> getActiveListeners() {

return new ArrayList<>(containerMap.keySet());

}

/\*\*

\* Returns last 10 messages received from a given queue.

\*/

public List<String> replayLastEvents(String queueSuffix) {

String queueName = getQueueName(queueSuffix);

return replayBuffer.getOrDefault(queueName, Collections.emptyList());

}

/\*\*

\* Declares a queue with a 5-minute idle TTL and binds it to the exchange.

\*/

private void declareQueue(String queueName, String routingKey) {

Queue queue = QueueBuilder.durable(queueName).withArgument("x-expires", 300000).build();

amqpAdmin.declareQueue(queue);

Binding binding = BindingBuilder.bind(queue).to(exchange).with(routingKey).noargs();

amqpAdmin.declareBinding(binding);

}

private String getQueueName(String suffix) {

return "subscription|" + suffix;

}

}

**Included Enhancements:**

| **Feature** | **Description** |
| --- | --- |
| 🧵 **Per-device queues** | Each device uses a unique `subscription |
| 🔁 **Batched delivery** | getAckEvents(...) returns up to 10 acks in one batch |
| ♻️ **Listener container reuse** | containerMap used to manage and reuse consumers |
| 🧠 **Replay buffer** | replayLastEvents(...) returns last 10 messages per queue |
| 🕒 **TTL queues** | Queues expire after 5 minutes of inactivity via x-expires |
| ⏹ **Stop consumer** | stopConsumer(queueSuffix) stops a listener and removes it |
| ⏸ **Pause consumer** | pauseListener(queueSuffix) pauses a listener without removing |
| 📡 **Monitoring** | getActiveListeners() returns a list of live queue listeners |

Would you like to:

* Add support for resuming paused listeners?
* Integrate WebSocket/HTTP handler on top of this?
* Auto-cleanup replay cache after queue TTL expiry?

Let me know and I’ll help build the next layer!

**1. Replay Buffer (replayBuffer)**

**✅ Purpose:**

To **cache** the last *N messages* (typically 10) **per device queue**, so that when a client **reconnects** or **misses a message**, it can still retrieve the most recent messages.

**🔧 Implementation:**

replayBuffer.computeIfAbsent(queueName, k -> new LinkedList<>());

List<String> cache = replayBuffer.get(queueName);

if (cache.size() >= 10) cache.remove(0);

cache.add(payload);

**📦 When to Use:**

* When a **device reconnects** after a network drop and wants the last known messages
* If a message was delivered but **client acknowledgment failed**
* To support **eventual consistency** between producer and consumer state

**2. TTL Queues (Queue Expiry)**

**✅ Purpose:**

To **automatically remove idle queues** after a set period of time (e.g., 5 minutes of inactivity), **saving resources** in RabbitMQ and avoiding stale connections.

**🔧 Implementation:**

java

CopyEdit

Queue queue = QueueBuilder.durable(queueName)

.withArgument("x-expires", 300000) // 5 minutes = 300000ms

.build();

**📦 When to Use:**

* When you want to **clean up queues** of disconnected devices automatically
* In **IoT or mobile** use cases where devices might connect irregularly
* Helps prevent **queue leaks** when devices crash or disconnect silently

**🧠 Notes:**

* This does **not delete the queue immediately** when it's inactive — only after the TTL.
* If a queue is accessed before TTL expiry, the timer **resets**.

**3. pauseListener(String queueSuffix)**

**✅ Purpose:**

To **temporarily stop** listening to a RabbitMQ queue **without destroying** the queue or stream mapping. This is like “muting” a subscriber.

**🔧 Implementation:**

java

CopyEdit

Optional.ofNullable(containerMap.get(queueName)).ifPresent(SimpleMessageListenerContainer::stop);

**📦 When to Use:**

* For **backpressure handling** (e.g., if server load is high)
* When device/client wants to **pause** receiving messages temporarily
* To **troubleshoot a specific queue** without affecting others

**🧠 Differences:**

* Unlike stopConsumer(), this method **does not remove** the container or stream from memory.
* You can **resume** later simply by calling getAckEvents(...) again.

**🔁 4. replayLastEvents(String queueSuffix)**

**✅ Purpose:**

To **return the last 10 events** stored in the replay buffer for a specific queue, e.g., on **reconnect**.

**🔧 Implementation:**

java

CopyEdit

return replayBuffer.getOrDefault(queueName, Collections.emptyList());

**📦 When to Use:**

* After a **connection failure**, the client polls this method to get recent unacknowledged events.
* Useful for implementing **idempotent** event handling or **deduplication**.
* Can also be used for **message recovery** in case of consumer failure.