



Invocación de servicios desde el proceso

Acceda a la lógica empresarial implementada para la VM de Java, así como a los servicios remotos, mediante pequeñas clases "JavaDelegate". Este código de pegamento asigna la entrada / salida del proceso a la lógica de su negocio por medio de las mejores bibliotecas de su elección. Algunos casos de uso requieren un enfoque de extracción, donde los hilos de trabajo externos consultan a Camunda para "tareas de servicio externo". En caso de que necesite / desee definir definiciones de proceso BPMN autónomas, puede aprovechar las secuencias de comandos o expresiones para pequeñas piezas de lógica y "conectores" para llamar a servicios web complejos / remotos.

Invocación de servicios desde el proceso ¿

Entendiendo **Push and Pull**

Selección del **enfoque de implementación**

Comprensión y uso de **delegados de Java**

Comprensión y uso de **tareas externas**

Manejo de **problemas y excepciones**

Prima 📦 Soluciones tecnológicas de ejemplo

Llamar a los servicios web **SOAP**

Llamar a los servicios web **REST**

Envío de mensajes **JMS**

Usar **SQL** para acceder a la base de datos

Llamar a sistemas **SAP**

Ejecutando un script **Groovy**

? best-practices@camunda.com

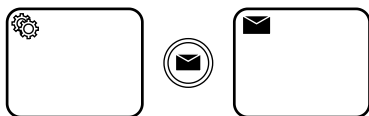
Entendiendo Push and Pull

El patrón típico al invocar servicios es **push**, lo que significa que el motor del proceso emite activamente una **llamada de servicio** (o ejecuta un **script**) a través de los mecanismos que se describen a continuación. Pero algunos casos de uso requieren un enfoque de **extracción**, donde los subprocesos de trabajadores externos consultan la API del motor de procesos para **tareas de servicio externo**. Luego hacen el trabajo real y notifican al motor del proceso la finalización de los trabajos.

Selección del enfoque de implementación

Camunda ofrece varias posibilidades. Por supuesto, puede mezclar esos enfoques seleccionando diferentes técnicas de implementación para diferentes tareas de servicio.

	Delegado de Java		Expresión	Conector	Tarea externa	Tarea de script
	Judía nombrada	Clase de Java				

	Llame a una clase bean o java con nombre que implemente la JavaDelegate interfaz.		Evaluar una expresión usando JUEL.	Use un conector configurable (los servicios REST o SOAP se proporcionan listos para usar).	Extraiga una tarea de servicio en un subproceso de trabajo externo e informe al motor del proceso de finalización.	Ejecute un script dentro del motor.
Usar con elementos BPMN						
Dirección de comunicación	Empuje el elemento de trabajo emitiendo una llamada de servicio				Extraer tarea del subproceso de trabajo	Empuje el elemento de trabajo ejecutando un script
Tecnología	Utilice su marco preferido, por ejemplo, un cliente JAX-WS para llamar a los servicios web SOAP			Utilice el conector REST / SOAP y la plantilla de mensaje	Use Camunda External Task Client o REST API para consultar por trabajo	Utilice el motor de secuencias de comandos compatible con JSR-223
Implementar vía	Java (en la misma JVM)		Lenguaje de expresión (puede hacer referencia al código Java)	Configuración BPMN	Configuración BPMN y lógica de extracción externa	Por ejemplo, Groovy, JavaScript, JRuby o Jython
Finalización de código y refactorización	✓	✓	Tal vez		✓	Depende del idioma / IDE
Comprobaciones del compilador	✓	✓			✓	Depende del idioma / IDE
Inyección de dependencia	✓ (cuando se usa Spring, CDI, ...)		✓ (cuando se usa Spring, CDI, ...)			
Fuerzas en las pruebas	Registrar simulacros en lugar de frijoles originales	Simulacros de lógica de negocios dentro de JavaDelegate	Registrar simulacros en lugar de frijoles originales	Difícil por falta de inyección de dependencia	Fácil, ya que el servicio no se llama activamente	Considere recursos de script externos
Configurar a través de	Atributo BPMN serviceTask camunda: delegate Expression	Atributo BPMN serviceTask camunda: class	Atributo BPMN serviceTask camunda: expression	BPMN Ext. Elemento + serviceTask camunda: connector	Atributos BPMN serviceTask camunda: type= 'external' and `camunda:topic	Elemento BPMN script o atributo BPMN scriptTask camunda: resource
Tolerancia a fallas y reintentos	Manejado por estrategias de reintento de Camunda y gestión de incidentes				Bloquee tareas por un tiempo definido. Utilice el reintento de Camunda y la gestión de incidentes	Manejado por estrategias de reintento de Camunda y gestión de incidentes

