



# Platform, Services, and Utilities

Generated on: 2024-11-08 15:01:29 GMT+0000

SAP Commerce | 2205

PUBLIC

Original content: [https://help.sap.com/docs/SAP\\_COMMERCE/d0224eca81e249cb821f2cdf45a82ace?locale=en-US&state=PRODUCTION&version=2205](https://help.sap.com/docs/SAP_COMMERCE/d0224eca81e249cb821f2cdf45a82ace?locale=en-US&state=PRODUCTION&version=2205)

## Warning

This document has been generated from the SAP Help Portal and is an incomplete version of the official SAP product documentation. The information included in custom documentation may not reflect the arrangement of topics in the SAP Help Portal, and may be missing important aspects and/or correlations to other topics. For this reason, it is not for productive use.

For more information, please visit the <https://help.sap.com/docs/disclaimer>.

# Building SAP Commerce

Because of its flexible structure, SAP Commerce needs a framework to automatically run tasks such as code generation and copying, parsing, and compiling files. These tasks are performed by the SAP Commerce build framework using XSL, Apache Ant for automated tasks, and the Eclipse compiler.

## Build Framework Basics

SAP Commerce functionality is built from extensions of the Platform, plus any other product packages. By referencing an extension in the `localextensions.xml` file (located in `<HYBRIS_CONFIG_DIR>`), that extension is integrated into the SAP Commerce build framework. For details on the `localextensions.xml` file, see [Installation Based on Specified Extensions](#).

There are several `ant` targets to build an extension or any of its modules. For details on extension modules, see [the Extension documentation](#). For an extensive list of `ant` targets, see [Ant Targets](#) section below.

The directory in which you run a build framework call such as `ant all` defines the scope of the build. There are two different kinds of scope: Platform-wide and extension-wide.

Scope	Ant Call Directory	Components Built	Example Directory Path
Platform-wide	<code>&lt;HYBRIS_BIN_DIR&gt;/platform</code>	Platform and all extensions	C:\hybris\bin\platform, /usr/hybris/bin/platform
extension-wide	<code>&lt;extensionhome&gt;</code>	Current extension only	C:\hybris\bin\modules\marketplace\marketplacestore, /usr/hybris/bin/modules/marketplace/marketplacestore

SAP recommends running full SAP Commerce builds. The SAP Commerce version is included in the `build.number` file located in the `<HYBRIS_BIN_DIR>/platform` directory.

## Platform Build Phases

There are three main phases during a build:

- Preparation
- Dependency update
- Extension building

### Preparation

In the first phase of a Platform build, the build program performs the following tasks:

- Checks the directories specified by the `<HYBRIS_DATA_DIR>`, `<HYBRIS_TEMP_DIR>`, and `<HYBRIS_LOG_DIR>` [SAP Commerce environment variables](#) and creates the directories if they do not exist.
- Checks the directory specified by the `<HYBRIS_CONFIG_DIR>` [SAP Commerce environment variable](#) and prompts for a path if the directory does not exist.
- Resolves [extension dependencies](#). By specifying the `<requires-extension="myextension" />` tag in an `extensioninfo.xml` file, you cause that extension to require `myextension`. The consequences are:

- During the build process, the dependent extension (the extension that depends on `myextension`) is only built after `myextension` has been built. The exact order in which extensions are built is not deterministic, but the build framework abides by extension dependency rules. In other words, you cannot predict exactly when `myextension` is built, but it is always built prior to the dependent extension.
- During a SAP Commerce initialization or update, `myextension` Manager is called prior to the dependent extension Manager. That way, the dependent extension can rely on data created by `myextension` Manager to be available (for example, sample products).

SAP Commerce abides by extension dependency rules. In other words, you cannot know when The exact order in which extensions managers are referenced is not deterministic, but the SAP Commerce `myextension` Manager is called, but it is always called before the dependent extension Manager. Refer to [Extension Dependencies](#) and the documentation on the [extensioninfo.xml](#) file.

- Prepares the build framework by bootstrapping and copying `build.xml` files into each extension directory.
- Generates source files for all extensions (for example, abstract [Jalo Layer](#) classes based on `items.xml` files)
- Generates [Models](#) for use with the SAP Commerce [ServiceLayer](#)

The preparation steps are necessary setup steps. They are always executed on a full SAP Commerce build regardless of the ant target. The ant target also determines whether the **extension building** and **extension deploying** steps are executed. Several parts of the build framework, such as the bootstrapping process, are listed for sake of completeness only; you cannot affect them.

