Orquestación distribuida con Camunda BPM, Parte 2

Simon Zambrovski
4 de noviembre · 7 min de lectura ★

Creo firmemente que la orquestación de Microservicios se convertirá en la próxima gran cosa a resolver. Al momento de escribir, varias soluciones intentan competir en esta área, principalmente construyendo sus propios lenguajes (textuales) específicos de dominio para describir la orquestación. En mi opinión, la orquestación debería expresarse en BPMN 2.x, ya que es un lenguaje bien adoptado, comprensible y maduro diseñado exactamente para este propósito.

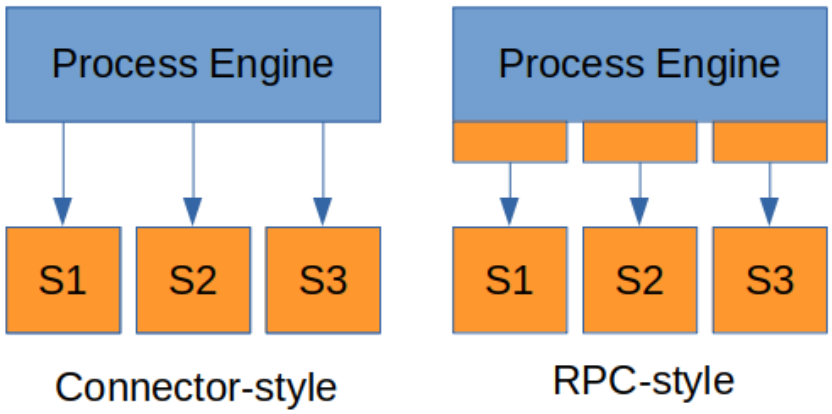
. . .

El término Orquestación en el contexto del microservicio puede ser ambiguo. Para aclararlo, me gustaría proponer la siguiente clasificación:

Orquestación tipo SOA

SOA se enfoca en la comunicación remota entre servicios, construida alrededor de las capacidades comerciales. El motor de proceso central llama de forma sincrónica a los servicios distribuidos de forma remota. La integración se realiza entre el motor del proceso de manejo de estado y el servicio sin estado.

> *Estoy simplificando un poco aquí y describiendo un "SOA mal diseñado / mal entendido", ya que, en esencia, SOA NO se trataba de servicios sin estado, pero a veces se implementó de esta manera.*



Orquestación sincrónica

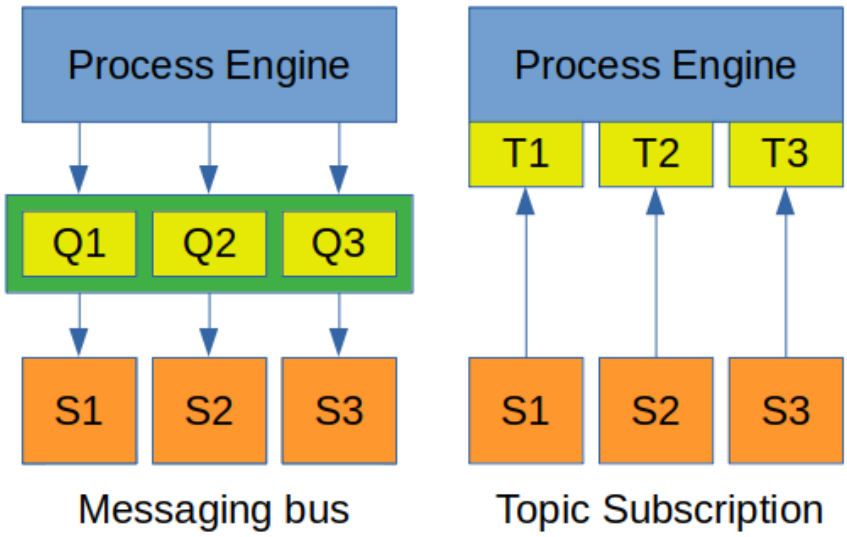
Hay dos estilos diferentes de implementación de esta clase de sistemas.