Escalado (tener múltiples subprocesos de trabajo)	A través del equilibrador de carga frente al servicio			Se pueden iniciar múltiples hilos de trabajo	A través de la configuración del ejecutor de trabajos
Regulación (p. Ej., "Una solicitud a la vez")	No es posible fuera de la caja, requiere que se implemente una lógica de aceleración propia			Inicie / detenga exactamente la cantidad de hilos de trabajo que necesita	No es posible fuera de la caja
Tareas Reutilizables	Usar inyección de campo [1]	Utilice los parámetros del método	Construye tu propio conector [2]	Reutilice temas de tareas externas [3] y configure el servicio a través de variables	
Usar cuando	Siempre si no hay razón en contra	Definir pequeñas piezas de lógica directamente en BPMN	Definir un proceso BPMN autónomo sin código Java	Ver casos de uso a continuación	Definición de procesos BPMN sin código Java.
	Aprende más [4]	Aprende más [5]	Aprende más [6]	Aprende más [7]	Aprende más [8]

Comprensión y uso de delegados de Java

Un delegado de Java es una clase Java simple que implementa la `JavaDelegate` interfaz Camunda . Le permite usar **la inyección de dependencia** siempre que esté construida como un CDI o Spring Bean y conectada a su BPMN a `serviceTask` través del `camunda:delegateExpression` atributo:

```
<serviceTask id="service_task_publish_on_twitter" camunda:delegateExpression="#{tweetPublicationDelegator}" name="Publish on Twitter">
</serviceTask>
```

XML

Aproveche la inyección de dependencia para obtener acceso a los beans de **servicio empresarial** del delegado. Considere que un delegado es una parte semántica de la definición del proceso en un sentido más amplio: se ocupa de los detalles necesarios para conectar la lógica empresarial a su proceso. Por lo general, hace lo siguiente:

1. Asignación de entrada de datos
2. Llamar a un método en el servicio comercial
3. Mapeo de salida de datos

Evite programar la lógica de negocios en delegados de Java , separe esta lógica llamando a una de sus propias clases como un "servicio de negocios":

```

@Named
public class TweetPublicationDelegate implements JavaDelegate {

    @Inject
    private TweetPublicationService tweetPublicationService;

    public void execute(DelegateExecution execution) throws Exception {
        String tweet = new TwitterDemoProcessVariables(execution).getTweet(); ①
        // ...
        try {
            tweetPublicationService.tweet(tweet); ②
        } catch (DuplicateTweetException e) {
            throw new BpmnError("duplicateMessage"); ③
        }
    }
}
// ...