## External Dependency Update During First Build

The exact order in which extension managers are referenced is not deterministic, but you can use Maven dependency resolving to download libraries during the first build. This feature is disabled by default because all necessary libraries are shipped with the SAP Commerce package. However, while developing a solution based on SAP Commerce, it might be a good idea to download libraries rather than keep them in local storage.

To use external library download, update your `external-dependencies.xml` (this is where all Maven-managed dependencies should be described) and `unmanaged-dependencies.txt` ( this is a list of libraries that are not managed by Maven, like extension-specific jar files). Also, change the value of the flag `usemaven` to `true` in the `extensioninfo.xml` file (`extension` section).

Because external dependencies are no longer part of Platform, the first build takes more time to complete: during the first build, the Platform downloads dependencies from the SAP Commerce repository.

### Caution

The ant `updateMavenDependencies` task deletes all `*.jar` files from the `lib` folder by default. Only libraries listed in `unmanaged-dependencies.txt` files are not deleted.

For more information, see [Extension Library Management Using Maven](#).

## Building Extensions

A full SAP Commerce build iterates through all available [extensions](#) and builds them in a nondeterministic order that complies with extension dependencies. If this phase is run explicitly by calling `ant` in an extension directory, only that one extension is built.

Before the extension is built, the build framework processes the `before_build` [callback target](#). Then, the build framework performs the following tasks:

- Runs some validations
- Generates source code files if necessary

- Starts compiling the actual extension, which compiles the following modules:
  - Extension core module,
  - Extension web module,
- Runs the **after\_build** [callback target](#)

An extension build does not run the preparation steps, so the bootstrapping and copy `build.xml` files steps are not executed. This means that required parts of the build framework may not be available. In other words, if you have not yet run a full Platform build, you may not be able to build an extension from its directory because the required parts of the build framework are not there.

Therefore, we recommend always running a full build after a SAP Commerce download so that the build framework components are available to all extensions.

During an extension build, SAP Commerce generates Java source files based on the type definitions of the extension in the `items.xml` file. There are different kinds of files that are generated:

- [Models](#) for the [ServiceLayer](#)
- Abstract [Jalo Layer](#) classes (carrying a prefix, such as `Generated`). These files are generated anew if you have modified the `items.xml` for the extension. For details, refer to the [Type System documentation](#).
- Nonabstract [Jalo Layer](#) classes. These are only generated when the file does not yet exist. If a file with the same name exists, the file is not generated anew.

## i Note

Jalo is deprecated.

The SAP Commerce build framework uses the `javac` ant task <http://ant.apache.org/manual/Tasks/javac.html> for compilation tasks; refer to the documentation on the `javac` ant task for additional details on the compilation process. One important aspect is that the compiler does not scan the source of Java files in full, but it compares the timestamp of the Java source file and compiled class file. The Java source file is compiled automatically only if the Java source file timestamp is newer than the compiled class file timestamp. Consequently, situations may arise where the compiled class files are older than the Java source files but the compiler does not detect this (for example, if you update source code from a repository for an extension, but not for the Platform or vice versa). In such cases, Java source files may rely on classes or methods that are not yet or no longer available in the compiled class files, and the Platform build fails. If you experience build fails, try calling `ant clean all` from the `<HYBRIS_BIN_DIR>/platform` directory to remove the compiled class files prior to the build.

## Strict Compilation Mode in Platform

SAP Commerce includes a Platform compilation mode called `strict`. Use this mode to compile all extensions using only their declared dependencies instead of one Platform-wide classpath.

To enable the mode, navigate to the `local.properties` file and use the following property with the value `true`:

```
build.strict.compilation.mode=true
```

## Pre-Bundled Apache Tomcat Configuration Files

Part of the [SAP Commerce Server](#) is an Apache Tomcat prebundled with the Platform. For details on how the Build Framework handles the prebundled Apache Tomcat configuration files, refer to the [SAP Commerce Server](#) documentation.

## Ant Targets

Depending on the folder that you are in, you have several ant targets at your disposal to start building different Platform components. For example, `ant -p` displays a list of suitable build targets for the current directory or extension.

Specifying multiple build targets causes the build framework to process all ant targets in the order they are written. For example, `ant clean all ear` causes SAP Commerce to execute the build targets `clean`, `all`, and `ear` in that order. For more information, see [Ant Targets](#).

