Facultad

de

Ciencias

Implementación de un sistema interno de mensajería para EMI-SUITE, basado en el protocolo AMQP

Implementation of an inner messaging system for EMI-SUITE, based on the AMPQ protocol

Trabajo de fin de grado para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Carlos Silva de Arozarena

Directora: Marta Elena Zorrilla Pantaleón

Co-Director: Miguel Sierra Sánchez

06 – 2024

AGRADECIMIENTOS

Quisiera agradecer, en primer lugar, a la empresa Soincon el haberme concedido la oportunidad de realizar este Trabajo de Fin de Grado sobre su plataforma, además de los dos meses de prácticas que realicé durante los meses del verano anterior. La experiencia me ha permitido entender mejor el funcionamiento de una empresa tecnológica, y ha sido enriquecedora a todos los niveles.

También quiero darle las gracias a la directora de este proyecto, Marta Zorrilla, por su labor tanto como docente como de tutora académica del trabajo.

RESUMEN

Uno de los puntos más importantes en el desarrollo de plataformas informáticas es el método empleado en la transmisión de su información.

Advance Message Quering Protocol, (AMQP), es un protocolo de mensajería de código abierto, en el que los mensajes se distribuyen a distintas colas de acuerdo a sus claves asociadas, y el software RabbitMQ ofrece un servidor que emplea ese protocolo, permitiendo la comunicación entre distintas aplicaciones.

En este proyecto, se reconfigura parte del código fuente de la plataforma EMI Suite de la empresa Soincon, para que su sistema haga un aprovechamiento más óptimo de las funcionalidades avanzadas del protocolo, añadiendo políticas para el almacenamiento y posible reenvío de mensajes en caso de error. También se refactoriza el código de cara a mejorar la mantenibilidad del sistema.

PALABRAS CLAVE

Cola de mensajería, Advance Message Quering Protocol.

ABSTRACT

One of the most important aspects on the development of digital platforms is the method used for the transmission of its information.

Advance Message Quering Protocol, (AMQP), is an open-source messaging protocol, in which the messages get distributed to queues depending on their associated keys, and the RabbitMQ software offers a server that makes use of that protocol, allowing communication between different applications.

In this proyect, part of the source code of the EMI Suite platform from Soincon is reworked, so that its system makes a more optimal use of the advance functions of the protocol, adding management for the storage and possible resubmission of messages in case of an error. The code is also refactorized to improve the maintainability of the system.

KEYWORDS

Message queue, Advance Message Quering Protocol.

ÍNDICE

1. ANTECEDENTES

Soincon (Soluciones Industriales de Conectividad) es una empresa tecnológica radicada en Cantabria que se dedica a la digitalización de procesos industriales.

Previamente al inicio de la realización de este trabajo de fin de grado, se habían realizado dos meses de prácticas externas en Soincon.

La primera parte fue extracurricular, durante el mes de julio de 2023. Sirvió de introducción a la plataforma de la empresa, EMI Suite. Se estudió su funcionamiento general y las partes en las que está dividida, y se plantearon los primeros ejercicios de formación internas para practicar con los entornos de desarrollo que se utilizan para gestionarla, siendo Java junto con el framework Spring los más importantes.

A lo largo de agosto de 2023 tuvo lugar la parte curricular, continuando con actividades similares a las del mes anterior. Se profundiza en el módulo Visual Tracking de EMI Suite, proporcionándose su código fuente para llevarse a cabo ejercicios directamente a partir de éste, y familiarizarse más con su estructura interna.

2. OBJETO DEL PROYECTO

El objetivo del presente trabajo de fin de grado es mejorar el uso que la empresa Soincon hace del protocolo de mensajería Advance Message Queuing Protocol, conocido por el acrónimo AMQP, en su plataforma EMI SUITE 4.0, que actualmente solo integran en una reducida parte de sus módulos y a un nivel muy básico.

En concreto, este TFG pretende satisfacer los siguientes requisitos:

- **Mejorar la mantenibilidad y escalabilidad del código:** En el estado actual del código fuente de EMI SUITE, casi toda la configuración relacionada con AMQP se encuentra en una misma clase de configuración. Es necesario remodularizarlo en lo posible para evitar que esa clase crezca de manera desmedida conforme nuevos módulos del sistema van integrando el protocolo. Con la remodularización, esta clase podrá mantenerse en un tamaño manejable de alrededor de unas 350 líneas de código.

- **Añadir política de reintentos:** Actualmente, los datos de los mensajes enviados a través del protocolo AMQP en la aplicación intentan enviarse una sola vez, sin importar que el envío haya sido exitoso o no. Se debe mejorar la funcionalidad de manera que los mensajes enviados puedan almacenarse temporalmente y tratar de reenviarse una cantidad determinada de veces, personalizable por los administradores en cada módulo.

- **Almacenamiento de mensajes no enviados:** El sistema tampoco implementa ningún mecanismo con el que se guarden los mensajes que no hayan podido ser enviados correctamente, por lo que sus datos se pierden en caso de que haya problemas en la conexión. Se solicita añadir esta funcionalidad para que, si se ha superado el número de reintentos anteriormente definido, estos mensajes se almacenen indefinidamente para su posterior revisión manual.

- **Consumo de mensajería por un solo receptor:** En AMQP, los diferentes módulos del sistema se mantienen a la escucha de diferentes mensajes en base a claves de enrutamiento, que dejan de almacenarse en el sistema al ser consumidos. Se debe garantizar, en la lógica del protocolo AMQP implementada, que si un módulo del sistema necesita recibir mensajes con determinada clave, todos los mensajes que la contengan y hayan sido correctamente enviados puedan ser consumidos por él. En el código fuente del que se parte, esto no se logra dado que no se impide la posibilidad de que dos o más consumidores se encuentren compitiendo por mensajes con una misma clave. Si esto se da, solo uno de los consumidores recibe el mensaje, por lo que en cada envío al menos un consumidor está perdiendo información que, en teoría, debería estar recibiendo.

- **Mayor integración de AMQP en EMI SUITE:** Una vez alcanzados los objetivos anteriores, empezar a implementar una solución que siga el mismo esquema en otras clases del dominio.

3. ADVANCE MESSAGE QUERING PROTOCOL

Advance Message Quering Protocol (AMQP) es un protocolo de mensajería asíncrono de código abierto [1]. Los principales elementos que lo conforman son los siguientes:

- **Mensaje**: Es el elemento central de la comunicación. Puede representar cualquier tipo de objeto o clase definida en la aplicación, que podrá usarse para notificar a otros componentes de eventos producidos en el sistema que deben conocer para su correcto funcionamiento. Aparte de poseer un identificador único, en el encabezado se definen varias propiedades, en su mayoría opcionales. La única que se utilizará en este proyecto es la clave de enrutamiento, que se utilizará para determinar hacia dónde se enviará. Los mensajes se mandan a un determinado intercambio, que es el que se encarga de enrutarlos a sus distintos destinos, en base a las claves que los mensajes contienen.

- **Productor:** Se refiere a los emisores de los mensajes en el sistema. Cualquier clase de la aplicación que genere notificaciones AMQP está actuando como productora. Después de transmitir el mensaje, un productor puede recibir notificaciones de confirmación o de rechazo de su mensaje y actuar en consecuencia.

- **Receptor:** Se refiere a toda clase de la aplicación que recibe y reacciona a los mensajes recibidos por AMQP. Para esto, un receptor se mantiene a la escucha de una o varias colas de mensajería, suscribiéndose a estas. El receptor procesará su información, y podrá confirmar la recepción del mensaje o, en caso de que haya algún error con sus datos, rechazar el mismo.

- **Cola de mensajería:** Los mensajes se envían a las colas, y en caso de que no haya ningún consumidor esperando a recibirlos, se almacenan en su interior hasta que exista. Las colas se enlazan a un intercambio utilizando una o varias claves de enrutamiento, y en base de sus valores comparados con los de los mensajes que reciba, el intercambio determinará si debe mandarse una copia de la información o no a cada cola que tenga enlazada. Las colas pueden definirse como persistentes, (no se borrarán al reiniciarse el bróker en el que están definidas), o no, siendo todas persistentes en el caso de EMI Suite. Además, se les puede dar argumentos opcionales en su declaración para alterar su comportamiento. Los argumentos relevantes a este proyecto son:

    - **x-message-ttl**, que define en milisegundos la cantidad de tiempo tras el cuál un mensaje no consumido será retirado de la cola.

    - **x-single-active-consumer**, que limita a uno el número de consumidores simultáneos que pueden encontrarse escuchándola.

    - **dead-letter-exchange**, con el que se le puede indicar un nuevo intercambio al que enviar el mismo mensaje una vez caduque, o en caso de que fuera rechazado por el emisor. Por último, para alterar las características de una cola, es necesario eliminar y volverla a crear desde cero; si se intentan sobreescribir sus características declarando una nueva del mismo nombre con una configuración diferente, se lanzará una excepción.

- **Clave de enrutamiento**: Puede especificarse una clave al momento de publicar un mensaje, para que el intercambio al que se realice el envío lo enrute a diferentes colas de mensajería en función de su valor. En determinados casos, es posible utilizar caracteres comodín que representan valores especiales, como * (una sola palabra), o # (una cantidad indeterminada de palabras), útil si las claves en el sistema van a seguir algún tipo de jerarquía o patrón.

- **Intercambios:** Todos los mensajes se envían a un intercambio concreto, que lo distribuye a las colas que tenga enlanzadas según las claves de los elementos involucrados. Se dividen en tres tipos.

- **Directo (Direct)**. Distribuye los mensajes con una clave dada a las colas cuyas claves de enrutamiento en el intercambio coincidan exactamente con las del mensaje enviado.

- **Fanout**. Ignoran las claves y propagan los mensajes a toda cola que tengan enlazada.

- **Topic**. son parecidos a los directos, con la diferencia de que se pueden usar los caracteres comodín anteriormente mencionados.

La figura 1 muestra un esquema de la propagación de mensajes según el tipo de intercambio.

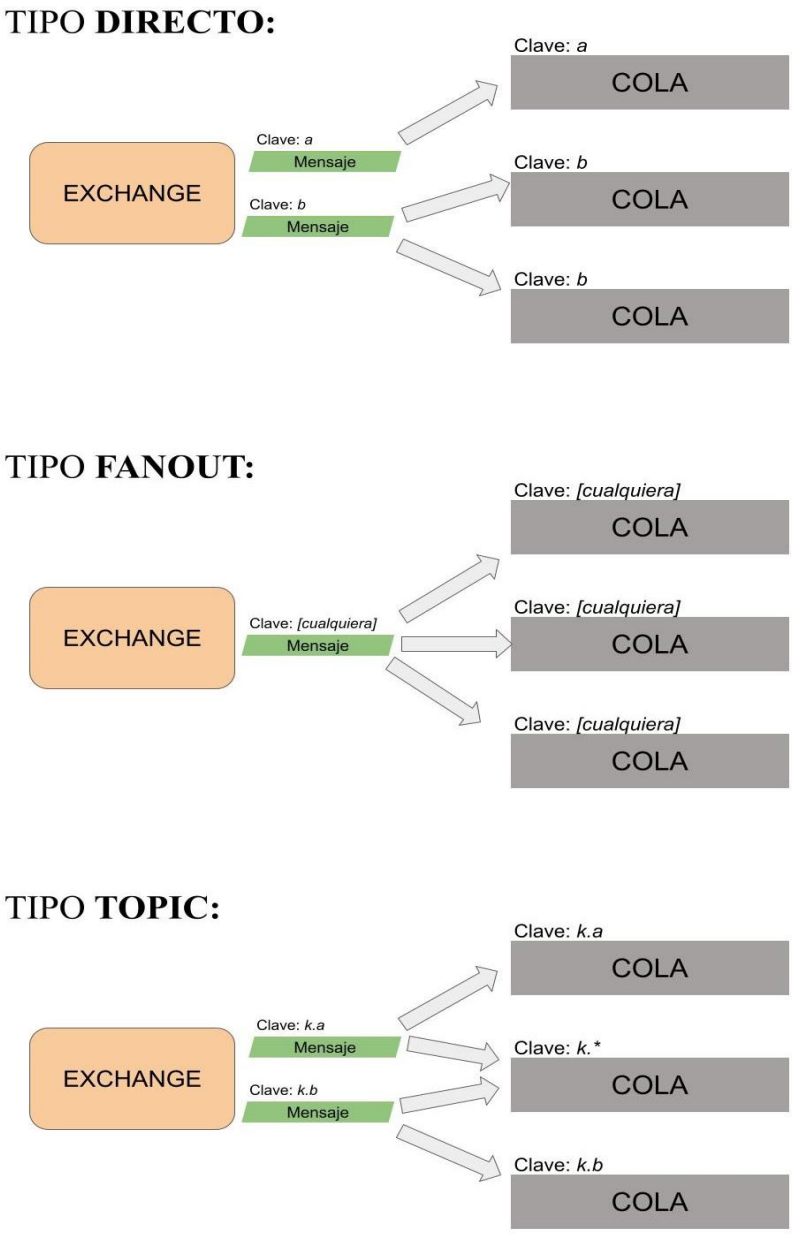


Figura 1: Esquema del funcionamiento de los tipos de intercambio

4. USO DE SPRING FRAMEWORK DE JAVA EN EL PROYECTO

El *framework* Spring es una herramienta de código abierto que facilita el desarrollo de aplicaciones empresariales en Java [2].