1. Se utiliza el patrón de integración del **conector** , si el motor de proceso está llamando al servicio (S1, S2, S3) utilizando el protocolo seleccionado directamente (generalmente HTTP).
2. Se utiliza el patrón de integración **RPC** , si el motor llama a un delegado local y estos invocan un servicio remoto (S1, S2, S3) a través del protocolo seleccionado (HTTP, Java RMI o cualquier otro protocolo síncrono).

En ambos casos, la integración requiere que el motor y los servicios estén en línea simultáneamente. El motor puede conocer la ubicación de los servicios o usar un registro o un corredor (recuerde el triángulo del servicio web) para resolver esto y los servicios usan una implementación orientada a la invocación para ejecutar el trabajo en nombre del motor de procesos.

Orquestación dirigida por mensajes

En lugar de invocación sincrónica, el motor central puede enviar mensajes a colas o temas y los servicios sin estado se suscriben a ellos. No se requiere la disponibilidad simultánea del motor y los servicios. Como resultado, los servicios utilizan una implementación orientada a la suscripción para ejecutar el trabajo en nombre del motor de procesos.



Orquestación asincrónica

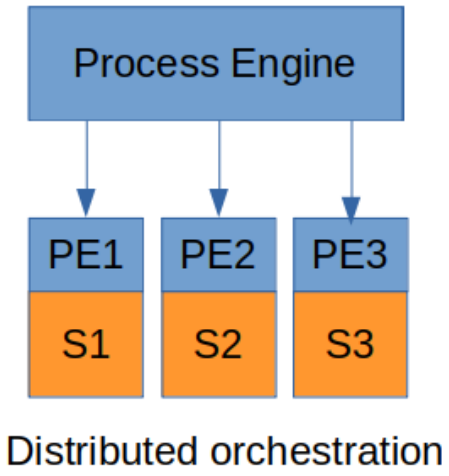
Hay dos tipos de implementación dependiendo de la abstracción de mensajería en uso:

1. La infraestructura de mensajería puede ser middleware (por ejemplo, utilizando un bus de mensajería central) que ofrece el concepto de **colas** (Q1, Q2, Q3). El motor envía mensajes asincrónicos a los servicios (S1, S2, S3) mediante colas.

2. En lugar de usar colas, el motor del proceso puede publicar la información en **temas** predefinidos (T1, T2, T3). La suscripción de temas puede ser parte del motor del proceso (también conocido como Patrón de tareas externas como se muestra arriba) o estar en el middleware de mensajería centralizado.

Orquestación distribuida

La orquestación en sí se distribuye. En lugar de la separación entre el motor lleno de estado y los servicios sin estado, los servicios se llenan de estado (y obtienen sus propios medios para manejar el estado, por ejemplo, utilizando la orquestación) y la integración se lleva a cabo entre los procesos empresariales (por ejemplo, la ejecución en los motores de proceso PE1, PE2, PE3)



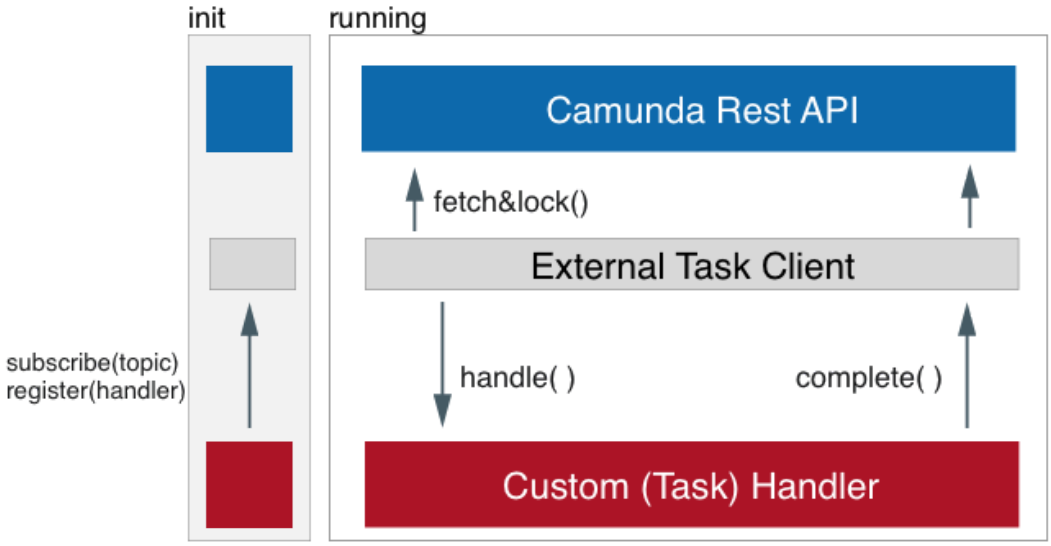
Orquestación distribuida entre motores de proceso.

