

processengine - Technical Guide

Created by [Arkadiusz Balwierz](#), last modified by [Axel Grossmann](#) on [16.10.2015 CEST](#)

The hybris **processengine** functionality enables users to define business processes through XML process definitions and runs these processes in an asynchronous way. It guarantees that actions are performed in the right order and on the right condition. A new process can be created for each process definition. It is possible to manage these processes by using their own context.

Additionally, the **processengine** delivers a solution for waiting for events, notifying users or user groups, firing actions (defined in spring beans), and flow decisions based on action results.

**Note**
It is a good idea to study java and xml files in the extension **yacceleratorfulfilmentprocess**. This is an illustration of the fulfillment process and neatly demonstrates how to create a process.

- [Introduction](#)
 - [Simplified Work Sequence Example](#)
- [Process Definition Creation](#)
 - [Process Definition](#)
 - [Introduction](#)
 - [Defining in Spring](#)
 - [Node Types](#)
 - [Action Node](#)
 - [Wait Node](#)
 - [Notify Node](#)
 - [End Node](#)
- [Actions](#)
 - [Interface](#)
 - [Transitions](#)
 - [Action Superclasses](#)
 - [Available Actions](#)
 - [Retries](#)
 - [SubProcesses](#)
- [Support for Scripts in Business Process](#)
 - [Calling a Script](#)
 - [Accessing a Business Process Context from a Script](#)
 - [Using hMC to Create a Dynamic Process Definition at Runtime](#)

About this Document

This document introduces the hybris **processengine** functionality. It informs you about structure, implementation, and useful routines.

Audience: Consultants, developers

Validity:
5.0.0 and higher

Based on hybris version:
5.2.0

Rate the document:

Your Rating:	Results:	24 rates
--------------	----------	----------

Related Resources

- [Business Process Management](#)
- [Order Management Module](#)

See Also

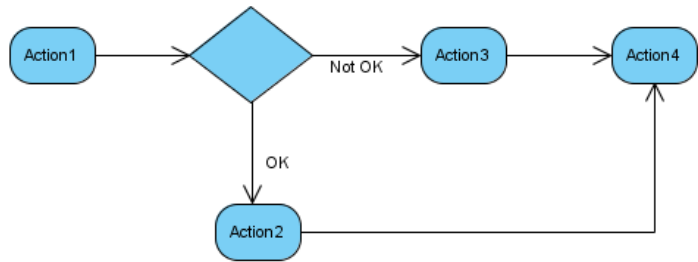
- <https://jaxb.dev.java.net/>

Introduction

The **processengine** represents an engine to build and interpret a memory structure of a process. This memory representation is based on **nodes**. To build a correct object tree, a **processdefinition.xsd** file is used by JAXB. The result of this processing is a map that relates nodes and their IDs. Information about the node ID that is the next one in the process is stored in different ways for different nodes. Wait, notify, and split nodes have their next nodes explicitly listed. In case of alternatives, use the action node. Its execution path logic is described further below.

Simplified Work Sequence Example

When starting to work with the **processengine**, it is advisable to do a business analysis first.



1. The workflow gained from this analysis then needs to be translated to the process XML file.

```
<?xml version="1.0" encoding="utf-8"?>
<process xmlns="http://www.hybris.de/xsd/processdefinition" name="Example" start="Action1">
  <action id="Action1" bean="Action1">
    <transition name="OK" to="Action2"/>
    <transition name="NOK" to="Action3"/>
  </action>
  <action id="Action2" bean="Action2">
    <transition name="OK" to="Action4"/>
  </action>
  <action id="Action3" bean="Action3">
    <transition name="OK" to="Action4"/>
  </action>
  <action id="Action4" bean="Action4">
    <transition name="OK" to="success"/>
  </action>
</process>
```

```
</action>
<end id="success"state="SUCCEEDED">Everything was fine</end>
</process>
```

2. Next, define the beans in `|your_extension_name|-spring.xml` file.

```
<bean id="Action1"class="org.training.actions.Action1" parent="abstractAction"/>
<bean id="Action2"class="org.training.actions.Action2" parent="abstractAction"/>
<bean id="Action3"class="org.training.actions.Action3" parent="abstractAction"/>
<bean id="Action4"class="org.training.actions.Action4" parent="abstractAction"/>
```