## Individual Override Configurations

The SAP Commerce allows an ant parameter that reads in another properties file after the `local.properties` file. Refer to [Configuring the Behavior of the SAP Commerce](#) for details.

## Extending the Build Framework

Extensions can affect the Platform build process by hooking in ant calls of their own (for example, to prepare files that need to be compiled or to copy or delete some files from a directory).

To include a custom build logic, edit the `buildcallbacks.xml` file that exists in each extension main directory. There are two ways for the `buildcallbacks.xml` file to affect the Platform build process: predefined macrodefs and global ant targets.

### Macrodefs

By specifying macrodefs in a `buildcallbacks.xml` file, you can integrate a custom build logic into the Platform build process. These macrodefs take effect before or after a certain ant target has completed a certain build process stage. You can, for example, overwrite, delete, or copy files that your extension needs.

The macrodefs names follow a certain pattern. The name starts with the name of the current extension, followed by the keyword `after` or `before`, and lastly the name of the build target that is to be intercepted. For example, the following sample macrodef snippet is part of the Catalog extension build framework and would be run after ant `clean` was performed on the Catalog extension.

```
<macrodef name="catalog_after_clean">
    <sequential>
        <delete dir="${ext.catalog.path}/bin" />
    </sequential>
</macrodef>
```

You can refer to the same extension, as well as other ones. For example, the following sample code snippet would copy the CSS files from the hac extension `ext/hac/web/webroot/static/css` directory once the `hac` extension has completed compiling.

```
<macrodef name="myextension_after_compile_hac">
    <sequential>
        <copy todir="${ext.myextension.path}/web/webroot/styles">
            <fileset dir="${ext.hac.path}/web/webroot/static/css">
                <include name="**.css" />
            </fileset>
        </copy>
    </sequential>
</macrodef>
```

Using the `yrun` macro, you can execute beanshell code.

```
<macrodef name="(extension_name)_after_yunitinit">
    <sequential>
        <yrun>
            de.hybris.platform.core.Registry.setCurrentTenantByID("junit");
            new de.hybris.platform.jalo.CoreBasicDataCreator().createEssentialData(null,null);
            de.hybris.platform.util.RedeployUtilities.shutdown();
        </yrun>
    </sequential>
</macrodef>
```

For a list of available buildcallbacks, refer to one of these files:

- The buildcallbacks.xml file in the <HYBRIS\_BIN\_DIR>/bin/modules/platform/yempty directory
- The buildcallbacks.xml file in your extension directory, such as <HYBRIS\_BIN\_DIR>/myextension

For more information, see [Writing and Executing Tests](#).

## Global Ant Targets

If you define ant targets within an extension buildcallbacks.xml file, they are globally available - unlike ant targets in the extension build.xml file, which are only available from the extension main directory. This allows you to start building your extension from the <HYBRIS\_BIN\_DIR>/platform directory, for example.

## Variables

There are also several variables to work with:

- <platformhome>: contains a reference to the Platform main directory.
- ext.name\_of\_extension.path: contains the path to the respective extension. For example, ext.catalog.path contains the path to the Catalog extension.

## Replacing Files in the \${HYBRIS\_BIN\_DIR} Directory

The directory structure of SAP Commerce makes a distinction between configuration files and binary files. However, you might want to replace some binary files. For example, you might want to keep custom versions of JSP or CSS files.

SAP Commerce contains a process to include custom files in the <HYBRIS\_BIN\_DIR> directory. This process copies the content of the <HYBRIS\_CONFIG\_DIR>/customize directory to the <HYBRIS\_BIN\_DIR> directory.

For example, to replace the hybris\_main.css file located in the <HYBRIS\_BIN\_DIR>/platform/ext/adminweb/web/webroot/styles directory, your version of the hybris\_main.css file needs to be located in the <HYBRIS\_CONFIG\_DIR>/customize/platform/ext/adminweb/web/webroot/styles directory.

To trigger the process, call the ant customize ant target in the <HYBRIS\_BIN\_DIR>/platform directory. The process is not triggered as part of another ant target so that it can be called before or after an SAP Commerce build.

## Reference: Initializing the SAP Commerce Through an Ant Target Call

The initialization and update process of SAP Commerce is triggered by a HTTP request to a JSP page of the SAP Commerce Administration Console. Therefore, it is possible to start the initialization and update process via the Apache Ant Get task.