Este estilo de orquestación se introdujo en el último artículo (ver Parte 1 de esta serie), en el que compartí mis pensamientos sobre los patrones de descomposición de la orquestación. En esta parte, me enfoco en más patrones y estrategias de implementación usando el Patrón de tareas externas.

. . .

Patrón de tareas externas

El patrón de tareas externas ha sido introducido por Camunda BPM en la versión 7.4 y es una de las características más importantes para romper con el monolito de flujo de trabajo hacia el flujo de trabajo distribuido. Originalmente, está destinado a proporcionar una implementación de tareas de servicio orientada a la suscripción en contraste con la orientada a la invocación. Es decir, si el motor ejecuta una tarea de servicio, no está llamando a un delegado para llamar a un servicio (remoto), sino que crea un registro de tarea externo y espera a que un trabajador de tareas externo (remoto) lo busque y ejecute.



Tarea externa de Camunda

El patrón de tareas externas tiene varias propiedades importantes:

- La tarea externa *siempre* es *un estado de espera* en el motor. El motor de proceso confirmará la transacción después de crear un registro de tarea externa.
- Se *revierte* la dirección de la comunicación. El motor de procesos no está llamando al servicio, pero el trabajador de tareas externo (ubicado junto con el servicio) está llamando al motor de procesos.
- Costumbre Handler recibe una notificación, si la tarea externa está disponible y se puede utilizar la `ExternalTaskService` API (`complete()` , `handleFailure()` y `handleBPMNError()`) para informar motor de procesos sobre el resultado.
- El significado del tiempo de espera de `fetchAndLock` es reservar la ejecución para un trabajador de tareas durante cierto tiempo. Si el tiempo de espera se produce antes de que se transmita la respuesta, la tarea vuelve a estar disponible para otros manejadores de tareas.