Su uso simplifica la programación, dado que permite automatizar la gestión de la inyección de dependencias, y conversiones de tipos de datos, como se verá en los siguientes apartados.

4.1. BEANS

La base del funcionamiento de Spring es un contenedor de objetos definidos en la aplicación llamados Beans. Estos objetos se generan en base a anotaciones a funciones en el código. Leyendo las anotaciones, el contenedor accede a los objetos Bean, pudiendo generar una instancia cuando uno de los objetos de la aplicación así lo requiera. A continuación, se muestra un ejemplo de esto:

```
@Bean
public Objeto objeto() {

    /* Se construye el objeto. */
    Objeto objeto = new Objeto();

    /* Se configuran sus características con los valores que se deseen. */
    objeto.setCaracteristica_1(valor1);
    objeto.setCaracteristica_2(valor2);

    /* Cada vez que se requiera un objeto de este tipo, se obtendrá
     * uno con las caracteristicas definidas en esta función. */
    return objeto;
}
```

Puede establecerse más de un Bean del mismo tipo, en cuyo caso se determinará cuál de los Beans debe inyectarse en cada caso con la anotación “Qualifier”. Esto se explicará más adelante.

Por defecto, la generación de Beans funciona de manera que solo puede existir una instancia de cada Bean definido en la aplicación. Es decir, si dos o más módulos están utilizando el mismo Bean, todas estarán haciendo uso de la misma instancia. Puede modificarse este comportamiento mediante la anotación “Scope” para que puedan generarse múltiples instancias del mismo Bean. Sin embargo, en el contexto de EMI Suite, esto no es útil e incluso resulta contraproducente, puesto que las colas de mensajería y otros elementos del protocolo AMQP están definidas a través de Beans, y no tendría sentido generar varias copias de colas o intercambios con el mismo nombre.

4.2. ANOTACIONES A NIVEL DE CLASE

Las clases en un proyecto Java Spring pueden estar anotadas de las siguientes maneras:

- **Entity.** La información de los objetos de las clases anotadas con “Entity” se guardará como una tabla en la base de datos de la aplicación. Ejemplo:

```
@Entity
/* Por defecto, la tabla tendra de nombre el mismo que la clase.
 * Puede establecerse otro con el parámetro name de la anotación @Table */
public class ClaseEntidad {

    /* El atributo id especifica la clave primaria de la tabla */
    @Id
    private Long id;

    /* Se definen otros campos de la tabla */
    private String parametro1;
    private String parametro2;

}
```

- **Repository.** Clases que se encargan de la gestión de los objetos guardados en la base de datos.

```
@Repository
/* Al extender el JpaRepository, ya se están definiendo sin necesidad de
 * añadir más código métodos básicos CRUD.
 * Se pueden extender otras interfaces que incluyan otras funciones. */
public interface ObjetoRepository extends JpaRepository<Objeto, Long> {

    /* En el cuerpo de la clase pueden añadirse métodos.
     * Siguiendo la estructura "findBy[nombre del atributo] pueden
     * crearse de manera automatizada metodos de busqueda en base
     * a un parámetro del objeto. */
    public Objeto findByCaracteristica1(String caracteristica1);


}
```

  Esta sería la manera más simple de definir un repositorio. Dentro de EMI Suite, primero se define una interfaz similar a la del código de ejemplo anterior, pero que no está anotada, y una clase sí anotada con “Repository” implementa dicha interfaz y extiende a la interfaz AbstractDaoWithFilter, que incluye funciones para filtrado en las búsquedas en la clase de datos. El cuerpo de las clases anotadas también incluye funciones con lógica específica necesaria para cada caso.

- **Configuration**: La clases que primero se procesan al desplegar la aplicación. Sirve como indicación de que en esta clase se establecerán Beans de los cuáles dependen el resto de componentes de la aplicación (aunque también pueden definirse Beans en clases no anotadas como “Configuration”)

- **PropertySource**: Anotación de conveniencia. Debe utilizarse en una clase anotada como “Configuration”. Indica la ruta a un fichero que contenga propiedades generales de la aplicación para poder leerlas desde esa clase. Ejemplo de uso:

```
@Configuration
@PropertySource( {"classpath:environment.properties"} )
public class RabbitConfig {

    private @Value ("${[nombre de atributo del fichero de propiedades]}")
        String parametro;
```

- **Service**: Indica que la clase anotada contiene la mayoría de la lógica de servicio de la aplicación. Como se procesa después de las clases anotadas como “Configuration”, puede tener dependencias hacia Beans definidos en ella. Normalmente, se llamará a métodos de las clases repositorio desde las clases Service.

- **Component:** Las clases anotadas como componentes son gestionadas automáticamente por Spring. Al ejecutar la aplicación, se ejecutará una instancia de esta clase y se gestionarán sus dependencias.

- **RestController**: Clases que definen los métodos HTTP utilizables por los usuarios de la aplicación. Normalmente, utilizará internamente los métodos definidos en las clases de servicio. Ejemplo de controlador:

```
@RestController
@RequestMapping(value = "/[Enlace base de los métodos de la clase]")
public class Controlador {

    @GetMapping(value = "/[Enlace al método concreto]")
    public Controlador metodoGetControlador() {
        /* Lógica de la función */
```

4.3. JACKSON CON SPRING FRAMEWORK

Json es un formato estandarizado para transmisión de mensajes en forma de texto basado en Java, cuyo uso está cada vez más extendido. Los mensajes de la plataforma EMI Suite se transmiten con este formato.

Dentro del *framework* Spring, existe la clase MappingJackson2MessageConverter, que permite la conversión de objetos Java en mensajes de tipo Json. Esta clase se utilizará durante la transmisión de mensajes a través del protocolo AMQP.

5. EMI SUITE 4.0

EMI SUITE es el producto base sobre el que Soincon desarrolla las soluciones software de los procesos productivos de una organización, personalizadas según las necesidades de cada cliente. La plataforma se divide en ocho módulos, a través de los cuáles se lleva a cabo el control de diferentes partes del proceso industrial, como la gestión del personal o de los materiales industriales. Una vez instalada la aplicación, el usuario puede acceder a ella a través

de una URL única que se le proporciona durante la instalación, junto con el nombre de usuario y contraseña de su perfil. Se le habilitarán aquellos módulos que sean relevantes para su negocio. La comunicación entre módulos previo a la implementación de amqp se realiza mediantes sockets.

Las modificaciones en la información deben notificarse correctamente entre todos los componentes de un mismo módulo y a través de ellos, dado que un cambio en los datos es susceptible de afectar a otro. Por ejemplo, si una máquina se elimina de la base de datos por dejar de estar operativa, todas las aplicaciones deben saberlo al momento para evitar que desde otro componente de sistema se le asigne otro trabajo a esa máquina ahora inutilizable.

En este proyecto, el alcance se limitará a la implementación de una solución válida que haga uso del protocolo AMQP dentro del módulo Visual Tracking de EMI SUITE, que se encarga del seguimiento de los procesos productivos de los materiales necesarios para la empresa cliente.

5.1. USO DE VISUAL TRACKING

El módulo Visual Tracking (VT) permite a sus usuarios realizar un seguimiento sencillo de los procesos que se inician, interrumpen o finalizan en su sistema. Se les presenta esta información gráficamente, a través dfe un diagrama temporal que también es editable si se requiere. Los datos se van actualizando en tiempo real.

Dentro de VT, los usuarios se dividen en dos grupos:

i) **Administradores**. Tienen control total sobre el sistema, pudiendo conceder o expulsar de cada grupo a otros usuarios.

ii) **Operarios**. Tienen acceso con permisos mínimos, que en este módulo significa que solo tendrán acceso al panel de operario (cuyo funcionamiento se explicará más adelante).

El administrador, a través del menú principal, puede generar una nueva orden de trabajo, que estará compuesta de aquellos procesos que sean necesarios para llevarla a cabo, especificando sus materiales, maquinaria necesaria y otros datos, algunos opcionales. Todas estas órdenes, después de generadas, pasan por las siguientes fases:

1) **Planificación** (temporal). Establecida en función de sus características.

2) **Producción**. Fase en la que la orden se está llevando a cabo y en la que los operarios pueden introducir datos al respecto de esta.

3) **Validación**. Etapa en la que la tarea ya habría finalizado y solo falta verificar definitivamente que se ha cerrado de manera correcta.

Las órdenes pueden generarse desde cero o a través de una plantilla basada en una orden ya definida anteriormente, y manualmente se pasan a la siguiente etapa tras haberse dado los pasos necesarios para completar la anterior. Cada actividad activa en estas fases puede verse

en la pantalla inicial de Visual Tracking, mediante un *dashboard*. Las órdenes que estaban en la etapa de validación y ya fueron verificadas desaparecen de éste.

El panel de operario es la interfaz a través de la cual los trabajadores suministran los datos sobre el trabajo que han realizado. Una vez seleccionada la máquina con la que van a trabajar, tendrá acceso a opciones que les permitirán interrumpir el proceso, informar de que se ha realizado un descanso, marcar materiales como usados durante la producción, indicar que el proceso se ha acabado, etcétera. El uso del ese panel será la principal fuente de envío de mensajes entre los módulos de EMI Suite.

Dentro de Visual Tracking, la conexión con el módulo My Factory resulta de especial importancia, dado que es dicho módulo el que almacena toda la maquinaria disponible y su estado. Un error que produzca una incoherencia entre los datos de ambos módulos tiene un alto riesgo de generar fallos críticos a la hora de seleccionar maquinaria para realizar un proceso, pues esta podría no existir o no estar operativa. Por esto, la falta de medidas para manejar los casos en los que mensajes enviados por un módulo no lleguen a su remitente es una carencia grave.