For more information, see:

- [Initializing and Updating SAP Commerce](#)
- <http://ant.apache.org/manual/Tasks/get.html> : Apache Ant Get task

## Related Information

[About Extensions](#)

[Configuration Templates](#)

[Installation Based on Specified Extensions](#)

[Creating a New Extension](#)

[Environment Variables](#)

[Running SAP Commerce](#)[Extension Dependencies](#)[SAP Commerce Directory Structure](#)[Apache Ant tasks](#)[Documentation on the Tanuki Java Service Wrapper](#)[Tomcat manager servlet documentation](#)[Jalo Layer](#)

## Ant Targets

SAP Commerce uses Apache Ant as an automation tool for the SAP Commerce build framework.

SAP Commerce relies on Apache Ant as an automation tool for the SAP Commerce build framework. To get a list of available ant targets for your SAP Commerce version, call `ant -p` from the  `${HYBRIS_BIN_DIR}/platform` directory of SAP Commerce.

## Related Information

[Build Framework](#)[Apache Ant project](#)

## Ant sonarcheck Target

You can use the Sonar and SonarQube server to check the code quality of your extensions.

SonarQube is a tool for scanning and reporting on the quality of your code. SAP Commerce ships with an Apache Ant target for generating the relevant information to pass to your SonarQube instance. For more information, see <http://www.sonarqube.org>.

## Ant sonarcheck Target Usage Examples

Here are some examples of what the `ant sonarcheck` target offers:

- One sonar target that handles both single and multiple extensions specified in the `sonar.extensions` property as a comma-separated list. If not specified, scans all extensions.
- Supports the `sonar.excludedExtensions` property that removes extensions from consideration.
- Defaults project name and project key to the first extension mentioned in the `sonar.extensions` property but should typically be overridden from the command line by the caller.
- Doesn't include `gensrc` directories in the scan.
- By default, excludes jalo classes, generated constants and \*Standalone classes but users can override from the command line if they want these scanned using the `sonar.exclusions` property.
- By default, scans for all programming languages and markup languages supported by the SonarQube server being used but you can override at the command line using the `sonar.language` property.
- Includes all the usual 'source' and 'test' directories for each extension being scanned.
- Examples of usage:
  - Usual usage of scanning the handful of extensions that make up a module or project-specific implementation:

```
ant sonarcheck -Dsonar.extensions commercefacades,commerceservices,acceleratorfacades,acce
```

- Exclude some extensions from the list being scanned:

```
ant sonarcheck -Dsonar.extensions commercefacades,commerceservices,acceleratorfacades,acc
```

- Scan a single extension defaulting project name and key to the extension name:

```
ant sonarcheck -Dsonar.extensions catalog
```

- Scan all extensions defaulting project name and key but including jalo classes, and others, that are normally omitted:

```
ant sonarcheck -Dsonar.exclusions ""
```

## Distribution Process

The Platform framework incorporates a distribution process that you can use to create custom distributions, containing only those extensions that you want to include.

### Distribution Filtersets

All extensions and the Platform are packaged according to filtersets. You can find the standard filtersets in `hybris/bin/platform/resources/ant/dist/filtersets.xml`. There are three predefined filters : `platform.filter`, `extension.source.filter` and `extension.binary.filter`. Since the last two overlap a lot, there is also a shared one called `extension.filter`.

#### i Note

A `.classpath` file specifies which Java source files and resource files in a project are considered by the Java IDE (Eclipse, IntelliJ IDEA), and specifies how to find types outside of the project. When some extension (for example `commerceservices`) is released as a binary, a java library file is created (for example `commerceservicesserver.jar`) and a path to this file needs to be inserted to a `.classpath` file, instead of paths to sources. When some extension is released as a source, a `.classpath` file contains a path to sources and libraries. That's why we have separate `.classpath` and `.classpath_binary` files.

By default, all extensions use these filters. If you have a custom logic (for example, additional jars that are built during buildcallbacks) you can create a new filter in `YOUR_EXTENSION/buildcallbacks.xml` called `extension.YOUR_EXTENSION.source.filter` and `extension.YOUR_EXTENSION.binary.filter`. Since all filters are native ant patternsets, you can also include other patternsets.