3. Finally implement the actions classes. Below find the example of implementation of the **Action1**.

```
package org.training.actions;public class Action1 extends AbstractSimpleDecisionAction
{
    @Override    public Transition executeAction(final BusinessProcessModel process)
    {
        if(.....)
        {
            return Transition.NOK;
        }
        else
        {
            return Transition.OK;
        }
    }
}
```

Process Definition Creation

To create a new process instance, call the **createProcess** method from the **BusinessProcessService** service. You can then run this service using the **startProcess** method. If a process definition has not been created before, the **ProcessDefinitionFactory** will create one in this step.

```
businessProcessService.startProcess(id, processName);
```

Process Definition

Introduction

The process definition defines a set of nodes that are connected with each other through their IDs. The process definition starts with the process header. To make the process definition visible to the **ProcessDefinitionFactory**, it is necessary to declare it as a resource in Spring.

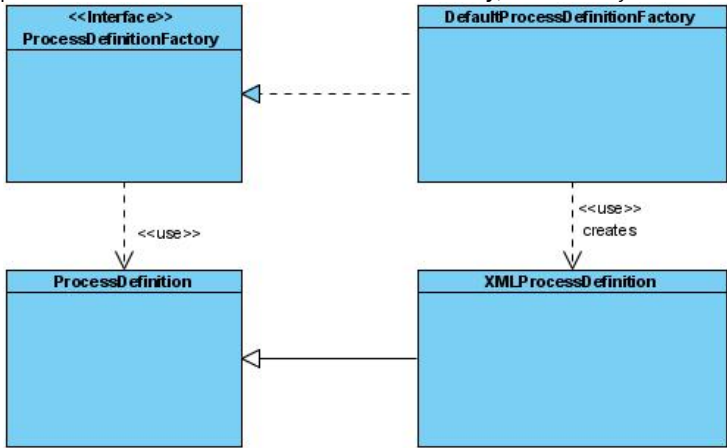


Fig. Process Definition

Defining in Spring

The Process

The Spring definition of a process is straightforward:

```
<bean id="placeorderProcessDefinitionResource"class="de.hybris.platform.processengine.definition.ProcessDefinitionResource"
>
  <property name="resource"value="classpath:/processdemo/placeorder.xml"/>
</bean>
```

The table below explains the meaning of the current `<bean>` element attributes.

Attribute	Description
id	This attribute needs to be unique in Spring

class	This attribute should always be as presented. It is the indicator for the ProcessDefinitionFactory ; where to search for beans in the Spring context for process definitions.
-------	--

The <bean> element contains the <property>element. The table below explains the meaning of the <property> element attributes.

Attribute	Description
name	The name of the property is: resources . The name indicates the value of this property shows where in the file system the process definition XML file is located.
value	The value attribute contains the actual path to the process definition XML file.

The Actions

The Action beans also need to be declared in Spring:

```
<bean id="checkOrder" class="de.hybris.platform.fulfilment.actions.CheckOrder"
  parent="abstractAction">
  <property name="checkOrderService" ref="checkOrderService"/>
</bean>
```

The table below explains the meaning of the current <bean> element attributes.

Attribute	Description
id	This attribute is important as it relates the Spring bean with the action node.
class	This attribute points to the class name that realizes the action interface.
parent	This attribute is only used for information.

Additionally, the properties can be set in Spring. In the example above, the property **checkOrderService** is set.

The Root Tag and the Process Class


The definition of the process needs to be written in an XML file. This file must be adequate to the definition in the **processdefinition.xsd** . Each process definition starts with a header.

```
<?xml version="1.0" encoding="utf-8"?>
<process xmlns="http://www.hybris.de/xsd/processdefinition" name="consignmentFulfilmentSubprocess"
start="waitBeforeTransmission" onError="onError" processClass="de.hybris.platform.fulfilment.model.ConsignmentProcessModel">
... ..
</process>
```

Attribute	Description
name	This attribute tells the process' name.
start	This attribute tells the ID of the start node.
onError	This attribute gives the node that is called when an error occurs.
processClass	This attribute refers to the class that implements the process context. The normal routine is to extend this class for extra fields where you can store extra process parameters.

Node Types

After the process is defined, content must be added to the process. A process is described by a set of nodes, which represent the steps in a given process. Each node, excluding the end node, needs to define which node has to be invoked next in the workflow. The most important field in each node is its ID. This is the key that joins two nodes in a workflow.