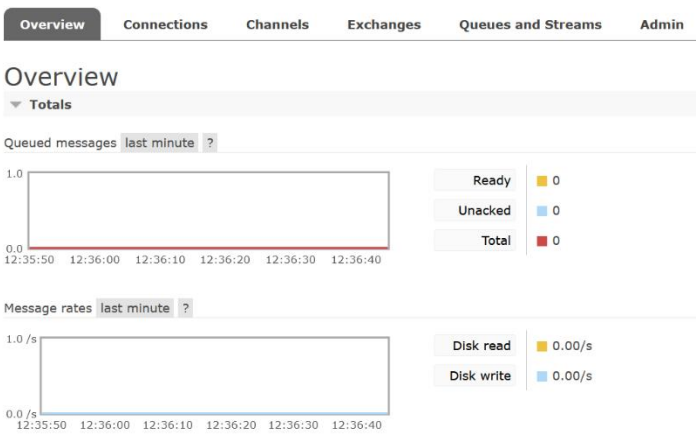
5.2. RABBIT MANAGEMENT

Para la gestión y supervisión de las colas de mensajerías, intercambios y los demás elementos que conlleva el uso del estándar de mensajería empleado por EMI SUITE, la aplicación utiliza el software de uso abierto RabbitMQ, que actúa como Middleware para el intercambio de mensajes entre diferentes aplicaciones [3].

RabbitMQ soporta el protocolo AMQP para operar, y almacena los intercambios, colas, claves de enrutamiento que los enlazan definidos en la aplicación, y los mensajes que son enviados desde esta. Permite la creación de usuarios con distintos niveles de acceso a la información y control de los parámetros.

Cuenta con una interfaz gráfica accesible a través del localhost. Para poder acceder a ella con todos los permisos necesarios para alterar los datos de la aplicación, se debe entrar con un usuario con la etiqueta “Administrador”.

La pantalla inicial de RabbitMQ ofrece información general sobre los mensajes enviados y conexiones abiertas, como se puede observar en la figura 2.



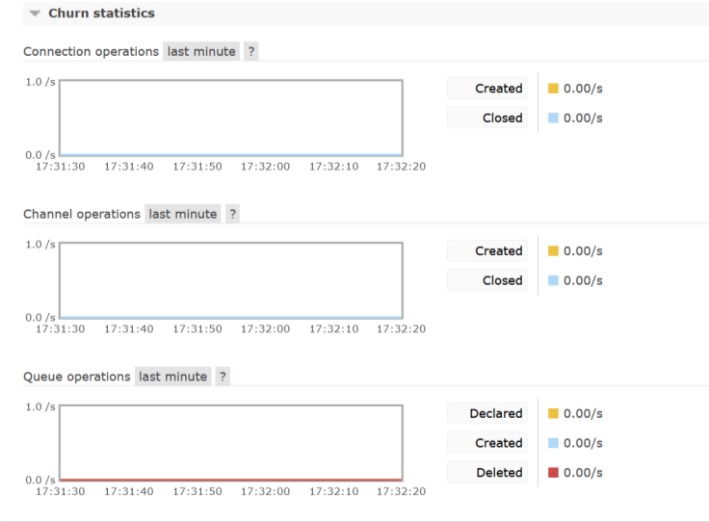


Figura 3: Pestaña de información general de Rabbit Management

Es posible generar Colas, (figura 4), e intercambio, (figura 5) directamente desde la interfaz gráfica además de consultarlos, con las mismas propiedades que se le pueden aplicar al generarlas desde código.

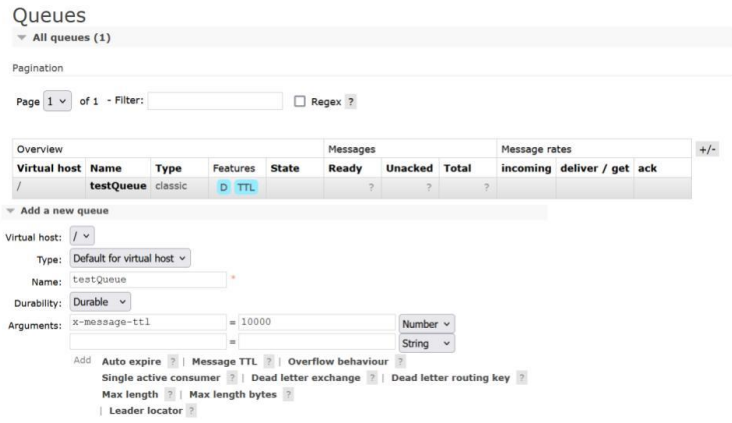


Figura 4: Declaración de una cola en Rabbit Management.

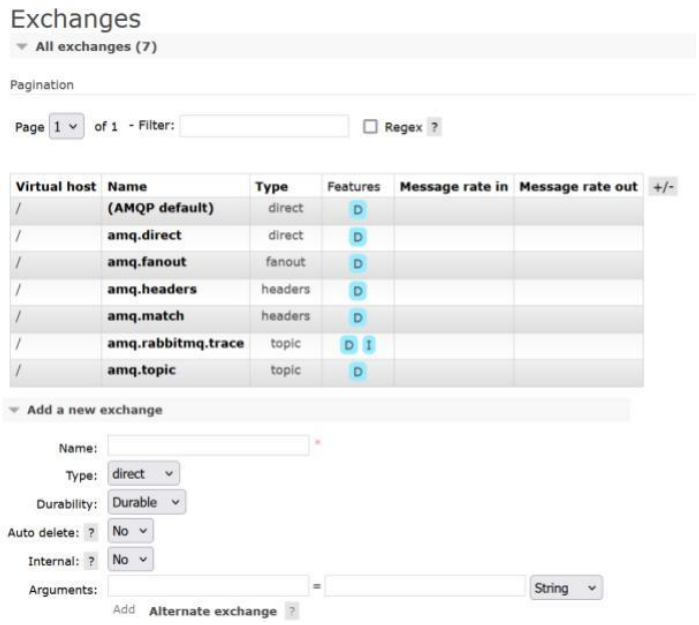


Figura 5: Creación de un intercambio en Rabbit Management.

5.3. DECLARACIÓN DE ELEMENTOS EN CÓDIGO

En esta sección se explicará la definición de los elementos necesarios para emplear el protocolo mediante código.

5.3.1. DECLARAR MENSAJES

A través de un objeto RabbitTemplate [4].

Esta es una clase de ayuda que proporciona la biblioteca **amqp** de Rabbit en Java. Ofrece métodos que simplifican la transmisión de información. Las funciones que lo emplean solo necesitan preparar el objeto de cualquier clase que deseen mandar y utilizar el método **ConvertAndSend** de RabbitTemplate, al que se le debe proporcionar el intercambio al que enviar y su clave de enrutamiento.

Este es un ejemplo de uso del RabbitTemplate:

```
if(result != null) {

    // AMQP message notification

    /* Se define la clave de enrutamiento que se va a asignar al mensaje. */
    String routingKey = String.format(manufacturingActivityRoutingKey, "create.registry");

    /*El mensaje es una clase definida en la aplicación. */
    MessageManufacturingActivityDto message = new MessageManufacturingActivityDto();

    /* Se establece el intercambio (objeto Java tipo Exchange) como campos del mensaje.
     * Esto es de utilidad si el mensaje para mantener esa información si el mensaje
     * no puede enviarse a su destino exitosamente. */
    message.setExchange(exchange);
    message.setRoutingKey(routingKey);

    /* Se usa el objeto template, definido en la clase, para realizar el envío. */
    template.convertAndSend(exchange, routingKey, message);

}
```

El objeto RabbitTemplate puede definirse como un Bean en una clase de configuración. De este modo, se definen las características del template de forma general para que todas las clases lo puedan aprovechar, y no se carga innecesariamente el entorno generando instancias del objeto para cada envío.

5.3.2. DECLARAR COLAS DE MENSAJES

Se genera un Bean de tipo Queue para generar la cola al arranque de la aplicación. El constructor recibe un String cuyo valor es el nombre que se le asignará a la cola, tres argumentos booleanos, y un mapa para representar los parámetros opcionales.

El primer argumento booleano define si la cola se mantiene guardada o se elimina en el caso de que el bróker se cierre (True si la cola se mantiene, False en caso contrario). Para la aplicación EMI Suite, interesa que los contenidos de las colas no se pierdan aunque el bróker de mensajería se reinicie, por lo que aquí se establecerá siempre a True.

El segundo booleano define su exclusividad. Si se le da True como valor, la cola solo podrá ser utilizada por la conexión que la ha declarado. Ya existe otro argumento opcional que permitirá garantizar que solo haya un consumidor activo por cada cola de mensajes, por lo que esta exclusividad es innecesaria y se dejará a False.

El tercer booleano decide si la cola se borra en el momento en el que no se encuentre en uso. (True si se desea este funcionamiento, False en caso contrario). Se quiere que las colas sean permanentes en el sistema y solo puedan borrarse manualmente, y por lo tanto este valor también se deja a False.

En último lugar está el mapa con los parámetros opcionales. Este mapa debe tener como key un String, que representará el nombre del argumento AMQP, y como valor un Object, puesto que diferentes argumentos pueden requerir valores de distintas clases.

A continuación, se muestra cómo se definiría una cola con las características básicas anteriormente descritas y un tiempo de vida como parámetro adicional:

```
@Bean
Queue defaultQueue() {

    Map<String, Object> args = new HashMap<>();

    args.put("x-message-ttl", 10000);

    return new Queue(defaultQueue, true, false, false, args);

}
```

5.3.3. DECLARAR INTERCAMBIOS

Existen tres objetos relativos a los intercambios, llamados DirectExchange, FanoutExchange y TopicExchange, que se corresponden con los tres tipos descritos anteriormente. Pueden declararse de la siguiente manera:

```
@Bean
DirectExchange directExchange() {

    return new DirectExchange("[nombre del intercambio tipo Directo]");
}

@Bean
FanoutExchange fanoutExchange() {

    return new FanoutExchange("[nombre del intercambio tipo Fanout]");
}

@Bean
TopicExchange topicExchange() {

    return new TopicExchange("[nombre del intercambio tipo Topic]");
}
```

5.3.4. DECLARAR ENLACES

Para establecer las uniones entre un intercambio y las diferentes colas, se utilizan las clases **Java Binding** y **BindingBuilder**.

Binding es la representación del enlace entre la cola y y el intercambio, mientras que **BindingBuilder** es una clase auxiliar mediante la cuál se generan las instancias de la clase anterior.

Para construir un objeto de enlace, basta con usar el método bind() de **BindingBuilder**, al que se le debe pasar la cola y el intercambio a enlazar, (no sus nombres, sino los objetos que los representan ya construidos), y un String que representa la clave de enrutamiento con el cuál se enlazan.

```
@Bean
Binding binding(Queue queue, TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with("key");
}
```

En un contexto real, habrá múltiples Beans de la misma clase que representarán las distintas colas o intercambios usados en la aplicación. Será necesario usar la anotación Spring ”Qualifier” para indicar a la función cuál es el Bean específico que se debe inyectar, como en el siguiente ejemplo:

```
@Bean
Binding binding(@Qualifier("AnSpecificQueue") Queue queue, TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with("key");
}
```

6. ESTADO INICIAL DE VISUAL TRACKING

Se comenzó a trabajar en este proyecto a través del código fuente del módulo Visual Tracking en el estado en el que se encontraba el 13 de noviembre de 2023. Disponía de dos clases de configuración para la integración que se había elaborado hasta el momento de AMQP. “RabbitPublisherConfig” y “RabbitListenerConfig”, y un fichero para almacenamiento y lectura de constantes nombrado environment.properties.