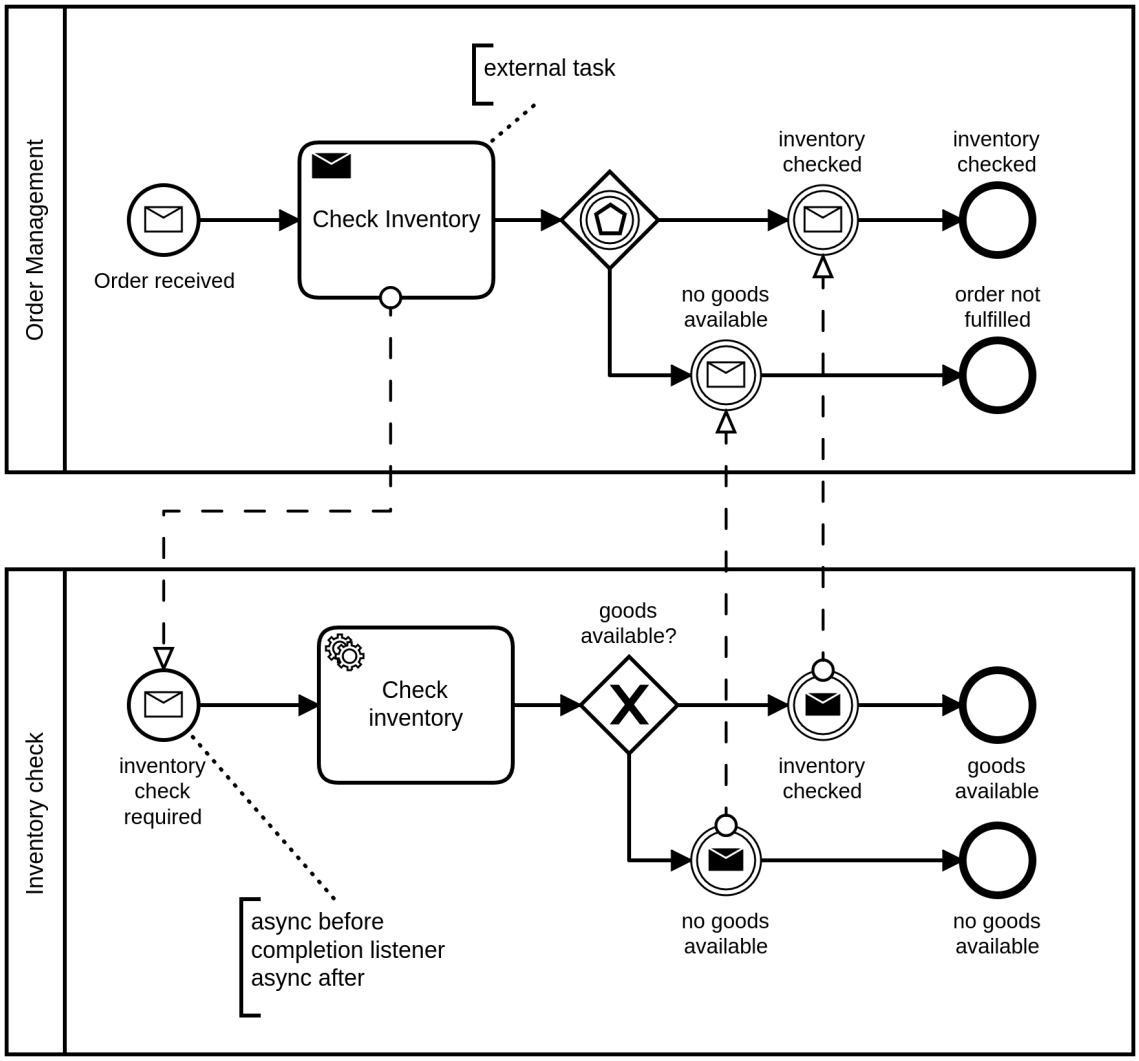
Se requiere un cliente de tareas externo para realizar trabajos sobre temas específicos. Camunda proporciona implementaciones propias en Java y JavaScript , que pueden usarse como una biblioteca.

. . .

Uso de tareas externas en orquestación distribuida

Un enfoque interesante es usar el patrón de tareas externas en la orquestación distribuida. ¿Recuerdas el ejemplo de la Parte 1 de esta serie? Tratemos de modelarlo usando un patrón de tarea externo.

Usando el patrón de tareas externas, propongo la siguiente implementación:



Permítanme explicar cómo puede funcionar esta implementación. Hay algunos requisitos para el `Inventory check` proceso para trabajar.

1. La `Check Inventory` tarea de envío de la Gestión de pedidos tiene una `External Task` implementación.
2. La Gestión de pedidos NO tiene estados de espera entre la tarea externa y los eventos de captura de mensajes.
3. El componente Comprobación de inventario proporciona un trabajador de tareas externas, que se conecta al motor del proceso de Gestión de pedidos e inicia el proceso de inventario.
4. El inicio del proceso de inventario es IDEMPOTENTE. Es decir, si el proceso ya se está ejecutando, el trabajador de tareas externo no debe iniciar uno nuevo, sino simplemente no hacer nada.
5. El trabajador de tareas externo solo obtiene la tarea externa de Gestión de pedidos, pero NO la está completando.

6. La finalización de la tarea externa la realiza un escucha de finalización, adjunta como un escucha de ejecución al final del evento de inicio del mensaje `Inventory check required`. Esta finalización es sincrónica. Dado que la identificación de la tarea externa es necesaria para su finalización, puede almacenarse en una variable de proceso del proceso de verificación de inventario.

7. El inicio del `Inventory check` proceso DEBE marcarse como `async before`. Si la finalización de la tarea externa falla, el proceso sigue ejecutándose, por lo que el ejecutor del trabajo `Inventory check` volverá a intentar la finalización.

8. El inicio del `Inventory check` proceso DEBE marcarse como `async after`. Si falla alguna actividad después del inicio, la transacción no se revertirá antes de la escucha de finalización (porque esto ya ha completado la tarea externa).