 **Remember**
The **start** attribute in the process root tag should point to one of the declared nodes

Action Node

Action nodes are the nodes that realize the process logic.

Example:

```
<action id="isProcessCompleted" bean="subprocessesCompleted">
  <transition name="OK" to="sendOrderCompletedNotification"/>
  <transition name="NOK" to="waitForWarehouseSubprocessEnd"/>
</action>
```

The **bean** attribute points to the bean **id** declared in the **spring.xml** file. It defines which action will be fired. For details, see the [Actions](#) section below.

The table below shows the attributes of the <transition> element.

Attribute	Description
name	Tells the result of an action.

to	Tells the Process Engine to which node the process should go after the result of an action is returned.
-----------	---

Wait Node

Wait nodes are used to communicate with the external environment. Use this node if somewhere in the process you need to wait for an external process result. It is also used if waiting for subprocesses to finish their routine.

Example:

```
<wait id="waitForWarehouseSubprocessEnd" then="isProcessCompleted"> <event>ConsignmentSubprocessEnd</event>
</wait>
```

The attribute **then** is the ID of the node that has to be invoked after the wait condition is fulfilled.

The element `<event>` defines the name of the event that activates this node. Internally, the **processengine** prepends the process's code to the event name to make it unique. Set **prependProcessCode="false"** to not prepend the process's code. As of 4.2.0.3 there is an expression language that you can use in your event names. If you have defined processes that rely on the old behavior, you need to change the definition to something like this:

```
<wait id="waitForWarehouseSubprocessEnd" then="isProcessCompleted" prependProcessCode="false">
  <event>${process.code}_ConsignmentSubprocessEnd</event>
</wait>
```



Note
ConsignmentSubprocessEnd is an event that you need to trigger by using `de.hybris.platform.processengine.BusinessProcessService.triggerEvent(String)`.



Important
Process is a process which waits for the event specified in the event parameter.

N.B.: In the expression language you have two variables:

- process: the current instance of `BusinessProcessModel`
- params: a `Map` with the current process parameters

Support for User Input

Since version 5.7 of the hybris Commerce Suite, there is also an option to provide additional information called **choice** when triggering an event. Based on the choice, a process may transit to different nodes. Here is an example of using a wait node with multiple choices:

```
<wait id='waitNode' then='nothingChoosen' prependProcessCode='false'>
  <case event='eventWithChoice'>
    <choice id='first_choice' then='firstChoiceChoosen' />
    <choice id='second_choice' then='secondChoiceChoosen' />
  </case>
</wait>
```

If you trigger an event without **choice**, the business process will transit to a node defined by the **then** argument ('nothingChoosen' in a given example). To trigger an event with **choice**, you can use this method:

```
de.hybris.platform.processengine.BusinessProcessService.triggerEvent(BusinessProcessEvent)
```

Here is an example showing how to trigger event with given **choice**:

```
final BusinessProcessEvent event = BusinessProcessEvent.builder("eventWithChoice").withChoice("second_choice").build();
businessProcessService.triggerEvent(event);
```

Timeout

Since version 5.7 of the hybris Commerce Suite, you can define a timeout on the wait node. When the node with defined timeout doesn't receive an event within a defined time, the transition configured on timeout element will be performed. Here is an example definition of a wait node with a defined timeout:

```
<wait id="waitForWarehouseSubprocessEnd" then="isProcessCompleted">
  <event>ConsignmentSubprocessEnd</event>
  <timeout delay='PT30S' then='timeout' />
</wait>
```



The delay format is defined by duration component of ISO 8601 standard.

Split Node



Note
Use of split nodes is discouraged.

The Split node is used when a process needs to run actions or strings of actions in parallel.

Example:

```
<split id="split">
  <targetNode name="rnd"/> <targetNode name="sayC"/>
</split>
```

The <targetNode> element defines the next nodes after the process is split.

Notify Node

Use this node if there is need to inform a user group or a particular user of a state of a process.

Example:

```
<notify id="notifyadmingroup"then="split">
  <userGroup name="admingroup"message="Perform action"/>
  <userGroup name="othergroup"message="other message"/>
</notify>
```

The attribute **then** of the <notify> element defines the next node in the process that is invoked after one member from the **userGroup** from each group accepts a message.

The table below shows the attributes of the <userGroup> element.