En environment.properties se definen los nombres del intercambio (solo uno en este caso), las claves de enrutamiento y las colas, que serán leídos como constantes en el código fuente. La mayoría de las claves utilizan los caracteres “%s”, por las clases que pueden generar mensajes con distintas claves pero que comparten una base inicial. Por ejemplo: Un método de creación de la clase “vtManufacturingActivity” envía mensajes al intercambio con el valor de clave “vt.manufacturing.activity.**create**”, y un método de borrado las manda con “vt.manufacturing.activity.**delete**”. De esta forma es fácil identificar y agrupar los mensajes almacenados en función de qué clase los haya generado, pero distinguiéndo claramente qué función representaban.

```
## Exchange
rabbitmq.exchange=emisuite.topic

## Manufacturing label routing key
rabbitmq.routingKey.manufacturingLabel.printer=vt.manufacturingLabel.printer

## Queue
rabbitmq.queue.shiftUpdate=vt.shift.update
rabbitmq.queue.workstationStop=vt.workstationStop.update.dev

## Manufacturing routing keys
rabbitmq.routingKey.manufacturing.activity=vt.manufacturing.activity.%s
rabbitmq.routingKey.manufacturing.material=vt.manufacturing.material.%s
rabbitmq.routingKey.manufacturing.operator=vt.manufacturing.operator.%s
rabbitmq.routingKey.manufacturing.output=vt.manufacturing.output.%s
```

En la clase de configuración RabbitPublisherConfig se define un único intercambio, al que estarían enlazados todas las colas de la aplicación, junto con el convertidor de formato de los mensajes enviados por las aplicaciones.

```
/**
 * Exchange to be used for Rabbit connection
 * @return the exchange, durable and non auto-delete
 */
@Bean
TopicExchange exchange() {

    return new TopicExchange(exchange);
}

/**
 * JSON message converter using Jackson 2 JSON library
 * @return the JSON message converter
 */
@Bean
Jackson2JsonMessageConverter producerJackson2MessageConverter() {

    return new Jackson2JsonMessageConverter();
}
```

El constructor de RabbitListenerConfig recibe como parámetros el nombre del host y número de puertos necesarios para conectarse al bróker de RabbitMQ, y credenciales, (usuario y contraseña), para su utilización, junto con un parámetro booleano que determina si en el servicio se debe habilitar el protocolo SSL para cifrado de mensajes.

```
public RabbitListenerConfig(WsStompClient wsStompClient, ServiceConfiguration serviceConfig,
        @Value("${jndi.env.spring.rabbitmq.host.name}") String jndiEnvSpringRabbitmqHostName,
        @Value("${jndi.env.spring.rabbitmq.port.name}") String jndiEnvSpringRabbitmqPortName,
        @Value("${jndi.env.spring.rabbitmq.username.name}") String jndiEnvSpringRabbitmqUsernameName,
        @Value("${jndi.env.spring.rabbitmq.password.name}") String jndiEnvSpringRabbitmqPasswordName,
        @Value("${jndi.env.spring.rabbitmq.ssl.enabled.name}") String jndiEnvSpringRabbitmqSslEnabledName,
        IManufacturingStopService manufacturingStopService,
        IRangeActivityService rangeActivityService) throws NamingException {

    this.wsStompClient = wsStompClient;
    this.serviceConfig = serviceConfig;
    this.manufacturingStopService = manufacturingStopService;
    this.rangeActivityService = rangeActivityService;

    JndiTemplate jndiTemplate = new JndiTemplate();

    this.host = (String)jndiTemplate.lookup("java:/comp/env/" + jndiEnvSpringRabbitmqHostName);
    this.port = (Long)jndiTemplate.lookup("java:/comp/env/" + jndiEnvSpringRabbitmqPortName);
    this.username = (String)jndiTemplate.lookup("java:/comp/env/" + jndiEnvSpringRabbitmqUsernameName);
    this.password = (String)jndiTemplate.lookup("java:/comp/env/" + jndiEnvSpringRabbitmqPasswordName);
    this.sslEnabled = (Boolean)jndiTemplate.lookup("java:/comp/env/" + jndiEnvSpringRabbitmqSslEnabledName);
}
```

Con un bean del tipo ConnectionFactory se configura dicha interfaz, que sirve para gestionar las conexiones con el servidor de RabbitMQ por el que circulan los mensajes. Utiliza como parámetros los mismos argumentos de entrada que recibió el constructor de la clase.

```
@Bean
ConnectionFactory connectionFactory() {

    CachingConnectionFactory connectionFactory = new CachingConnectionFactory();

    connectionFactory.setHost(host);
    connectionFactory.setUsername(username);
    connectionFactory.setPassword(password);
    connectionFactory.setPort(port.intValue());

    if (sslEnabled) {
        try {
            connectionFactory.getRabbitConnectionFactory().useSslProtocol();
        } catch (Exception e) {
            LOGGER.error("SSL ERROR");
        }
    }

    return connectionFactory;
}
```

Se define un Logger para registrar información, alertas y fallos producidos en el sistema, y dos interfaces de servicio que corresponden a clases en las que se encuentran integradas las notificaciones AMQP.

```
static final Logger LOGGER = LoggerFactory.getLogger(RabbitListenerConfig.class);

private final IManufacturingStopService manufacturingStopService;
private final IRangeActivityService rangeActivityService;
```

Los nombres del intercambio, las claves de enrutamiento y las colas son leídas del fichero environment.properties y almacenadas en variables tipo String en esta clase de configuración.

```
/**
 * Exchange
 */
private @Value("${rabbitmq.exchange}") String exchange;

/**
 * MF response queue and routing keys
 */
private @Value("${rabbitmq.queue.shiftUpdate}") String shiftUpdateQueue;

private @Value("${rabbitmq.queue.workstationStop}") String workstationStopQueue;

/**
 * Routing key of the time interval modification
 */
private @Value("${rabbitmq.routingKey.timeInterval}") String timeIntervalRoutingKey;
```

En esta clase también se declaran los enlaces al intercambio y sus Listeners (serán los consumidores de mensajes y en su lógica contienen la respuesta que deben dar al recibirlo), utilizando las variables String anteriormente almacenadas como argumentos de entrada para generar los beans requeridos.

```
    @Bean
    Queue shiftUpdateQueue() {

        Map<String, Object> args = new HashMap<>();

        /* Tiempo de vida para un mensaje publicado en esta cola, definido en milisegundos.
         * Si pasan 10 segundos sin que se consuma, el mensaje es descartado de la cola. */
        args.put("x-message-ttl", 10000);

        return new Queue(shiftUpdateQueue, true, false, false, args);

    }

    @Bean
    Queue workstationStopQueue() {

        Map<String, Object> args = new HashMap<>();

        /* Tiempo de vida para un mensaje publicado en esta cola, definido en milisegundos.
         * Si pasan 10 segundos sin que se consuma, el mensaje es descartado de la cola. */
        args.put("x-message-ttl", 10000);

        return new Queue(workstationStopQueue, true, false, false, args);

    }

@Bean
Binding timeIntervalBinding(@Qualifier("shiftUpdateQueue") Queue shiftUpdateQueue,
        TopicExchange exchange) {

    return BindingBuilder.bind(shiftUpdateQueue).to(exchange).with(timeIntervalRoutingKey);

}

@Bean
Binding assetCalendarBinding(@Qualifier("shiftUpdateQueue") Queue shiftUpdateQueue,
        TopicExchange exchange) {

    return BindingBuilder.bind(shiftUpdateQueue).to(exchange).with(assetCalendarRoutingKey);

}
```

Una instancia de la clase MessageHandlerMethodFactory es generada como otro bean y se pasa como parámetro a la clase de configuración RabbitListenerConfigurer, para los métodos de escucha. Esta factoría define el comportamiento a seguir en la transmisión de mensajes, asignándole un convertidor. En este caso, el convertidor utiliza una clase de mapeado de objetos a Jackson, como se explicó anteriormente.

```java
/**
 * Handler method factory for processing messages
 *
 * @return the message handler method factory
 */
@Bean
MessageHandlerMethodFactory messageHandlerMethodFactory() {

    DefaultMessageHandlerMethodFactory factory = new DefaultMessageHandlerMethodFactory();
    MappingJackson2MessageConverter jsonConverter = new MappingJackson2MessageConverter();

    jsonConverter.getObjectMapper().registerModule(new ParameterNamesModule(JsonCreator.Mode.PROPERTIES));
    factory.setMessageConverter(jsonConverter);

    return factory;

}

/**
 * Configuration of Rabbit listener
 *
 * @param messageHandlerMethodFactory message handler method factory
 * @return the configuration for the Rabbit listener
 */
@Bean
RabbitListenerConfigurer rabbitListenerConfigurer(MessageHandlerMethodFactory messageHandlerMethodFactory) {

    return c -> c.setMessageHandlerMethodFactory(messageHandlerMethodFactory);

}
```

7. PROBLEMAS DE LA IMPLEMENTACIÓN DE PARTIDA

La implementación que Soincon había realizado del protocolo AMQP en EMI Suite presentaba las siguientes limitaciones:

- **Dependencias circulares:** Existen en las clases de configuración métodos Listener con dependencias a otros componentes levantados como Beans de la aplicación. En un caso, esto ya ha causado un problema en el código que Soincon ya había implementado. El Listener requería de una clase de servicio llamada manufacturingStopServiceImpl que, a su vez, necesitaba un objeto RabbitTemplate que dependía de un bean aún no construido de la propia clase de configuración. Por lo cuál, saltaba un error que se muestra en la figura 6. Sin reajustes, Java no es capaz de escapar de este bucle.

- **Modularización del código:** En la implementación de partida de RabbitMQ en EMI Suite, todas las colas empleadas por la aplicación, así como sus enlaces a los intercambios y sus métodos Listener, se generan como Beans en una misma clase de configución. Esto, a medida que el uso del protocolo AMQP se extienda dentro de la aplicación, y con ello el número de colas, provocaría que esta clase de configuración fuera creciendo y complicándose en exceso.

- **Límite de un consumidor por cola:** Para la lógica de Emi Suite, se quiere que ninguna cola de mensajes pueda tener dos o más consumidores a la vez. Tal y como estaban implementadas originalmente, no había ningún tipo de mecanismo implementado que garantizara el cumplimiento de esto.

- **Procesamiento de mensajes:** No está programada la confirmación o negación de envío de mensajes. Por lo tanto, si un mensaje no puede ser procesado correctamente por su receptor, expira o no ha podido enviarse a ninguna cola, el método que publicó el mensaje no tiene modo alguno de saberlo. Esto abre la puerta a errores posteriores en la ejecución de otras funciones.

8. SOLUCIONES PROPUESTAS

Para corregir los problemas de partida que contiene la implementación de la que se parte, se han diseñado y llevado a cabo las siguientes soluciones para cada uno:

8.1. SOLUCIÓN A DEPENDENCIAS CIRCULARES

Ambas problemáticas pueden solucionarse moviendo el código que no vaya a ser usado por toda la aplicación fuera de las clases de configuración. Las declaraciones de las colas, sus listeners y sus enlaces a los intercambios pueden moverse a las respectivas clases que los utilizan sin que esto afecte al funcionamiento del sistema en su conjunto. Esto evita la aparición de las dependencias circulares, dado que los Beans relacionados con esas funciones ya no se encontrarán en ningún caso en la misma clase que los Beans de la clase de configuración que los requieren, y evitará el crecimiento desmedido de dicha clase de configuración. Además, desde un punto de vista organizativo también tiene más sentido que los Listeners se encuentren fuera de esta, dado que el código que contienen es parte de la lógica de negocio de la aplicación.