This example for the extension `sampleextension` shows how to extend the standard filter:

```
<patternset id="extension.sampleextension.binary.filter">
    <patternset refid="extension.binary.filter" />
    <exclude name="doc/confidential/**/" />
</patternset>
```

### Creating a Distribution from the Command Line

In the `hybris/bin/platform/resources/ant/dist/distmacros.xml` you can find a generic macro `dist`. The `dist` macro has the following attributes:

Attribute	Command Line Argument	If omitted	Description
<code>extensions.source</code>	<code>-Ddist.extensions.source</code>	<i>no extensions</i>	List of extension name that get packaged with the source code
<code>extensions.binary</code>	<code>-Ddist.extensions.binary</code>	<i>all extensions from (local)extensions.xml</i>	List of extension name that get packaged as

Attribute	Command Line Argument	If omitted	Description
			binary
extensions.activated	-Ddist.extensions.activated	<i>all extensions from (local)extensions.xml</i>	List of extension name that will be active in the standard platform/extensions.x
extensions.deactivated	-Ddist.extensions.deactivated	<i>no extensions</i>	List of extension name that will be in the standard platform/extensions.x but out-commented
include.platform	-Ddist.include.platform	true	Include the Platform framework (not platform/ext) in the distribution
finalzipname	-Ddist.finalzipname	<code>#{HYBRIS_TEMP_DIR}/hybris-distribution-\${build.version}-\${DSTAMP}_\${TSTAMP}.zip</code>	Location and name of the final zip name
public.dist	-Dpublic.dist=true	<i>empty</i>	Always use this attribute when creating a public distribution. Using ant dist command with this attribute ensures that the distribution does not include the <b>mysql connector</b> , which cannot be distributed with the Platform for legal reasons: <b>mysql connector</b> file and any other file in <b>public.dist.delete.filter</b> in <b>filtersets.xml</b> .
create.en.langpack	-Ddist.create.en.langpack	true	creates langpack.zip containing all files for 'en' localization

## Targets for Platform and Extensions

### i Note

When creating a public distribution of SAP Commerce, use the target **ant dist -Dpublic.dist=true**. This ensures that the distribution does not include the **mysql connector**, which cannot be distributed with the Platform for legal reasons. See [the table with target attributes](#) for details.

Target	Effect
<b>ant dist</b> in the Platform directory.	By default this target will include the Platform and all extensions as binary that are currently in the <b>(local)extensions.xml</b> .
<b>ant dist</b> in an extension directory.	By default this target will package the extension as binary.

You can extend both targets in the command line, for example:

```
ant dist -Ddist.extensions.source=sampleextension -Ddist.finalzipname=\home\me\dist.zip -
Ddist.activated=sampleextension ant dist -Ddist.include.platform=true -
Ddist.finalzipname=\home\me\distwithplatform.zip
```

## Creating a Distribution from a Distribution Property File

A better way of creating distributions is to define all properties in a file. A distribution property file could look like this:

```
finalzipname=hybris-multichannel-suite-5-${DSTAMP}-${TSTAMP}
extensions.source=acceleratorcms,acceleratorfacades,acceleratorsample
extensions.binary=admincockpit,advancedsavedquery,amazoncloud,azurecl
include.platform=true
```

and could be executed with this command:

```
ant dist -Ddist.properties.file=path/to/file/dist.properties
```

## Package Structure

There are two subfolders in the **hybris/bin** folder in the final distribution package: **modules**, and **platform**. Both have folders that group modules and extensions. These paths are defined by a property called **ext.EXTENSION\_NAME.package**, for example **ext.payment.package=ext-commerce** or **ext.common.package=platform/ext**. You can find the standard definitions in **hybris/bin/platform/resources/ant/dist/packaging.properties**. Whenever you create a new distribution, this file is updated as well. You can set your own package definitions either in  **\${HYBRIS\_CONFIG\_DIR}/packaging.properties** or directly in your extensions **project.properties**.

You can also sneak in your own properties file by referencing it via **-Ddist.packaging.file**

## Localization

By default at the end of dist process **localization.zip** archive containing all files for 'en' localization is created. This behaviour can be altered by setting **-Ddist.create.en.langpack** to false from command line or by including **create.en.langpack=false** property in properties file.

## Using JRebel

JRebel is a productivity tool for Java EE development. By eliminating the wasteful build and redeploy phases from the development cycle, teams can focus on adding new features, improving code quality, and finishing projects faster.

### Installing JRebel

1. Download the latest generic version of JRebel.
2. Extract the zip to a directory of your choice. This manual refers to this directory as **<JREBEL\_HOME>** (for example **c:/jrebel**).

## Configuring the Platform

1. Adjust your JVM start parameters. To start JRebel, add the agent to your run configuration. Use the **<HYBRIS\_CONFIG\_DIR>/local.properties** file to make changes in the JVM parameters defined in **<HYBRIS\_BIN\_DIR>/platform/project.properties**.