Attribute	Description
name	The group of users to which a message (one after other) has to be sent. There can be several <userGroup> elements. The order of the informed groups is the same as in the definition.
message	Based on the presented example, after a member of the admingroup has committed a message, a message for othergroup is generated. After this message has also been committed the process is informed and the next node (in this example split) is executed.

End Node

This node ends the process and stores state and message in a process item.

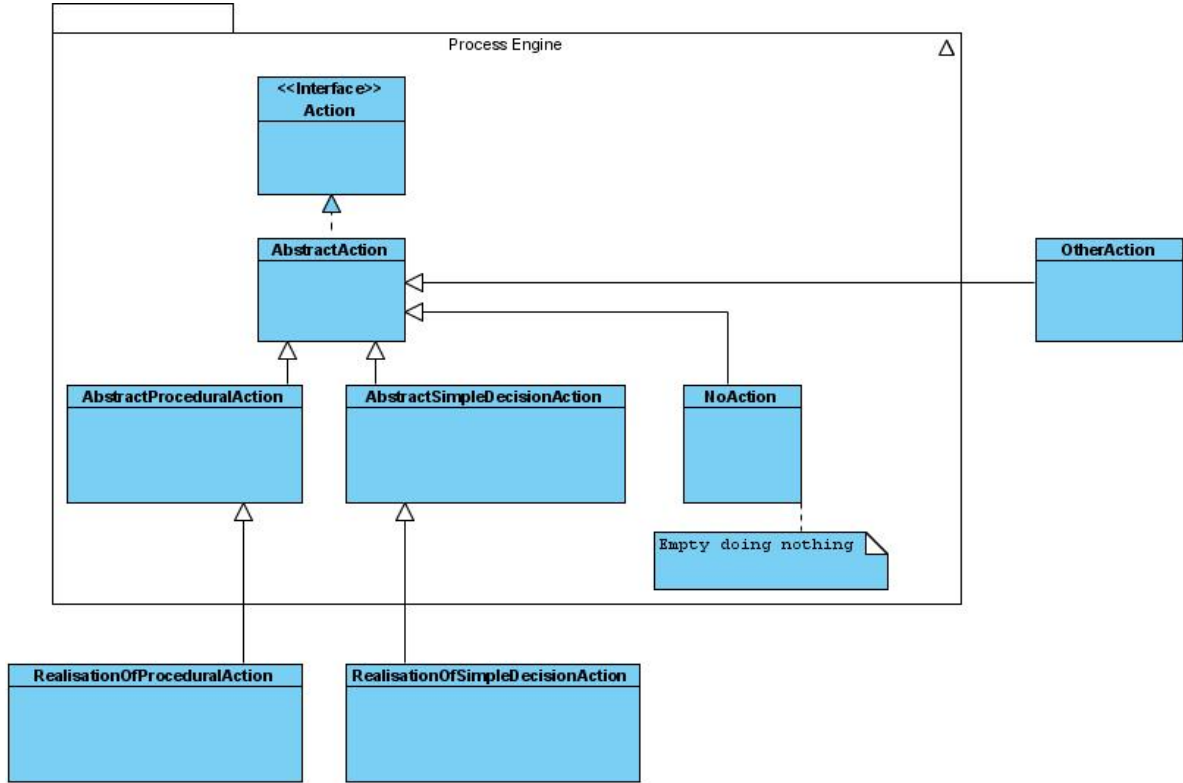
Example:

```
<end id="error" state="ERROR">All went wrong.</end>
<end id="success" state="SUCCEEDED">Everything was fine</end>
```

The **state** attribute tells a process state set after the node is executed. The content of this element, for example **All went wrong.**, is a sample message stored in a process item.

Actions

Actions are the most important part of the **processengine** functionality. Normally, they have to implement a logic or call specialized services to execute tasks that are necessary in a process. An Action performs a single piece of work within a process. Usually it acts upon input data which has been fed into the engine or has been produced by previous actions. Each action produces an action result, which enables the engine to direct the process to the next action. Each action is part of the process definition.



Interface

The Action Interface has two methods:

```
<String> getTransitions();
```

- This method is used for validation if all possible results from an action are mapped in the process definition. In this method a Set of all possible return codes should be returned.

```
String execute(BusinessProcessModel process) throws RetryLaterException, Exception;
```

- This method is used to implement the main logic of an action. The reason for having separated **RetryLaterException** is that this exception is meant to inform the engine to fire an action once again.