8.2. SOLUCIÓN PARA ESTABLECER LÍMITE DE UN CONSUMIDOR POR COLA

Las colas definidas en la implementación de la que se parten no garantizaban esta característica. La manera de lograr el funcionamiento que se busca es proporcionar un nuevo argumento opcional en el momento de declararlas: “x-single-active-consumer”. Cuando el valor de este argumento se establece a true, RabbitMQ internamente se encargará de que en ningún momento una cola tenga más de un consumidor activo. Solo concederá permiso a otro en caso de que el actual se desconecte. Esta solución implica el redeclarar las colas utilizadas por Visual Tracking de la forma en la que se ve a continuación para incluir este argumento.

```
@Bean
Queue testQueue() {

    Map<String, Object> args = new HashMap<>();
    args.put("x-single-active-consumer", true);

    return new Queue("testQueue", true, false, false, args);

}
```

8.3. SOLUCIÓN PARA PROCESAMIENTO DE MENSAJES (NEGACIÓN MANUAL)

La biblioteca del software RabbitMQ ofrecen mecanismos que permiten la posibilidad de habilitar los rechazos manuales del mensaje, pero la implementación que Soincon hacía de este hasta ahora no lo tenía en cuenta. El productor generaba el mensaje, lo transmitía y, al no existir ninguna gestión respecto a los mensajes no rechazados o no recibidos, en la práctica daba por hecho que el envío y recepción se habían producido con éxito en todas las ocasiones. El esquema de la figura 7 representa el procesamiento de cada mensaje con el código fuente en este estado. Se omitirán los intercambios que actúen de intermediarios, por simplificación.

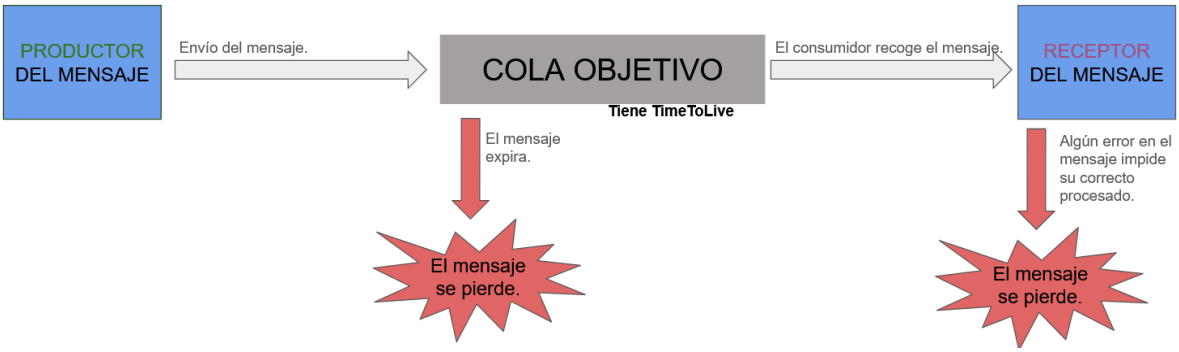


Figura 7: Esquema del envío de mensajes sin control de errores

Para corregir este defecto en la gestión de los mensajes, será necesario modificar el código de los Listeners del sistema.

En primer lugar, será necesario añadir el parámetro “ackMode” en la anotación RabbitListener, y establecerlo como valor “MANUAL”. Esto habilita la confimación manual de los mensajes en ese Listener.

```
@RabbitListener(queues = "${rabbitmq.queue.default}", ackMode = "MANUAL")
```

En la definición de la propia función anotada como Listener, es posible añadir como argumento de entrada un objeto de tipo Channel que representa el canal a través del cual se envió el mensaje dentro de la conexión utilizada. También será necesario añadir un nuevo parámetro de tipo long precedido por la anotación Header para poder acceder a los datos de la cabecera

del mensaje, información entre la que se encuentra el Delivery Tag, la cifra que identifica un envío concreto dentro de un canal.

```
public void queueDefaultListener(ClaseDTO objetoRecibido, Channel channel,
        @Header(AmqpHeaders.DELIVERY_TAG) long tag)
```

Una vez añadidos estos parámetros, en función de la lógica necesaria en cada caso puede decidirse si se envía un NotAcknowledgement, través del método de la clase Channel basicNack(). Esta función recibe tres argumentos de entrada:

- **Delivery tag**. Ya se ha obtenido automáticamente desde la cabecera del mensaje al haber utilizado la etiqueta Header, y es lo que permite identificar qué mensaje se está rechazando.

- **Multiple** (booleano). Determina si el rechazo es múltiple o no; si el valor es true, se considera que se han rechazado todos los mensajes con un valor de delivery tag igual o inferior al proporcionado como argumento anterior, mientra que si es false, el rechazo solo se aplica al mensaje con el delivery tag dado. En el caso de Emi Suite, se quiere que estos rechazos sean estrictamente individuales siempre, por lo que se establece a false en todos los casos.

- **Requeue** (booleano). Si se establece a true, el mensaje rechazado vuelve a incorporarse al inicio de la cola de la que se consumió. Con el valor false, se tratará el mensaje de la misma forma en la que hubiera gestionado un mensaje que expira o que deba ser eliminado por exceso de datos en la cola a donde se envió. Por lo tanto, en este caso el procesamiento de este mensaje dependerá de qué gestión haga la cola de estas situaciones. Nuevamente en el caso de EMI Suite, este valor siempre se configurará como false.

```
channel.basicNack(tag, false, false);
```

8.4. SOLUCIÓN PARA PROCESAMIENTO DE MENSAJES (POLÍTICA DE REINTENTOS)

Aún con la configuración manual de rechazo de mensajes, el problema persistiría. Al no existir un manejo de los mensajes que han sido descartados de la cola de mensajería por un motivo u otro, estos desaparecerían del sistema tras no ser correctamente consumidos, por lo que el resultado sería el mismo.

La inclusión de una política de tratamiento e intentos de reenvío de los datos rechazados conllevará la creación de nuevos intercambios, a los que se les asociarán nuevas colas destinadas exclusivamente a estas gestiones.

En primer lugar, será necesario modificar todas las colas asociadas al único intercambio que hasta ahora utilizaba EMI Suite, para añadirles un nuevo argumento en su creación. “x-dead-letter-exchange". A este parámetro opcional se le da como valor una cadena de caracteres que debería coincidir exactamente con el nombre de otro intercambio activo en el sistema.

De esta manera, en el momento en el que un mensaje sea descartado de la cola, este se reenviará con la misma información al intercambio con el nombre indicado. Existe un parámetro “x-dead-letter-routing-key" que permite especificar la clave de enrutamiento con la que se realizan los reenvíos. Para la gestión planeada, las claves no deberían alterarse en ningún punto, por lo que este parámetro no se utiliza.

El valor que todas las colas actuales de Visual Tracking le darán a ese argumento es “emisuite.deadLetter”, que representa a un nuevo intercambio también de tipo topic. A este deadLetterExchange se le enlazan colas con las mismas claves que los de las colas cuyos mensajes descartados deben tratar. El resultado de la adición de esta cola se muestra en el esquema de la figura 8.

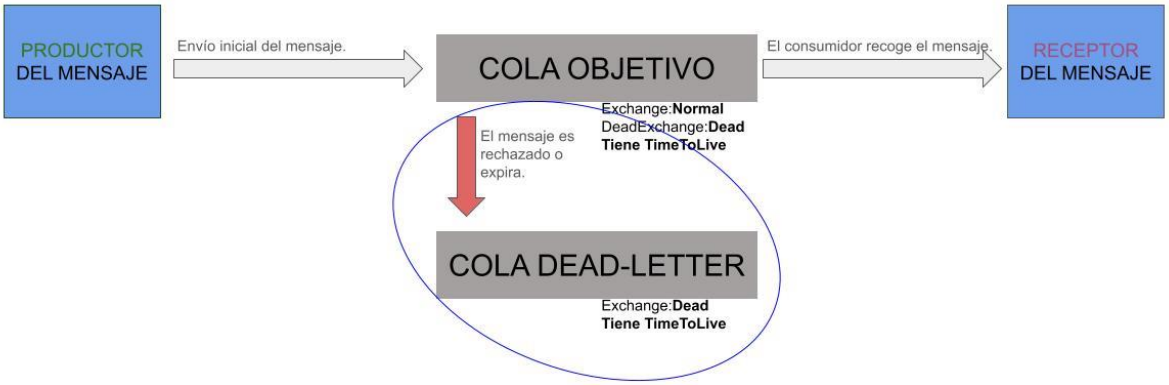


Figura 8: Esquema del envío de mensajes con cola Dead-Lettering

Las funciones de Listeners de las colas asociadas a Dead-Lettering se encargarán de manejar el reenvío en función de la información que encuentren en el mensaje.

Para tener acceso a datos relevantes para decidir qué hacer con el mensaje recibido, estas colas leerán nueva información de la cabecera. Cuando un mensaje se retira de una cola por rechazo o expiración, de manera automática se le añade un nuevo array a su cabecera de nombre “x-death”. Se puede obtener directamente ese header de la manera que se muestra a continuación:

```
public void queueDeadLetterListener(Message message, Channel channel,
        @Header(name = "x-death") List<Map<String, Object>> deathHeader)
```

Este array contiene, entre otros, dos datos de relevancia para la política de reintentos en la aplicación:

- Parámetro **reason**. Es un String que especifica cuál fue el motivo de que el mensaje haya sido descartado. Sus posibles valores son los tres siguientes:
    - **rejected**. Indica que el consumidor original mandó un NotAcknowledgmenet explícitamente.
    - **expired**. Indica que el tiempo de vida especificado en la cola transcurrió antes de que el mensaje fuera consumido.

- **maxLen**. Indica que el mensaje se expulsó debido a ser el más antiguo en el instante en el que se rebasó el límite de mensajes almacenables por la cola. Este último control no se emplea en EMI Suite, por lo que solo se tendrán en cuenta los dos primeros posibles valores.

Estos valores se extraen en código del siguiente modo:

```
String deathReason = deathHeader.get(0).get("reason").toString();
if (deathReason.equals("expired")) {
    /* Hacer algo */
}
if (deathReason.equals("rejected")) {
    /* Hacer algo */
}
```

- Parámetro **count**. Por cada cola de la cuál el mensaje ha sido descartado, se guarda la cantidad de veces que ha sido rechazado. Esto resulta de utilidad para llevar la cuenta del número de veces que se ha intentado reenviar una misma información. En la siguiente línea se muestra cómo obtenerlo:

```
Long retries = (Long) deathHeader.get(0).get("count");
```

Como últimos elementos añadidos a la aplicación para poder realizar estos controles, se crearán dos nuevos intercambios llamados ”emisuite.retry” y ”emisuite.parking”. Hay una cola asociada al intercambio retry por cada una enlazada al intercambio dead-letter, y tienen como valor del parámetro ”x-dead-letter-exchange” el mismo nombre al que los productores envían los mensajes por primera vez, el valor de TimeToLive también activado, y ningún consumidor a la espera. De esta manera, los mensajes que lleguen a esta cola se limitarán a permanecer allí a la espera de ”expirar” para ser reenviados a la cola a la que originalmente se mandaron. Con esta parada intermedia se pretende dar prioridad a los datos que son generados y enviados por primera vez, antes de reinsertar la información reenviada a la cola. En la figura 9 se muestra el funcionamiento del envío habiendo integrado la cola de reenvíos.

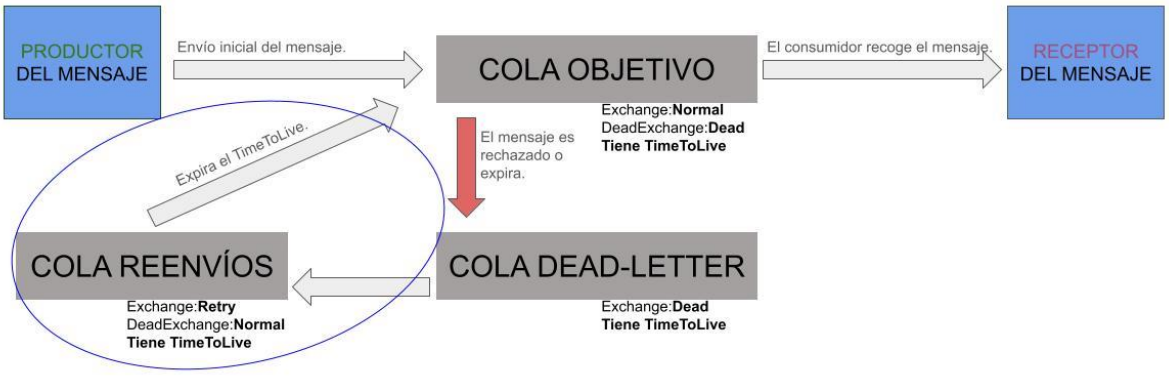


Figura 9: Esquema del envío de mensajes con cola de reenvíos.