To set JRebel for normal run of SAP Commerce server, add the following line:

```
tomcat.javaoptions=-agentpath:"%JREBEL_HOME%/lib/libjrebel64.so"
```

The actual agent library depends on your OS. In the example above, we used agent for linux 64bit. JRebel supports Windows (jrebel64.dll) and Mac OS X (libjrebel64.dylib). 32bit versions are also available.

To set JRebel for debug configuration (<HYBRIS\_BIN\_DIR>/platform/hybrisserver.bat debug), add this line:

```
tomcat.debugjavaoptions=-agentpath:"%JREBEL_HOME%/lib/libjrebel64.so" -Xdebug -Xrunjdwp:transpo
```

2. Execute ant deploy in <PLATFORM\_HOME>.

## Testing JRebel Integration

Right now the JRebel integration has only been tested with SAP Commerce Server (Apache Tomcat), therefore this test assumes you are using the embedded SAP Commerce Server.

Start the SAP Commerce Server using the provided Batch or Shell scripts (hybrisserver.bat or hybrisserver.sh) in <PLATFORM\_HOME>. During startup, you should see several JRebel messages (JRebel header, Directories monitored by JRebel, Spring XML files monitored by JRebel):

```
SYDM50863115A:platform i303703$ ./hybrisserver.sh debug
Running hybrisPlatform on Tomcat...
--> Wrapper Started as Console
Java Service Wrapper Standard Edition 64-bit 3.5.24
Copyright (C) 1999-2014 Tanuki Software, Ltd. All Rights Reserved.
http://wrapper.tanukisoftware.com
Licensed to hybris software - www.hybris.com for hybris Platform
Launching a JVM...
objc[1235]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines,
JRebel: Starting logging to file: /Users/i303703/.jrebel/jrebel.log
2016-07-28 22:56:14 JRebel: Found offline lease from local storage.. executing JRebel with o1
2016-07-28 22:56:14 JRebel:
2016-07-28 22:56:14 JRebel: #####
2016-07-28 22:56:14 JRebel:
2016-07-28 22:56:14 JRebel: JRebel Agent 6.4.6 (201606221135)
2016-07-28 22:56:14 JRebel: (c) Copyright ZeroTurnaround AS, Estonia, Tartu.
2016-07-28 22:56:14 JRebel:
2016-07-28 22:56:14 JRebel: Over the last 2 days JRebel prevented
2016-07-28 22:56:14 JRebel: at least 6 redeloys/restarts saving you about 0.2 hours.
2016-07-28 22:56:14 JRebel:
2016-07-28 22:56:14 JRebel: License acquired from License Server: http://zeroturnaround.mo.s
2016-07-28 22:56:14 JRebel:
2016-07-28 22:56:14 JRebel: Licensed to SAP SE.
2016-07-28 22:56:14 JRebel:
2016-07-28 22:56:14 JRebel: You are using an offline license.
2016-07-28 22:56:14 JRebel:
2016-07-28 22:56:14 JRebel: #####
2016-07-28 22:56:14 JRebel:
Listening for transport dt_socket at address: 8000
.....
2016-07-28 22:56:23 JRebel: Monitoring Spring bean definitions in '/SAPDevelop/hybris/6.0.0.0'
2016-07-28 22:56:23 JRebel: Monitoring Spring bean definitions in '/SAPDevelop/hybris/6.0.0.0'
2016-07-28 22:56:23 JRebel: Monitoring Spring bean definitions in '/SAPDevelop/hybris/6.0.0.0'
2016-07-28 22:56:23 JRebel: Monitoring Spring bean definitions in '/SAPDevelop/hybris/6.0.0.0'
2016-07-28 22:56:23 JRebel: Monitoring Spring bean definitions in '/SAPDevelop/hybris/6.0.0.0'
.....
```

You can also run Platform in debug mode (hybrisserver.bat debug / hybrisserver.sh debug) but remember to set tomcat.debugjavaoptions in your local.properties.

# Using JRebel

## IDE Independent

Whenever you make a change to your code and would like JRebel to redeploy it for you, execute `ant build` from the extension you've changed. This compiles the changed files, and puts them into the `classes` directory where they are picked up by JRebel.