```

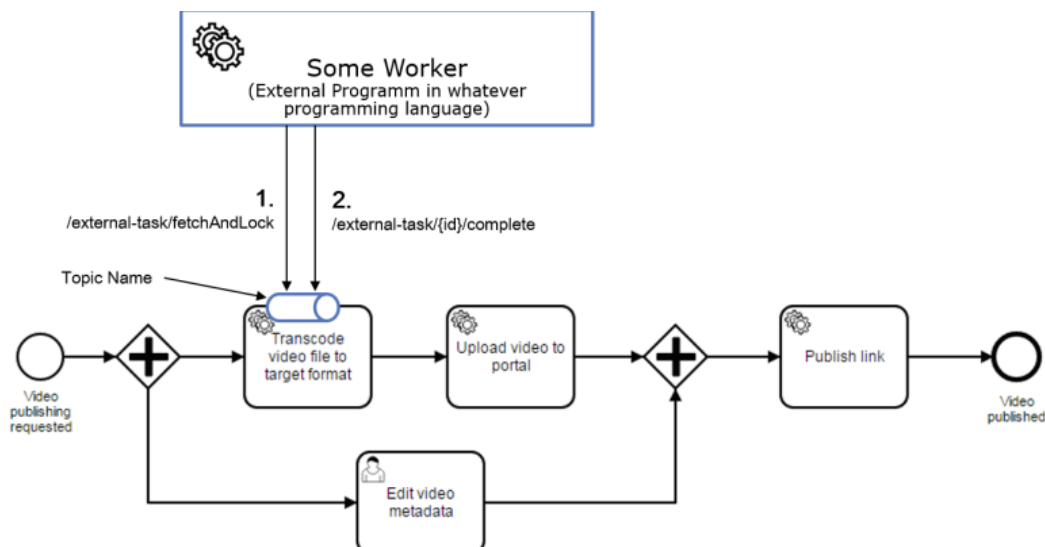
- Recuperar el valor de esta variable de proceso pertenece a lo que llamamos el **mapeo de entrada** del código delegado y, por lo tanto, se considera parte de la definición más amplia del proceso. (Para obtener una explicación de la clase de acceso variable utilizada aquí, consulte [Manejo de datos en procesos](#))
- ② Este método ejecuta **la lógica empresarial** independiente del motor de proceso , por lo tanto, ya no forma parte de la definición más amplia del proceso y se coloca en un bean de servicio empresarial separado.
- Esta excepción es específica del motor de proceso y, por lo tanto, generalmente no se produce por el método de servicio de su empresa. Es parte del **mapeo de salida** que necesitamos traducir la excepción comercial a la excepción necesaria para impulsar el proceso; nuevamente, el código es parte de la definición de proceso "más amplia" y debe implementarse en el Delegado de Java.
- ③



En caso de que desee crear Delegados Java que sean **reutilizables** para otras definiciones de proceso, aproveche la **inyección de campo** [1] para pasar la configuración de la definición de proceso BPMN a su Delegado Java.

Comprensión y uso de tareas externas

Una **tarea externa** es una tarea que espera ser completada por algún trabajador de servicio externo sin llamar explícitamente a ese servicio. Se configura declarando un llamado **tema** (que caracteriza el tipo de servicio). La API de Camunda debe sondearse para recuperar tareas externas abiertas para un tema de un determinado servicio y debe ser informada sobre la finalización de una tarea:



La **interacción con la API de tareas externas** se puede realizar de dos maneras diferentes:

- Use [las bibliotecas de cliente de tareas externas](#) de Camunda [9] para [Java](#) [10] y [Node.js](#) [11]: introducidas con Camunda 7.9, estas bibliotecas hacen que sea muy fácil implementar su trabajador de tareas externo personalizado.
- Cree su propio cliente de tareas externas basado en la API de REST o Java de Camunda: este enfoque es particularmente útil para cada idioma que aún no es compatible con una biblioteca de cliente oficial de Camunda. Puede encontrar un [ejemplo para .NET en GitHub](#) [12].

Las tareas externas admiten los siguientes **casos de uso**:

1. **Desacoplamiento temporal**: el patrón puede reemplazar una cola de mensajes entre la tarea de servicio (el "consumidor") y la implementación del servicio (el "proveedor"). Puede eliminar la necesidad de operar un bus de mensajes dedicado mientras mantiene el desacoplamiento que proporcionaría la mensajería.
2. **Arquitecturas de Polyglott**: el patrón se puede utilizar para integrar, por ejemplo, servicios basados en .NET, cuando podría no ser tan fácil escribir delegados de Java para llamarlos. Las implementaciones de servicios son posibles en cualquier idioma que pueda usarse para interactuar con una API REST.
3. **Mejor escala**: el patrón le permite iniciar y detener a los trabajadores como desee, y ejecutar tantos como necesite. Al hacerlo, puede escalar cada tarea de servicio (o para ser precisos: cada "tema") individualmente.
4. Conecte **Cloud BPM** a los **servicios locales**: el patrón lo ayuda a ejecutar Camunda en algún lugar de la nube (como suelen hacer nuestros clientes), porque aún puede tener servicios en las instalaciones, ya que ahora pueden consultar su trabajo a través de REST a través de SSL, que También es bastante amigable con el firewall.
5. **Evite los tiempos de espera**: el patrón le permite llamar asincrónicamente a los servicios de larga ejecución, que eventualmente se bloquean durante horas (y, por lo tanto, causarían tiempos de espera de transacciones y conexiones cuando se los llama sincrónicamente).
6. **Ejecutar servicios en hardware especializado**: cada trabajador puede ejecutarse en el entorno que mejor se adapte a la tarea específica de ese trabajador, por ejemplo, instancias de nube optimizadas para CPU para procesamiento de imágenes complejas e instancias optimizadas para memoria para otras tareas.