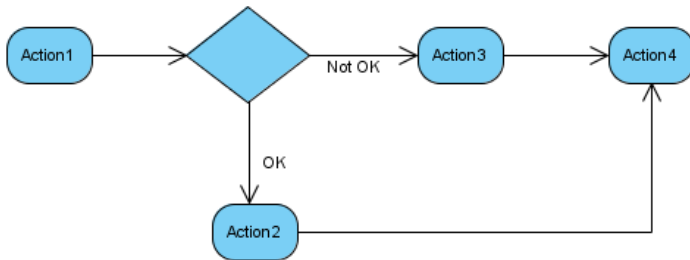
In addition, there are two codes that an action could return outside of the normal action procedure:

- RETRY_RETURN_CODE - same as throwing **RetryLaterException**
- ERROR_RETURN_CODE - same as throwing another Exception

Transitions

The transitions define the target for the next step of a process. The action result determines the route.

Example:



Example workflow should be defined like this

```
<?xml version="1.0" encoding="utf-8"?>
<process xmlns="http://www.hybris.de/xsd/processdefinition" name="Example" start="Action1">
  <action id="Action1" bean="Action1">
    <transition name="OK" to="Action2"/>
    <transition name="NOK" to="Action3"/>
  </action>
  <action id="Action2" bean="Action2">
    <transition name="OK" to="Action4"/>
  </action>
  <action id="Action3" bean="Action3">
    <transition name="OK" to="Action4"/>
  </action>
  <action id="Action4" bean="Action4">
    <transition name="OK" to="success"/>
  </action>
  <end id="success" state="SUCCEEDED">Everything was fine</end>
</process>
```

As we can see, result of **Action1** determines if **Action2** or **Action3** occur, and then workflow in both cases goes back to **Action4**. Of course bean and action ID do not need to be the same.

Action Superclasses

To make an action implementation easy, define an **abstractAction** that implements useful routines in action implementation, such as **de.hybris.platform.processengine.action.AbstractAction**.

There is a set of methods for logging both messages and errors, as well as getters and setters for **ModelService**, **ProcessParameterHelper**, and at the end:

- getProcessParameterValue**: Gets a parameter stored in the process context
- setOrderStatus**: Sets a status of the order
- createTransitions**: Creates a set of transitions for lists of strings

Available Actions

There are also two additional templates to use for a procedural or simple decision action:

- AbstractProceduralAction**: It simply returns OK whatever happens. It is useful to split a Process into smaller pieces. Only an implementation of the **execute** method is necessary.
- AbstractSimpleDecisionAction**: It returns one of OK or NOK values. It is useful to make a simple decision. Transitions are defined and only the **execute** method must be implemented.

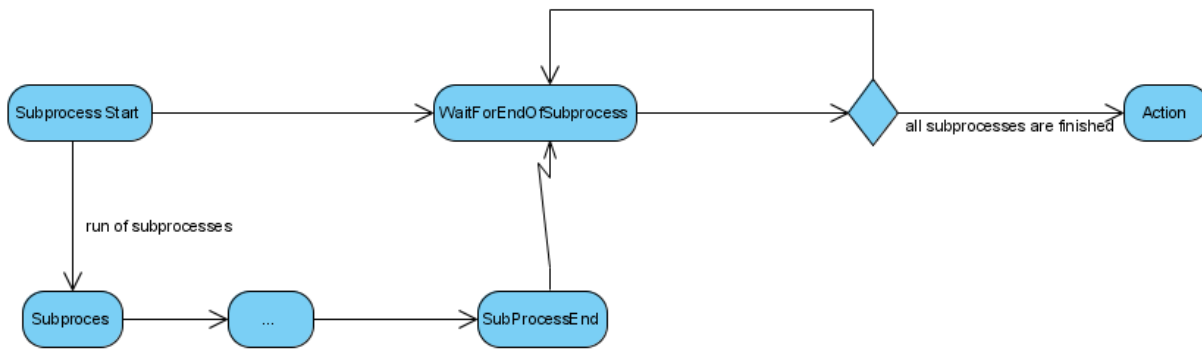
Retries

If an action should be executed once again, the **RetryLaterException** exception or **RETRY_RETURN_CODE** could be returned.