## IntelliJ IDEA

1. Install the JRebel IntelliJ Plugin by following this tutorial: <https://zeroturnaround.com/software/jrebel/quickstart/intellij/>
2. Install SAP Commerce integration plugin for IntelliJ IDEA and follow instructions on importing a SAP Commerce project into IDEA. This is an important step because the plugin sets up all necessary output paths for JRebel.
3. Once you have changed the class, click **Build > Recompile <classname>**. This compiles the changed class into the `classes` directory where it is picked up by JRebel. Alternatively, choose **Build > Recompile project** in case you have changed multiple files.

## Eclipse

1. Install the JRebel Eclipse Plugin following this tutorial: <https://zeroturnaround.com/software/jrebel/quickstart/eclipse/>
2. Open your extension project (or a Platform project) in Eclipse and select **Generate jrebel.xml** from **Context Menu** (Right mouse) and select `yourextenstion/resources/` as a target directory (or `platform/ext/core/resources` for the Platform project).

## Supported Changes

The table gives an overview of which changes can be made by JRebel in a SAP Commerce System at runtime. The goal is to adjust configuration and SAP Commerce internals to support as many use cases as possible.

Area	Description	Supported
Core Module		
Change Item class	Changed <code>Item.java</code> . Test case: Added <code>System.out</code> to existing method.	
Change Spring XML config		
Change Service Interface		
Change Service Implementation	Changed impl of an Interceptor. Test case: Add <code>System.out</code> to <code>ManadatoryAttributesValidator</code> .	
Web Module		
Change class file	Works with <code>ant build</code> .	
Change property file		
Change Spring XML config		
Addon extension		
Change class in <code>acceleratoraddon/web/src</code>	Works with <code>ant build</code> on the storefront extension	

Area	Description	Supported
Change request mapping of a controller in acceleratoraddon/web/src	Works with ant build on the storefront extension	
Change property file		
Change Spring XML config		
Change property file		
Backoffice Module		
Change widget class file	Works with ant build	
Change Spring XML config		

## Gradle Support

Gradle is a dependency management and a build automation tool. Platform supports many of Gradle features.

Gradle doesn't replace Ant, but adds other capabilities, and improves overall project handling. Gradle makes running tests more standardized.

Platform support for Gradle covers:

- generating Gradle project files, and importing projects into an IDE
- reflecting dependencies between standard extensions
- running JUnit tests
- binary and source extensions

The information covered in this topic include:

### [Project Import in IDEs](#)

Gradle allows easy, IDE-agnostic project import. Platform support for Gradle covers project import into Eclipse and IntelliJ.

### [JUnit Tests](#)

When you run JUnit tests from IDE, your results may differ from the results you get from Ant. Gradle prevents such situations thus making running tests more standardized.

### [Spock Tests](#)

With Gradle, you can run Spock tests.

### [Configurable Folders](#)

You can set a custom Gradle folder configuration for a given Gradle project.

## Project Import in IDEs

Gradle allows easy, IDE-agnostic project import. Platform support for Gradle covers project import into Eclipse and IntelliJ.

SAP Commerce is shipped with Eclipse .classpath project files that serve as a generic project configuration for Eclipse only. The developer decides which extensions to include in their custom configuration. With Gradle, you generate settings.gradle and build.gradle configuration files using your local.extensions configuration, which allows IDE-agnostic project import. Gradle makes sure that dependencies between extensions are correct.

For more information, see:

### [Import a Project into IntelliJ IDEA](#)

Follow the steps to import your project into IntelliJ IDEA using Gradle.

### [Import a Project into Eclipse](#)

Follow the steps to import your project into Eclipse using Gradle.

## Import a Project into IntelliJ IDEA

Follow the steps to import your project into IntelliJ IDEA using Gradle.

### Context

The root directory of the Gradle project is `bin/platform`.

To import a project into IntelliJ IDEA using Gradle:

### Procedure

1. Declare extensions for your project in the project `localextensions.xml` file:

```
<hybrisconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="localextensions.xsd">
<extensions>
    <path dir="${HYBRIS_BIN_DIR}" />
    <extension name="deltadetection" />
</extensions>
</hybrisconfig>
```