Sin embargo, para evitar una cantidad de tiempo de espera excesivo en los mensajes rechazados, se configuran los Listeners dead-Letter de manera que si el mensaje ha expirado, en lugar de mandarlo a la cola de reintentos, se reenvíe directamente a la cola de la que procede, dado que ya habrá esperado la cantidad de tiempo especificada en el tiempo de vida de su cola. Por lo que se mandan al intercambio estándar ya anteriormente definido en EMI Suite. (Figura 10)

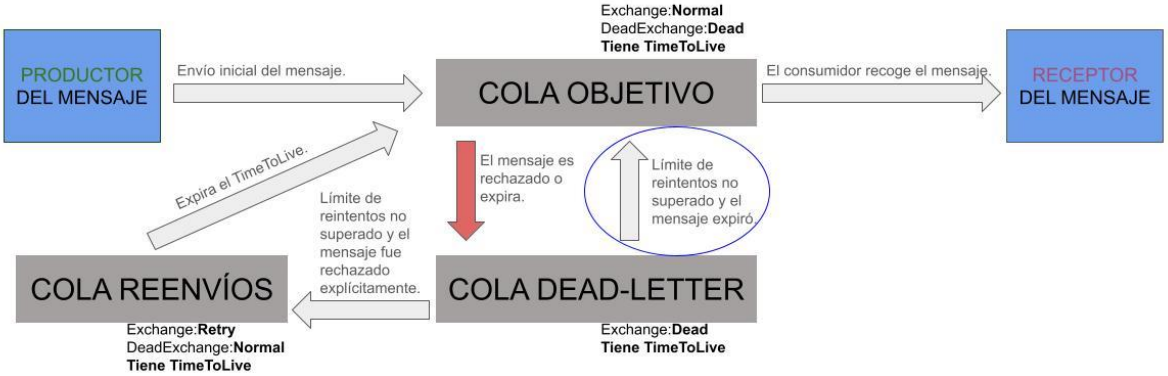


Figura 10: Esquema del envío con funcionamiento especial de Dead-Letter en caso de expiración

Las colas enlazadas al intercambio emisuite.parking tampoco tendrán ningún consumidor asociado. Esta colas están destinadas a almacenar los mensajes por un tiempo indefinido para su posterior revisión manual. Por ello, no se le dará un tiempode vida ni se definirá el argumento ”x-dead-letter-exchange". Poseyendo estas características, esta cola nunca retirará su contenido de manera automática mientras tenga capacidad para soportar más información. Como las claves de enrutamientos nunca cambian en el tránsito del mensaje por ese motivo, en una revisión manual se podrá identificar a dónde iba destinado cada uno sin dificultad. Además, este se añade como “x-dead-letter-exchange" a las colas del exchange.deadLetter, para que, en caso de que los Listeners de una de estas fallaran, el mensaje trate de enviarse directamente al parking, minimizando más la posibilidad de pérdida.

Originalmente se planteó que hubiera una cola de parking por cada cola de Dead-Lettering, al igual que con las del exchange.retry. Pero al ser esta una cola que solo almacena y a la que deberá entrarse con cierta regularidad a revisar el contenido, se ha decidido que solo se genere una para todo el módulo de Visual Tracking de EMI Suite, para simplificar la localización de mensajes descartados. En la figura 11 se muestra el esquema habiendo añadido la cola de parking.

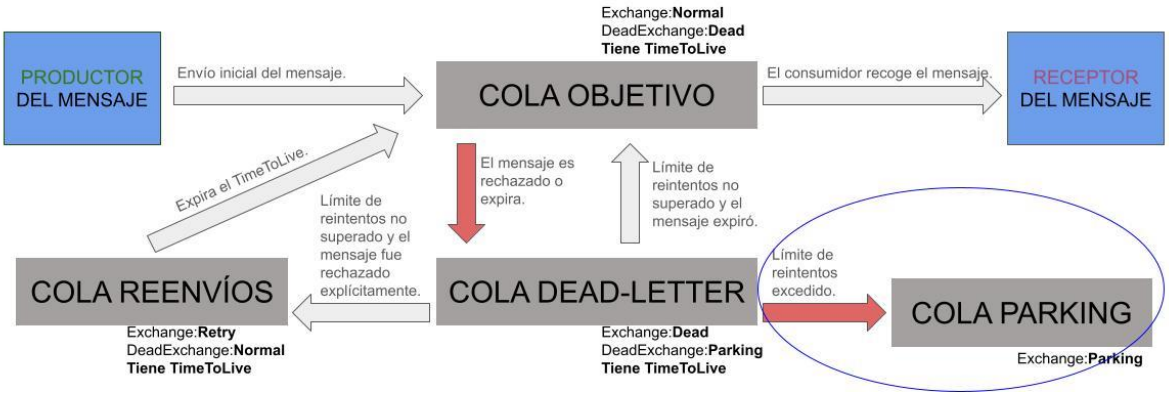


Figura 11: Esquema del envío con cola de parking

Como última posible situación a cubrir con los mecanismos que ofrece la biblioteca de **amqp**, está la posibilidad de que, al enviarse un mensaje, este no llegue al alcanzar su cola de destino.

Si no se establece ninguna configuración, por defecto el productor no realiza ninguna acción en caso de que el mensaje no llegue a su destino, pues no lo detectaría. Para cambiar esto, se le puede pasar al objeto template de envío un “ReturnCallBack”.

Cada RabbitTemplate puede tener definido un solo ReturnCallBack, y este debe implementar la función returnedMessage(). Esta función se activará. Para que esta funcionalidad pueda producirse, la ConnectionFactory que se le proporcionó al Template en su declaración debe activar estos retornos con la función setPublisherReturns().

```
connectionFactory.setPublisherReturns(true);
```

La función returnedMessage() recibe como argumentos, entre otros, parámetros que recogen la siguiente información: El mensaje que se intentó transmitir en sí mismo, el intercambio al que se envió, y la clave de enrutamiento con el que se mandó. Al disponer de estos datos, es posible codificar una implementación que actúe de manera distinta según a dónde fuera destinado el mensaje.

Para EMI Suite, se ha decidido que, si el mensaje había intentado mandarse a la cola de parking de Visual Tracking, el mensaje no se reenviará a ninguna cola, pero toda la información disponible a través de la función se convertirá a String y se guardará en el log. Con esto se evita cargar e incluso posiblemente colapsar el sistema debido a mensajes que estarían circulando indefinidamente al no poder entrar en la cola de parking, pero se consigue que al menos la información del mensaje no sea totalmente perdida. En caso de que el mensaje hubiera intentado ser mandado a una cola enlazada a cualquier otro intercambio, sí que se hará un intento de mandar el mensaje directamente a parking para su revisión manual posterior.

```
template.setReturnsCallback((returned) -> {
    if (returned.getExchange().equals(parkingExchange)) {
        LOGGER.error("Visual Tracking's parking queue failed to recieve the message with RoutingKey {}."
                + " Message could not be saved", returned.getRoutingKey());
        LOGGER.info("Message data: {}", returned.getMessage().toString());
    } else {
        LOGGER.warn("Message to Exchange {} with RoutingKey {} was unroutable",
                returned.getExchange(), returned.getRoutingKey());
        LOGGER.info("Sending message with RoutingKey {} to Visual Tracking's parking queue",
                returned.getRoutingKey());
        template.convertAndSend(parkingExchange, returned.getRoutingKey(), returned);
    }

});
```

La figura 12 muestra el esquema final de la política de reintentos que se va a implementar.

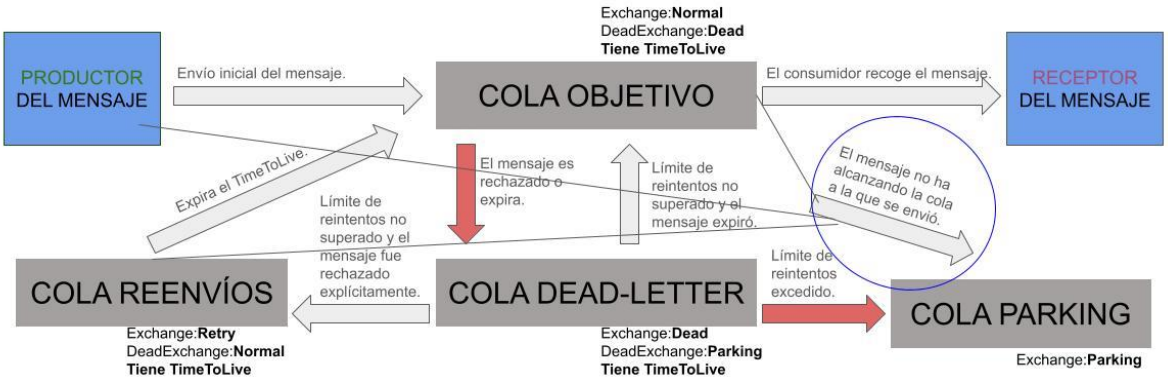


Figura 12: Esquema del envío definitivo

9. IMPLEMENTACIÓN DEL DISEÑO EN EL MÓDULO VISUAL TRACKING

En este apartado se explicará cómo se ha modificado el código para integrar la solución planteada en la parte del módulo Visual Tracking que ya estaba implementando notificaciones AMQP.

9.1. FICHERO ENVIRONMENT.PROPERTIES ACTUALIZADO

Es necesario añadir nuevos parámetros al fichero de propiedades que serán leídas desde las clases de configuración y de servicio.

En primer lugar, se añaden los siguientes identificadores para los tres nuevos intercambios de Visual Tracking:

```
rabbitmq.deadLetterExchange=emisuite.deadLetter
rabbitmq.retryExchange=emisuite.retry
rabbitmq.parkingExchange=emisuite.parking
```

También se añadirán nombres únicos para las nuevas colas de mensajería de reintentos y de dead-letter de cada clase de negocio. Las claves de enrutamiento se mantienen iguales, al no modificarse en ningún momento del protocolo de reintentos.

Se añaden unas constantes totalmente nuevas correspondientes al número límite de reintentos de reenvío para cada clase. El Listener de la cola de Dead-Lettering leerá esta constante y la comparará con el valor "count" del *death header* del mensaje a procesar para determinar si se debe enviar ya al parking. Se han dejado en un valor de 2 para todas ellas en este caso, pero estando definidas en este fichero, si se opta por modificarlas de cara al futuro el cambio será sencillo.

```
##Retry limits
rabbitmq.retryLimit.manufacturing.activity=2
rabbitmq.retryLimit.manufacturing.label=2
rabbitmq.retryLimit.manufacturing.material=2
rabbitmq.retryLimit.manufacturing.operator=2
rabbitmq.retryLimit.manufacturing.output=2
```

Por último, se hace necesario definir una clave de enrutamiento con la que enlazar la cola de parking. Como se va a declarar una sola para todo Visual Tracking, la clave tendrá de nombre "vt.#" por lo cuál todas los mensajes enviados con una clave que empiece por vt. (es decir, todas las de Visual Tracking), llegarán a esta cola al ser enviadas al intercambio de parking.

```
rabbitmq.queue.parking.manufacturing.visualTracking=vt-parking
rabbitmq.routingKey.parking.visualTracking=vt.#
```

9.2. CLASE DE CONFIGURACIÓN ACTUALIZADA PARA VISUAL TRACKING

Las clases "RabbitPublisherConfig" y "RabbitListenerConfig" se fusionarán en una sola llamada simplemente "RabbitConfig". Como se ha visto antes, el tamaño de la clase Publisher era muy reducido, no justificándose el encontrarse en una clase separada a la del resto de la configuración.