Obtenga más información sobre las tareas externas en la [guía](#) del [usuario](#) [3], así como la [referencia](#) [13] y explore el ejemplo de procesamiento de video que se muestra arriba en mayor detalle leyendo la [publicación](#) del [blog al](#) [14] respecto.

Manejo de problemas y excepciones

Al invocar servicios, puede experimentar fallas y excepciones. Consulte nuestra guía de mejores prácticas por separado sobre cómo [abordar problemas y excepciones](#).

Prima 🎁 Soluciones tecnológicas de ejemplo

Llamar a los servicios web SOAP

Cuando necesite llamar a un **servicio web SOAP**, normalmente se le dará acceso a una descripción del servicio basada en WSDL legible por máquina. Luego puede usar [JAX-WS](#) [15] y, por ejemplo, la [generación de cliente JAX-WS de Apache CXF](#) [16] para **generar un Cliente de servicio web Java** haciendo uso de un complemento Maven. Se puede llamar a ese cliente desde su `JavaDelegate`.

Encuentre un ejemplo completo que use la generación de clientes JAX-WS en el [repositorio de ejemplos de Camunda BPM](#) [17].



Por lo general, preferimos la generación del código del cliente sobre el uso del [conector Camunda SOAP](#) [6], debido al mejor soporte de IDE para hacer la asignación de datos mediante el uso de la finalización del código. También puede aprovechar los enfoques de prueba estándar y los cambios en el WSDL reactivarán la generación de código y su compilador verificará cualquier problema que surja de una interfaz cambiada. Sin embargo, si necesita un XML BPMN 'autónomo' sin ningún código Java adicional, el conector podría ser el camino a seguir. Ver [ejemplo de conector SOAP](#) [18].

Llamar a los servicios web REST

Si necesita llamar a un **servicio web REST**, generalmente se le dará acceso a una documentación del servicio legible por humanos. Puede utilizar bibliotecas de cliente REST Java estándar como [RestEasy](#) [19] o [JAX-RS](#) [20] para **escribir un cliente de servicio REST Java** que se pueda llamar desde un `JavaDelegate`.



Por lo general, preferimos escribir clientes Java sobre [Camunda REST Connector](#) [6], debido a la mejor compatibilidad con IDE para hacer la asignación de datos mediante el uso de la finalización de código. De esta manera, también puede aprovechar los enfoques de prueba estándar. Sin embargo, si necesita un XML BPMN 'autónomo' sin ningún código Java adicional, el conector podría ser el camino a seguir. Consulte el [ejemplo del conector REST](#) [21].

Envío de mensajes JMS

Cuando necesite enviar un **mensaje JMS**, use un cliente Java simple e invoque desde una tarea de servicio en su proceso, por ejemplo, usando un delegado de Camunda Java:

```

@Named("jmsSender")
public class SendJmsMessageDelegate implements JavaDelegate {

    @Resource(mappedName = "java:/queue/order")
    private Queue queue;

    @Resource(mappedName = "java:/JmsXA")
    private QueueConnectionFactory connectionFactory;

    public void execute(DelegateExecution execution) throws Exception {
        String correlationId = UUID.randomUUID().toString(); ①
        execution.setVariable("jmsCorrelationId", correlationId);

        Connection connection = connectionFactory.createConnection(); ②
        Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(queue);

        TextMessage message = session.createTextMessage( ③
            "someOwnContent, e.g. Tweet Object Data, plus " + correlationId); ④
        producer.send(message);

        producer.close();
        session.close();
        connection.close();
    }
}

```

Considere qué información puede usar para correlacionar una respuesta asincrónica a su instancia de proceso.