Permítanme explicar por qué la tarea externa debe ser completada por el oyente de finalización, en lugar del trabajador de tareas externo y cómo esto resuelve el problema de concurrencia. La finalización de la tarea externa por parte del oyente es una llamada sincrónica al `Order Management` motor del proceso, ejecutada, por ejemplo, a través de REST. Esta llamada inicia una nueva transacción en el `Order Management` motor de procesos, que completa la tarea externa y escribe las suscripciones de mensajes para los mensajes `inventory checked` y `no goods available` ANTES de que la respuesta se envíe nuevamente al `Inventory check` motor de procesos. Debido a este hecho, no hay condición de carrera entre dos motores y `Order Management` está esperando el mensaje antes de que el `Inventory check` motor de proceso pueda enviarlo .

Para implementar este comportamiento, necesitamos un Cliente de tareas externas modificado, capaz de recuperar y completar por separado una tarea externa. La biblioteca del Cliente de tareas externas proporcionada por Camunda no es capaz de hacer esto y se requiere una pequeña modificación del cliente existente (y el generador de clientes). Aquí está el código:

```
paquete  io.holunda.spike.external.task.client ;

import  org.camunda.bpm.client.ExternalTaskClient ;
import  org.camunda.bpm.client.impl.ExternalTaskClientBuilderImpl ;
import  org.camunda.bpm.client.task.ExternalTaskService ;
import  org.camunda.bpm.client.task.impl.ExternalTaskServiceImpl ;

/ **
 * Generador de sobrescritura para poder configurar cosas relacionadas con el cliente.
 * /
 clase  pública MyExternalTaskClientBuilder  extiende  ExternalTaskClientBuilderImpl {

  / **
   * Crea una instancia de trabajo.
   * /
  public  static  MyExternalTaskClientBuilder  create () {
    devolver  nuevo  MyExternalTaskClientBuilder ();
  }
```

```
  / **
   * Recupera el contenedor de tareas externo, que contiene el cliente de tareas externas y el s
   * /
  public  ExternalTaskClientWrapper  buildClientWrapper () {
    / *
     * Esto crea el cliente externo e inicializa el cliente del motor
     * /
     Cliente final ExternalTaskClient =  super . construir();
    / *
     * Está bien crear nuestra propia instancia de servicio de tareas externo, ya que no tiene e
     * y solo proporciona métodos para operar en tareas, que delegan al cliente del motor.
     * /
    última  ExternalTaskService servicio =  nueva  ExternalTaskServiceImpl (engineClient);
    / *
     * Devolver instancia envuelta.
     * /
    devolver  nuevo  ExternalTaskClientWrapper (cliente, servicio);
  }
}
```

Desarrollador extensivo para suministrar al cliente y al servicio.

Usando el generador anterior, es posible crear un par que consiste en el Cliente de tareas externas y el Servicio de tareas externas del lado del cliente (el devuelto `ExternalTaskClientWrapper` es solo un POJO que contiene dos referencias).

Para la creación de instancias del constructor, se puede usar el siguiente código:

```
paquete  io.holunda.spike.external.task.client ;

import  lombok.val ;
import  org.springframework.context.annotation.Bean ;
importar  org.springframework.context.annotation.Configuration ;

/ **
 * Configuración.
 * /
@Configuración
 clase  pública MyConfiguration {

  / **
   * Crea el contenedor del cliente de tareas con el cliente configurado.
   * @return task client wrapper.
   * /
  @Frijol
  public  ExternalTaskClientWrapper  externalTaskClient () {

    MyExternalTaskClientBuilder builder =  MyExternalTaskClientBuilder . crear();

    constructor
      .workerId ( " my-worker-id " )
      .baseUrl ( " http: // localhost: 8080 / rest / " )
      .asyncResponseTimeout ( 80 000 );
```

```
26
27      Constructor de retorno . buildClientWrapper ();
28    }
29
30  }
```