Los cuatro intercambios a utilizar por la aplicación se declaran como Beans en dicha clase.

```
/**
 * Default exchange to be used for Rabbit connection
 * @return the exchange, durable and non auto-delete
 */
@Bean
TopicExchange exchange() {

    return new TopicExchange(exchange);

}

/**
 * Exchange the queues in charge of handling dead-lettered messages are bounded to
 * @return the exchange, durable and non auto-delete
 */
@Bean
TopicExchange deadLetterExchange() {

    return new TopicExchange(deadLetterExchange);
}

/**
 * Exchange used for retrying to send a message. Queues bounded to this exchange have no listeners
 * and simply send the message back to the default exchange after their Time To Live expires
 * @return the exchange, durable and non auto-delete
 */
@Bean
TopicExchange retryExchange() {

    return new TopicExchange(retryExchange);
}
/**
 * Exchange to be used for storing messages that surpassed the retrying limit
 * @return the exchange, durable and non auto-delete
 */
@Bean
TopicExchange parkingExchange() {

    return new TopicExchange(parkingExchange);
}
```

La rabbitTemplate existe como instancia única para todo el módulo Visual Tracking, dado que todos los mensajes enviados y descartados deben seguir el mismo protocolo de reintentos. El template será configurado de manera que use el convertidor de mensajes a Json que sigue definido como Bean en la misma clase de configuración.

Se le programa el ReturnsCallBack de manera que actúe según el esquema mostrado anteriormente. Si el mensaje que ha captado iba con destino a la cola de parking, significa que debido a un error de conexión esa cola no está disponible. Siendo esta la peor situación posible, puesto que el mensaje no podrá preservarse, se genera una entrada en el logger con un String representando los datos del mensaje, para garantizar que al menos quede registrado el objeto que intentó enviarse y hacia dónde iba dirigido.

```java
/**
 * Configuration of the Rabbit template
 *
 * @param connectionFactory the connectionFactory to use for the template
 * @return the rabbit template
 */
@Bean
RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {

    RabbitTemplate template = new RabbitTemplate(connectionFactory);
    template.setMessageConverter(new Jackson2JsonMessageConverter());
    template.setMandatory(true);

    template.setReturnsCallback((returned) -> {
        if (returned.getExchange().equals(parkingExchange)) {
            LOGGER.error("Visual Tracking's parking queue failed to recieve the message with RoutingKey {}."
                    + " Message could not be saved", returned.getRoutingKey());
            LOGGER.info("Message data: {}", returned.getMessage().toString());
        } else {
            LOGGER.warn("Message to Exchange {} with RoutingKey {} was unroutable",
                    returned.getExchange(), returned.getRoutingKey());
            LOGGER.info("Sending message with RoutingKey {} to Visual Tracking's parking queue",
                    returned.getRoutingKey());
            template.convertAndSend(parkingExchange, returned.getRoutingKey(), returned);
        }

    });

    return template;
}
```

Adicionalmente, dado que se en la implementación definitiva solo existirá una cola de parking para todo Visual Tracking, y esta no pertenecerá a ninguna clase concreta de la aplicación, esta cola y su enlace también son declarados en RabbitConfig.

```java
/**
 * Queue for the messages to stay in after having surpassed the retry limit
 *
 * @return the queue for the messages to stay in after having surpassed the retry limit
 */
@Bean
Queue visualTrackingParkingQueue() {

    return new Queue(visualTrackingParkingQueue, true, false, false, null);

}
/**
 * Binding of the parking exchange and the parking Manufacturing Label service queue
 *
 * @param printerDeadLetterQueue the parking printer queue. Must indicate the bean name of the queue
 * @param exchange the exchange to bind the queue
 * @return the exchange and queue binding
 */
@Bean
Binding ParkingBinding(@Qualifier("visualTrackingparkingQueue") Queue visualTrackingParkingQueue,
        @Qualifier("parkingExchange") TopicExchange parkingExchange) {

    return BindingBuilder.bind(visualTrackingParkingQueue).to(parkingExchange)
            .with(visualTrackingParkingRoutingKey);

}
```

9.3. ACTUALIZACIÓN DE CLASES QUE UTILIZABAN AMQP

Las siguientes clases de Visual Tracking que enviaban notificaciones AMQP en la implementación de base han sido modificadas para implementar la solución planteada en su funcionamiento.

- **ManufacturingActivityRegistryServiceImpl**: Registra los cambios producidos en las actividades que se encuentran en producción. Es decir, si se produce una parada o se reinicia una de ellas.

- **ManufacturingLabelServiceImpl:** Gestiona la creación, la actualización, la búsqueda y el borrado de las etiquetas que opcionalmente el usuario puede añadir a las órdenes de trabajo.

- **ManufacturingMaterialServiceImpl:** Gestiona la creación, la actualización, la búsqueda y el borrado de los materiales registrados en la plataforma para poder ser asignados a procesos.

- **ManufacturingOperatorServiceImpl:** Gestiona la información de los operarios que trabajan con las diferentes máquinas registradas en VT.

- **ManufacturingOutputServiceImpl:** Lleva el control de los materiales que han sido producidos como resultado de un proceso.

Para todas estas clases, la manera de implementar la solución planteada sigue los mismos pasos. Se utilizará como ejemplo para ilustrarlo la clase “ManufacturingLabelServiceImpl”

1) Se especifican en el fichero de propiedades los parámetros necesarios para implementar la solución en esta clase. Serán la constante para el número de reintentos de reenvío que quieran especificarse para esta, identificadores para las colas añadidas Dead-Lettering y Retry y una clave de enrutamiento adicional para ellas. Esta nueva clave tendrá de valor base el mismo que el de la clave que se estaba utilizando al enviar desde el resto de sus métodos, pero en vez de la combinación especial de caracteres “%s”, terminará con el caracter “#”, que representa cualquier cantidad de cualquier palabra, lográndose que todos los mensajes enviados desde esa clase lleguen hasta la cola de Dead-Lettering en caso de ser descartados.

```
rabbitmq.routingKey.manufacturing.label=vt.manufacturing.label.%s
rabbitmq.routingKey.deadLettering.manufacturing.label=vt.manufacturing.label.#
```

2) Se guardan como atributos de la clase todas estas nuevas constantes definidas para ella, además de los nombres de los tres intercambios nuevos, de la misma forma en la que se hace en la clase de configuración.

3) Se declaran las colas Dead-Lettering y Retry como Beans en el cuerpo de la clase.

```java
/**
 * Queue for handling dead-lettered messages for the Manufacturing Label service
 *
 * @return the queue for handling dead-lettered messages for the Manufacturing Label service
 */
@Bean
Queue manufacturingLabelDeadLetterQueue() {

    Map<String, Object> args = new HashMap<>();
    args.put("x-message-ttl", 10000);
    args.put("x-dead-letter-exchange", parkingExchange);

    return new Queue(manufacturingLabelDeadLetterQueue, true, false, false, args);

}

/**
 * Queue for retrying to send a message for the Manufacturing Label service
 *
 * @return the queue for retrying to send a message for the Manufacturing Label service
 */
@Bean
Queue manufacturingLabelRetryQueue() {

    Map<String, Object> args = new HashMap<>();
    args.put("x-message-ttl", 5000);
    args.put("x-dead-letter-exchange", exchange);

    return new Queue(manufacturingLabelRetryQueue, true, false, false, args);

}
```

La variable llamada “exchange” ya estaba definida en la implementación de base, y representa al único intercambio que había originalmente en Visual Tracking.

4) Generar los enlaces de estas dos colas a sus intercambios correspondientes. Se utiliza la anotación Qualifier para especificar qué Beans de los tipos Exchange y Queue deben extraerse en cada caso.

```java
/**
 * Binding of the dead-letter exchange and the dead-letter Manufacturing Label Service queue
 *
 * @param printerDeadLetterQueue the dead-letter printer queue. Must indicate the bean name of the queue
 * @param exchange the exchange to bind the queue
 * @return the exchange and queue binding
 */
@Bean
Binding deadLetterBinding(@Qualifier("manufacturingLabelDeadLetterQueue") Queue manufacturingLabelDeadLetterQueue,
        @Qualifier("deadLetterExchange") TopicExchange deadLetterExchange) {

    return BindingBuilder.bind(manufacturingLabelDeadLetterQueue).to(deadLetterExchange)
            .with(manufacturingLabelDeadLetteringRoutingKey);

}

/**
 * Binding of the retry exchange and the retrying Manufacturing Label service queue
 *
 * @param printerDeadLetterQueue the retrying printer queue. Must indicate the bean name of the queue
 * @param exchange the exchange to bind the queue
 * @return the exchange and queue binding
 */
@Bean
Binding retryBinding(@Qualifier("manufacturingLabelRetryQueue") Queue manufacturingLabelRetryQueue,
        @Qualifier("retryExchange") TopicExchange retryExchange) {

    return BindingBuilder.bind(manufacturingLabelRetryQueue).to(retryExchange)
            .with(manufacturingLabelDeadLetteringRoutingKey);

}
```

5) Se programa el Listener para la cola de Dead-Lettering. El método escribirá en el Logger informando del tipo de error que provocó que el mensaje llegara a la cola y hacia dónde se reenviará. Se comparará el número de reintentos de reenvío por los que ha pasado el mensaje con el límite establecido para la clase, enviándose al parking si éste ha sido superado, directamente a la cola de la que se descartó si los reintentos no fueron superados pero la "deathReason" fue su expiración, y a la cola Retry si no se cumple ninguna de las dos condiciones anteriores.

```
@RabbitListener(queues = "${rabbitmq.queue.deadLetter.manufacturing.label}")
public void PrinterServiceDeadLetterListener(Message message,
        @Header(name = "x-death") List<Map<String, Object>> deathHeader) {
    String deathReason = deathHeader.get(0).get("reason").toString();
    Boolean messageExpired = false;

    if (deathReason.equals("expired")) {
        LOGGER.warn("Message with key {} from Manufacturing Label Service expired on 'vt-manufacturing",
                message.getMessageProperties().getReceivedRoutingKey());
        messageExpired = true;
    }
    if (deathReason.equals("rejected")) {
        LOGGER.warn("Message with key {} from Manufacturing Label Service was rejected"
                + " by vt-manufacturing's consumer",
                message.getMessageProperties().getReceivedRoutingKey());
    }

    Long retries = (Long) deathHeader.get(0).get("count");

    if (retries > MAX_RETRIES_MANUFACTURING_LABEL) {
        LOGGER.info("Sending message with key {} to parking queue");
        template.convertAndSend(parkingExchange,
                message.getMessageProperties().getReceivedRoutingKey(), message);
    } else {
        if (messageExpired) {
            LOGGER.info("Resending message with key {} to 'vt-manufacturing'");
            template.convertAndSend(exchange,
                    message.getMessageProperties().getReceivedRoutingKey(), message);
        } else {
            LOGGER.info("Sending message with key {} to retry queue");
            template.convertAndSend(retryExchange,
                    message.getMessageProperties().getReceivedRoutingKey(), message);
        }
    }

}
```

10. VALIDACIÓN DEL DESARROLLO

En esta sección se detallarán las pruebas llevadas a cabo para comprobar que la solución implementada en EMI Suite funciona como se esperaba.

10.1. PRUEBA EN LOCAL DE LA SOLUCIÓN

Se creó un proyecto Java con el objetivo de comprobar que la funcionalidad era la esperada en todas las posibles situaciones que pueden darse en la política de reintentos propuesta.