- ① Por lo general, preferimos un UUID artificial generado para la comunicación, que el proceso de espera también deberá recordar.
- ② Deberá abrir y cerrar conexiones JMS, sesiones y productores. Tenga en cuenta que este ejemplo solo sirve para comenzar. En la vida real, deberá decidir qué conexiones necesita abrir y, por supuesto, cerrar correctamente.
- ③ Deberá crear y enviar su mensaje específico.
- ④ Agregue datos comerciales relevantes a su mensaje junto con información de correlación.



Este ejemplo solo sirve para comenzar. En la vida real, considere si necesita encapsular el cliente JMS en una clase separada y simplemente conectarlo desde el delegado de Java. También decida qué conexiones necesita abrir y cerrar correctamente en qué puntos peristálticos.

En GitHub, puede encontrar un ejemplo más completo para la [mensajería asíncrona con jms](#) [22] .

Usar **SQL** para acceder a la base de datos

Utilice JDBC simple si tiene requisitos simples. Invoque su declaración SQL de una tarea de servicio en su proceso, por ejemplo, utilizando un delegado de Camunda Java:

```

@Named("simpleSqlDelegate")
public class simpleSqlDelegate implements JavaDelegate {

    @Resource(name="customerDB")
    private javax.sql.DataSource customerDB;

    public void execute(DelegateExecution execution) throws Exception {
        Statement statement = null;
        Connection connection = null;

        try {
            connection = customerDB.getConnection();
            String query = "SELECT name " + ❶
                "FROM customer " +
                "WHERE id = ?";
            statement = connection.createStatement();
            statement.setString(1, execution.getProcessBusinessKey()); ❷
            ResultSet resultSet = stmt.executeQuery(query);
            if (resultSet.next()) {
                execution.setVariable("customerName", resultSet.getString("name")); ❸
            }
        } finally {
            if (statement != null) statement.close();
            if (connection != null) connection.close();
        }
    }
}

```

- ❶ Por supuesto, deberá definir su declaración SQL. Considere usar declaraciones preparadas, si desea ejecutar un objeto de declaración muchas veces.
- ❷ Por lo general, necesitará introducir parámetros en su consulta SQL que ya se conocen durante la ejecución de la instancia del proceso ...
- ❸ ... y devolver un resultado potencial que tal vez sea necesario más adelante en el proceso.



Este ejemplo solo sirve para comenzar. Para la vida real, considere si necesita encapsular el código JDBC en una clase separada y simplemente conectarlo desde el Delegado de Java. También decida qué conexiones necesita abrir y cerrar correctamente en qué punto.

Tenga en cuenta que el motor del proceso Camunda habrá abierto una transacción de base de datos para sus propios fines de persistencia al llamar al Delegado de Java que se muestra arriba. Tendrá que tomar una decisión consciente si desea unirse a esa transacción (y configurar su **gestión de TX en consecuencia**) o no.



En lugar de invocar SQL directamente, considere usar [JPA](#) [23] si tiene requisitos más complejos. Sus técnicas de mapeo de objetos / relacionales le permitirán vincular tablas de bases de datos a objetos Java y abstraer de proveedores de bases de datos específicos y sus dialectos SQL específicos.

Llamar a sistemas **SAP**

Para llamar a un sistema **SAP** , tiene las siguientes opciones:

- Use llamadas de cliente REST o SOAP, conectando Camunda a **SAP Netweaver Gateway** o **SAP Enterprise Services**.
- Utilice **los conectores Java de SAP (JCo)**. Quizás considere usar el Open Source Frameworks [Hibersap](#) [24] y el [Cuckoo Resource Adapter](#) [25] para usar esos conectores Java cómodamente.