MyConfiguration.java procesar alojado con ♡ por GitHub sinver

Instanciación del constructor extendido

Ahora imagine al Trabajador de tareas externo, responsable de iniciar el proceso de Verificación de inventario. Consiste en un controlador de tareas externo y el escucha de finalización (ambos encapsulados por el mismo componente Spring):

```
1   paquete  io.holunda.spike.external.task.client ;
2
3   import  org.camunda.bpm.client.task.impl.ExternalTaskImpl ;
4   import  org.camunda.bpm.engine.RuntimeService ;
5   import  org.camunda.bpm.engine.delegate.ExecutionListener ;
6   import  org.camunda.bpm.engine.runtime.ProcessInstance ;
7   import  org.camunda.bpm.engine.variable.VariableMap ;
8   importar  org.springframework.scheduling.annotation.Scheduled ;
9   import  org.springframework.stereotype.Component ;
10
11  import  java.util.List ;
12
13  importar estática  org.camunda.bpm.engine.variable.Variables.putValue ;
14
15  @Componente
16  public  class  InventoryCheckWorker {
17
18    privada  estática  última  cadena  EXTERNAL_TASK_ID  =  " EXTERNAL_TASK_ID " ;
19     Cadena estática  privada final  MENSAJE = " Inventory_check_needed " ;
20
21    privada  última  ExternalTaskClientWrapper envoltorio;
22     RuntimeService final  privado runtimeService;
23
24    public  InventoryCheckWorker ( externalTaskClientWrapper  wrapper , RuntimeService  runtimeSer
25      esta . envoltorio = envoltorio;
26      esta . runtimeService = runtimeService;
27    }
28
29    / **
30     * Trabajador de instalación
31     * /
32    @Scheduled ( initialDelay  =  60_000 , fixedDelay  =  Long . MAX_VALUE )
33    public  void  registerWorker () {
34      envoltura
35        .getExternalTaskClient ()
36        .subscribe ( " tema " )
37        .lockDuration ( 60_000 )
38        .entrenador de animales(
39          (externalTask, externalTaskService) - > {
40            Cadena businessKey = externalTask . getBusinessKey ();
41            // verifica si ya comenzó
42            List < ProcessInstance > running = runtimeService
```

```
            .createProcessInstanceQuery ()
            .processInstanceBusinessKey (businessKey)
            .lista();
          // sé idempotente, comienza solo si aún no lo has hecho
          if (ejecutando . isEmpty ()) {
            VariableMap variables = putValue ( EXTERNAL_TASK_ID , externalTask . GetId ());
            runtimeService
              .startProcessInstanceByMessage (
                MENSAJE ,
                businessKey,
                variables
              );
          }
        }
      )
      .abierto();
  }


  / **
   * Finalización oyente.
   * *
   * @return un escucha de ejecución que completa la tarea externa que inició el proceso.
   * /
  public  ExecutionListener  RecognizeStarted () {
    ejecución de devolución - > {

      String externalTaskId = ( String ) ejecución . getVariable ( EXTERNAL_TASK_ID );
      ExternalTaskImpl task =  new  ExternalTaskImpl ();
      tarea . setId (externalTaskId);

      esta . envoltorio . getExternalTaskService () . completa (tarea);
    };
  }
}
```

Tenga en cuenta que el controlador solo está recuperando la tarea, inicia el proceso si aún no se está ejecutando y almacena la identificación de la tarea externa en una variable de proceso (pero NO la está completando). El oyente de finalización es responsable de leer la identificación de la tarea externa de la variable y completar la tarea externa.

. . .

Conclusión

Este artículo propone una clasificación de la orquestación basada en las propiedades del patrón de comunicación. Me concentro en la orquestación distribuida, como ya se discutió en la Parte 1 de esta serie. Dado que este patrón no es fácil de implementar si

está intentando un enfoque ingenuo, le propongo usar el Patrón de tareas externas y el Cliente de tareas externas Camunda ligeramente modificado (para Java). Al hacerlo, obtenemos lo mejor de ambos mundos:

- El modelo BPMN expresa exactamente lo que se pretende, permanece limpio y no está contaminado con ninguna solución de marco / implementación.
- No se requiere la disponibilidad simultánea de componentes (si se trata de un problema real, también existe la posibilidad de reemplazar la entrega del mensaje del `Inventory check` proceso por otro Trabajador de tareas externo a costa del modelo de proceso más contaminado)
- La dirección de comunicación se revierte evitando la construcción del monolito distribuido en función de múltiples servicios.

Microservicios Orquestación Camunda Bpm Patrón de tareas externas