Se compone de las siguientes clases, aparte de la main generada para Spring:

- **RabbitConfiguration**: Anotada con @Configuration. Contiene los Beans de las declaraciones de un rabbitTemplate (y los demás Beans que necesita para su funcionamiento) configurado de la misma manera que en Visual Tracking, de las cuatro colas que aparecen en el esquema de la figura 12, los cuatro intercambios correspondientes a los existentes en Visual Tracking y los enlaces entre estos y las

colas con la clave “clave” para todos menos para el parking, que tendrá de clave “*” para poder recibir todos los mensajes que no hayan podido enviarse a otra cola. En la configuración de RabbitTemplate, se emiten mensajes de control directamente por consola, en vez de usar un logger, para poder comprobar que el mensaje sigue el camino esperado en cada prueba.

```
@Bean
RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {

    RabbitTemplate template = new RabbitTemplate(connectionFactory);
    template.setMessageConverter(new Jackson2JsonMessageConverter());
    template.setMandatory(true);

    template.setReturnsCallback((returned) -> {
        if (returned.getExchange().equals("parkingExchange")) {
            System.out.println("La cola de parking no pudo obtener datos"
                    + " del mensaje con clave " + returned.getRoutingKey());

        } else {
            System.out.println("Mensaje al intercambio " +
                returned.getExchange() + " con clave " +
                returned.getRoutingKey() + " no pudo enviarse");
            template.convertAndSend("parkingExchange",
                returned.getRoutingKey(), returned);
        }

    });

    return template;
}

@Bean
TopicExchange normalExchange() {
    return new TopicExchange("normalExchange");
}

@Bean
TopicExchange deadLetterExchange() {
    return new TopicExchange("deadLetterExchange");
}

@Bean
TopicExchange retryExchange() {
    return new TopicExchange("retryExchange");
}

@Bean
TopicExchange parkingExchange() {
    return new TopicExchange("parkingExchange");
}
```

```
@Bean
Queue colaObjetivo() {
    Map<String, Object> args = new HashMap<>();
    args.put("x-message-ttl", 15000);
    args.put("x-dead-letter-exchange", "deadLetterExchange");
    return new Queue("colaObjetivo", true, false, false, args);
}

@Bean
Queue colaDeadLetter() {
    Map<String, Object> args = new HashMap<>();
    args.put("x-message-ttl", 15000);
    args.put("x-dead-letter-exchange", "parkingExchange");
    return new Queue("colaDeadLetter", true, false, false, args);
}

@Bean
Queue colaRetry() {
    Map<String, Object> args = new HashMap<>();
    args.put("x-message-ttl", 15000);
    args.put("x-dead-letter-exchange", "normalExchange");
    return new Queue("colaRetry", true, false, false, args);
}

@Bean
Queue colaParking() {
    Map<String, Object> args = new HashMap<>();
    return new Queue("colaParking", true, false, false, args);
}

 @Bean
 Binding normalBinding(@Qualifier("colaObjetivo") Queue colaObjetivo,
         @Qualifier("normalExchange") TopicExchange normalExchange) {
     return BindingBuilder.bind(colaObjetivo).to(normalExchange)
             .with("clave");

 }

 @Bean
 Binding deadLetterBinding(@Qualifier("colaDeadLetter") Queue colaDeadLetter,
         @Qualifier("deadLetterExchange") TopicExchange deadLetterExchange) {
     return BindingBuilder.bind(colaDeadLetter).to(deadLetterExchange)
             .with("clave");

 }

 @Bean
 Binding retryBinding(@Qualifier("colaRetry") Queue colaRetry,
         @Qualifier("retryExchange") TopicExchange retryExchange) {
     return BindingBuilder.bind(colaRetry).to(retryExchange)
             .with("clave");

 }

 @Bean
 Binding parkingBinding(@Qualifier("colaParking") Queue colaParking,
         @Qualifier("parkingExchange") TopicExchange parkingExchange) {
     return BindingBuilder.bind(colaParking).to(parkingExchange)
             .with("*");

 }
```

- **RabbitService**: Anotada con @Service para que se despliegue después del RabbitConfiguration. Contiene dos Listeners, uno para la cola objetivo, que lanzará un NotAcknowledgment al recibir un mensaje (para poder comprobar el funcionamiento del sistema de reintentos), y otra para la cola de Dead-Lettering que es igual al los Listener Dead-Lettering integradas en Visual Tracking.

```
@RabbitListener(queues = "colaObjetivo")
public void normalListener(Message message, Channel channel,
        @Header(AmqpHeaders.DELIVERY_TAG) long tag) throws IOException {
    System.out.println("Se rechaza el mensaje");
    channel.basicNack(tag, false, false);

}
```

```java
@RabbitListener(queues = "colaDeadLetter")
public void DeadLetterListener(Message message,
        @Header(name = "x-death") List<Map<String, Object>> deathHeader) {
    String deathReason = deathHeader.get(0).get("reason").toString();
    Boolean messageExpired = false;

    if (deathReason.equals("expired")) {
        System.out.println("El mensaje ha expirado");
        messageExpired = true;
    }
    if (deathReason.equals("rejected")) {
        System.out.println("El mensaje ha sido rechazado");
    }

    Long retries = (Long) deathHeader.get(0).get("count");

    if (retries > 2) {
        System.out.println("Reenviando a parking");
        template.convertAndSend("parkingExchange",
                message.getMessageProperties().getReceivedRoutingKey(),
                message);
    } else {
        if (messageExpired) {
            System.out.println("Reenviando a normalExchange");
            template.convertAndSend("normalExchange",
                    message.getMessageProperties().getReceivedRoutingKey(),
                    message);
        } else {
            System.out.println("Reenviando a retryExchange");
            template.convertAndSend("retryExchange",
                    message.getMessageProperties().getReceivedRoutingKey(),
                    message);
        }
    }
}
```

- **ClasePublicacion**: Anotada como @Component para que se construya una instancia de esta clase nada más iniciarse la aplicación. Tiene un método anotado con @PostConstruct, anotación que indica que el método debe ejecutarse cuando acabe de construirse su clase, que contiene un envío al intercambio normalExchange con la clave correspondiente.

```java
@Component
public class ClasePublicacion {

    private RabbitTemplate template;

    public ClasePublicacion(RabbitTemplate template) {
        this.template = template;
    }

    @PostConstruct
    public void iniciarProceso() throws InterruptedException {
        String mensaje = "contenido";
        template.convertAndSend("normalExchange","clave", mensaje);
    }

}
```

10.1.1. PRUEBA: RECHAZO

Se envía el mensaje siguiendo la estructura del código mostrada en el apartado anterior. Se comprueba que el mensaje llega a la cola de Dead-Lettering, desde donde es enviada a la de retry, y que esto se repite hasta que se supera el número de reintentos (establecido en 2), reenviándose entonces a la cola de parking, revisando los mensajes mostrados por consola.

10.1.2. PRUEBA: EXPIRACIÓN

Se elimina el Listener de la cola objetivo, para que el mensaje no sea consumido y su tiempo de vida se agote en la cola. Se comprueba que el mensaje llega a la cola de Dead-Lettering, desde donde es reenviada directamente al intercambio normalExchange, donde vuelve a expirar, y que esto se repite hasta que se supera el número de reintentos (establecido en 2), reenviándose entonces a la cola de parking, revisando los mensajes mostrados por consola.

10.1.3. PRUEBA: CLAVE INCORRECTA

Se envía el mensaje con una clave incorrecta, de manera que no pueda llegar a la cola objetivo. Se comprueba que el mensaje es reenviado a la cola de parking, revisando los mensajes mostrados por consola.

10.1.4. PRUEBA: PARKING NO ALCANZABLE

Se elimina el enlace de la cola de parking con su intercambio, de manera que, al superarse el número de reintentos, el mensaje no pueda almacenarse. Se comprueba que el mensaje es rechazado dos veces y que el posterior reenvío a cola de parking no puede realizarse con éxito, revisando los mensajes mostrados por consola.

10.2. VALIDACIÓN DE LA EMPRESA

Una vez realizadas las pruebas, se entrega en la empresa el producto. La opinión del responsable de desarrollo se recoge a continuación:

“Dada la solución propuesta sobre la nueva estructura para el flujo de la información en la comunicación entre las diferentes aplicaciones de EMI Suite, consideramos que se solucionan los problemas de pérdida de información que se tenía en el modelo anterior.

- La implementación de herramientas para el **re-encolado de mensajes** nos permite que no se pierda ningún mensaje en caso de algún error puntual, por ejemplo, de conexión, evitando así la pérdida de información.
- El control de **reintentos** en el re-encolado de mensajes evita que un mensaje que produzca un error no obstruya la cola, y permita el correcto flujo del resto de información.
- Finalmente, el almacenado de mensajes erróneos en una nueva cola de **descarte** nos permite tener siempre los mensajes que producen errores, con el fin de poder trazar e identificar las causas para así poder solucionarlos.

La implementación de esta estructura nos ha permitido pasar de ningún tipo de control de errores, con la consecuente deficiencia en la consistencia de la información y en la identificación de dichos errores, a un control automático para minimizar los errores puntuales

debidos a la comunicación y a poder tener almacenados los mensajes que provocaron errores, y así tener la información original, ya no sólo para trazar mejor el error, sino también para evitar la pérdida de dicha información en el sistema".

11. CONCLUSIONES Y POSIBLES MEJORAS

Cada vez más empresas necesitan digitalizar de sus procesos de negocio para ofrecer sus servicios, y los estándares de calidad que deben ofrecer estos sistemas y sus bases de datos cada vez son mayores. Por ello, se requieren plataformas robustas, tolerantes a fallos y que garanticen que no se produce pérdida de información.

En ese trabajo de fin de grado se han mostrado cómo, partiendo de una implementación parcial y con carencias, puede mejorarse el sistema de envío de mensajes de manera que pueda garantizarse la ausencia de esos errores.

Se ha elaborado primero una recopilación de los problemas a resolver de una plataforma, y una vez identificados, se ha planificado una solución que resuelva estos puntos de fallo críticos. La implementación de dicha solución en base a su esquema previamente desarrollado es sencilla de integrar, mecánica y fácilmente extendible a otros módulos del sistema.

De cara a futuro, se podría seguir avanzando en la mejora de la implementación de AMQP dentro de EMI SUITE, en los siguientes puntos:

- **Implementación de un confirmador de escucha**: Dentro de las clases Java de gestión de AMQP, existe una clase llamada ConfirmListener() que permite enviar confirmaciones a los productores de que los mensajes que han enviado han llegado con éxito a una cola. Aunque su integración podría ser de utilidad, no se ha incluido en la solución final porque se considera un factor secundario y poco relevante en comparación a la gestión de mensajes descartados mediante colas auxiliares.

- **Extensión de la solución a otras clases de Visual Tracking y a otros módulos**. El esquema de la solución sirve de manera general para todas las clases del sistema. Sería de interés modificar de manera gradual todo el código de la plataforma para terminar abandonando el uso básico que se hacía de Sockets, que no garantizaba entrega sin errores, y adoptar a nivel global estos mecanismos AMQP.

BIBLIOGRAFÍA

1. Macero García, M. (2017). *Learn Microservices with Spring Boot: A Practical Approach to RESTful Services using RabbitMQ, Eureka, Ribbon, Zuul, and Cucumber*. Apress.

2. Spring. (n.d.) *The IoC Container* [Consulta: 24 de noviembre de 2023] Disponible en: https://docs.spring.io/spring-framework/reference/core/beans.html

3. CloudAMQP (2019) *The RabbitMQ Management Interface* [Consulta: 12 de noviembre de 2024] Disponible en: https://www.cloudamqp.com/blog/part3-rabbitmq-for-beginners_the-management-interface.html

4. SpringAMQP. (n.d.) *Class RabbitTemplate*. [Consulta: 8 de diciembre de 2023]. Disponible en: https://docs.spring.io/spring-amqp/api/org/springframework/amqp/rabbit/core/RabbitTemplate.html