El artículo alemán [Alternativen für die Integration von SAP-Systemen mit Java EE](#) [26] escrito por Carsten Erker y Torsten Fink se publicó en JavaSPEKTRUM 6/2014 y analiza varias opciones con gran detalle. Los autores también publicaron una [aplicación de ejemplo para Hibersap y Cuckoo](#) [27] en GitHub.

Ejecutando un script **Groovy**

Una tarea de script ...



... se define especificando el script y el scriptFormat.

```
<scriptTask id='theScriptTask' scriptFormat='groovy' camunda:resultVariable="size">
  <script>anArray.size()</script>
</scriptTask>
```

XML

Para obtener un código más extenso (que también debe probarse por separado), considere utilizar secuencias de comandos externas a su archivo BPMN y haga referencia a ellas con un `camunda:resource` atributo en `scriptTask`.

Obtenga más información sobre las muchas formas en que los scripts pueden usarse con Camunda en la [Guía del usuario](#) [8].

Links

- [1] <https://docs.camunda.org/manual/7.11/user-guide/process-engine/delegation-code/#field-injection>
- [2] <https://docs.camunda.org/manual/7.11/reference/connect/>
- [3] <https://docs.camunda.org/manual/7.11/user-guide/process-engine/external-tasks/>
- [4] <https://docs.camunda.org/manual/7.11/user-guide/process-engine/delegation-code/#java-delegate>
- [5] <https://docs.camunda.org/manual/7.11/user-guide/process-engine/expression-language/#delegation-code>
- [6] <https://docs.camunda.org/manual/7.11/user-guide/process-engine/connectors/>
- [7] <https://docs.camunda.org/manual/7.11/reference/rest/external-task/>
- [8] <https://docs.camunda.org/manual/7.11/user-guide/process-engine/scripting/>
- [9] <https://docs.camunda.org/manual/7.11/user-guide/ext-client/>
- [10] <https://github.com/camunda/camunda-external-task-client-java>
- [11] <https://github.com/camunda/camunda-external-task-client-js>
- [12] <https://github.com/berndruecker/camunda-dot-net-showcase/tree/master/CamundaClient>
- [13] <https://docs.camunda.org/manual/7.11/reference/bpmn20/tasks/service-task/#external-tasks>
- [14] <https://blog.camunda.org/post/2015/11/external-tasks/>
- [15] <http://docs.oracle.com/javaee/6/tutorial/doc/bnayl.html>
- [16] <http://cxf.apache.org/docs/maven-cxf-codegen-plugin-wsdl-to-java.html>
- [17] <https://github.com/camunda/camunda-bpm-examples/tree/master/servicetask/soap-cxf-service>
- [18] <https://github.com/camunda/camunda-bpm-examples/tree/master/servicetask/soap-service>
- [19] <http://resteasy.jboss.org>
- [20] <http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html>
- [21] <https://github.com/camunda/camunda-bpm-examples/tree/master/servicetask/rest-service>
- [22] <https://github.com/camunda/camunda-consulting/tree/master/snippets/asynchronous-messaging-jms>
- [23] <http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html>
- [24] <http://hibersap.org/>
- [25] <https://sourceforge.net/projects/cuckoo-ra/>
- [26] http://www.sigs-datacom.de/uploads/tx_mwjournal/pdf/erker_fink_JS_06_14_Bp6d.pdf
- [27] <https://github.com/cerker/cuckoo-hibersap-example>

Descargo de responsabilidad y derechos de autor

Sin garantía : las declaraciones hechas en esta publicación son recomendaciones basadas en la experiencia práctica de los autores. No forman parte de la documentación oficial del producto de Camunda. Camunda no puede aceptar ninguna responsabilidad por la exactitud o puntualidad de las declaraciones realizadas. Si se muestran ejemplos de código fuente, no se puede garantizar una ausencia total de errores en el código fuente proporcionado. Se excluye la responsabilidad por cualquier daño resultante de la aplicación de las recomendaciones presentadas aquí.

Copyright © Camunda Services GmbH - Todos los derechos reservados. La divulgación de la información presentada aquí solo se permite con el consentimiento por escrito de Camunda Services GmbH.

Printed November 26, 2019. Applies to Camunda **7.11**. Any feedback? **best-practices@camunda.com**!