SubProcesses

To run a sub-process, simply call the **BusinessProcessService.startProcess**. Use events to inform the parent process that a subprocess has ended. To fire an event, use **de.hybris.platform.processengine.BusinessProcessService.triggerEvent(String)** Example:

Not All subprocess finished



Support for Scripts in Business Process

With support for scripting and the dynamic process definition, it is possible to declare not only the structure of the business process but also to define the behavior directly in the xml which defines the process.

It is possible to use the hMC to insert a process definition at runtime.

Calling a Script

Here is an example of a process definition with a script that returns the next transition:

```
<?xml version='1.0' encoding='utf-8'?>
<process xmlns='http://www.hybris.de/xsd/processdefinition' start='action0' name='testProcessDefinition'>
  <scriptAction id='action0'>
    <script type='javascript'>(function() { return 'itworks' })()</script>
    <transition name='itworks' to='success' />
  </scriptAction>
  <end id='success' state='SUCCEEDED'>Everything was fine</end>
</process>
```

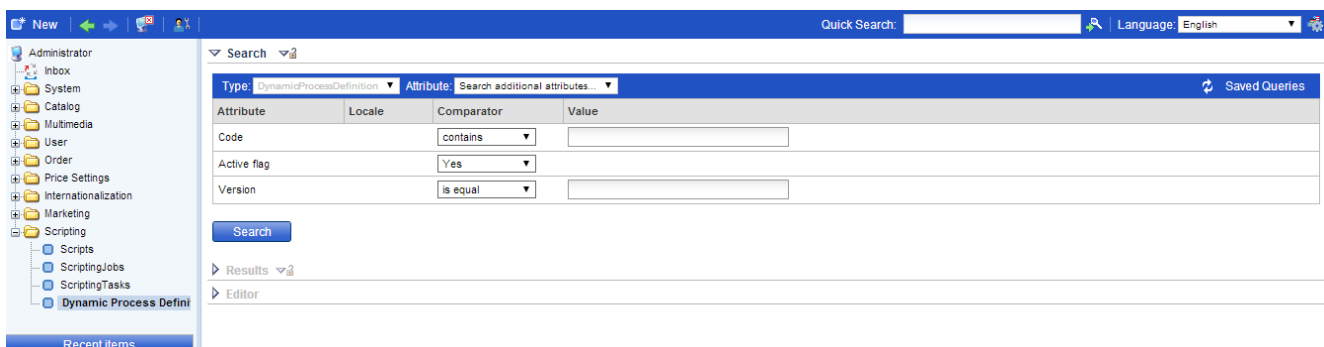
This simple process definition shows the usage of a **scriptAction** element. It is almost exactly the same as an **action** element, but instead of executing some logic from a spring bean, it executes the script defined in a **script** element. Here it only invokes an anonymous function that returns the next transition ('**itworks**').

Accessing a Business Process Context from a Script

Scripts from process definitions are invoked with a special **process** parameter passed to the script. The **process** parameter is an instance of the **BusinessProcessModel** describing the running business process. Here is an example of script that modifies a context parameter of a business process:

```
<?xml version='1.0' encoding='utf-8'?>
<process xmlns='http://www.hybris.de/xsd/processdefinition' start='action0' name='testProcessDefinition'>
  <contextParameter name='testParameter' use='required' type='java.lang.String' />
  <scriptAction id='action0'>
    <script type='javascript'>
      var parameter = process.contextParameters.get(0);
      parameter.setValue('changedFromScript');
      modelService.save(parameter);
      'itworks'
    </script>
    <transition name='itworks' to='success' />
  </scriptAction>
  <end id='success' state='SUCCEEDED'>Everything was fine</end>
</process>
```

Using hMC to Create a Dynamic Process Definition at Runtime






To create a new Dynamic Process Definition, right click the Dynamic Process Definition node, paste your definition in the window, and click **Create**. The code is set automatically based on the name field in the process definition. Content of the process is checked for validity against the platform .xsd schema.

It is worth noting that, if you update a process definition for a process that is already running, the historical version of the process definition is used until the process has finished. When the process is started again, the new definition is used.

If you update a process definition, a normal search will only show the most recent version of the definition that you've updated. However, you can display all versions of your definitions by setting the **Active flag** in the **Search Editor** to **No Preference**. It is also possible to run a Cron job that will automatically remove old process versions.

Child Pages

-  [How To Start processengine Process with a Service Activator - Tutorial](#)
-  [Order Management Tutorial - An Example Process](#)
-  [Process Routine of processengine](#)