2. Build your project:

```
ant clean all
```

3. Generate the `settings.gradle` and `build.gradle` files:

```
ant gradle
```

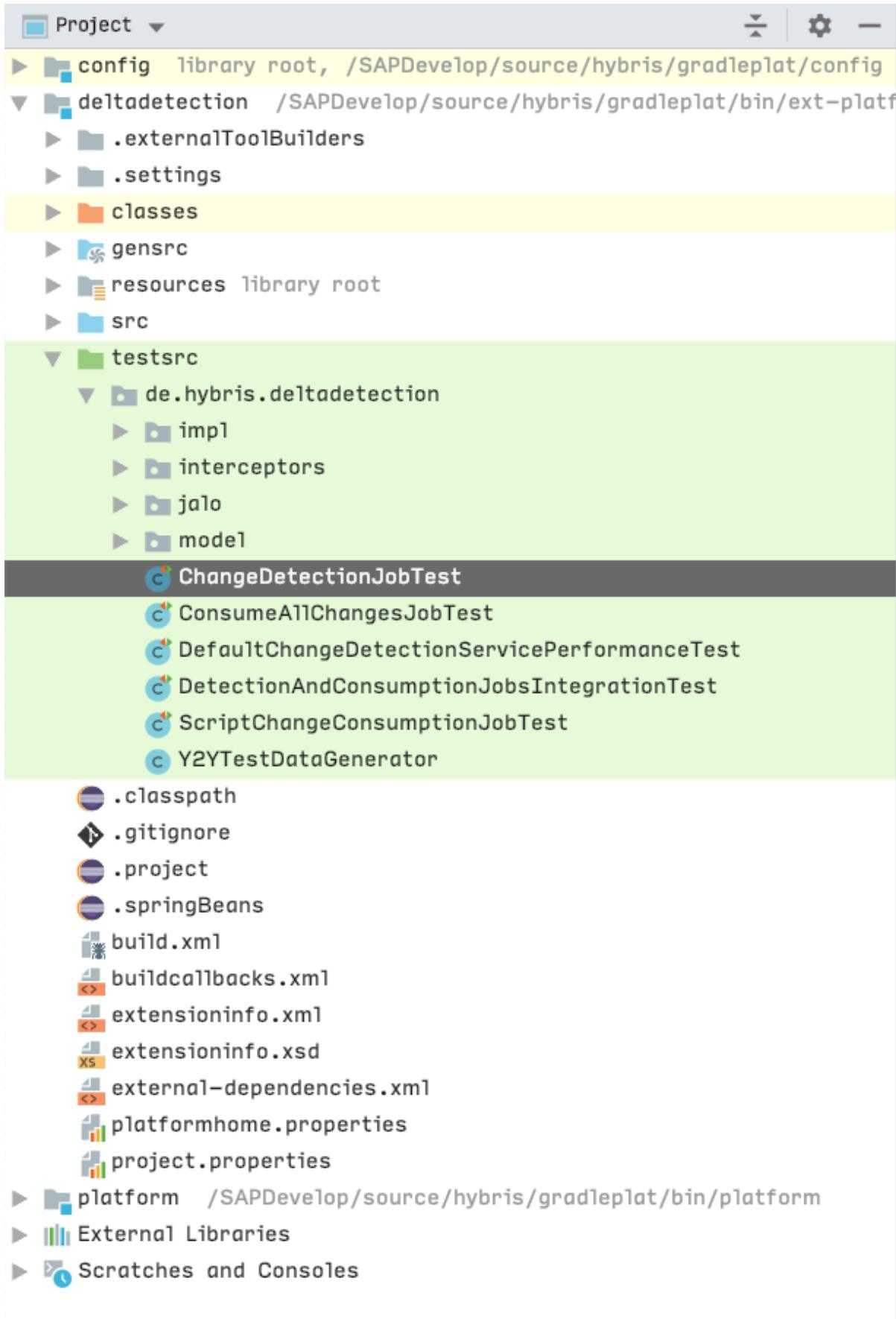
4. Import your project into IntelliJ IDEA using these settings:

- For **Group modules**, check **using explicit module groups**.
- Uncheck **Create separate module per source set**.
- Set your Gradle version in the project preferences settings.

#### → Tip

Use Gradle 6.x or lower to ensure full compatibility.

You should see a similar project structure:



## Synchronizing Project Changes

You may decide to change something in your project, for example add or remove an extension. After you make such a change in your `localextensions.xml` file, you must synchronize it with the project in your IDE.

5. **Optional:** To synchronize a change for your project:

- Run `ant clean all` to rebuild your project.

- b. Run `ant gradle` to update `build.gradle` and `settings.gradle`.
- c. Refresh your project in IntelliJ IDEA.

You have synchronized your changes.

6. **Optional:** To fix possible package import errors, add the following line inside the `<GradleProjectSettings>` element in the `idea/gradle.xml` file and refresh your project:

```
<option name="resolveModulePerSourceSet" value="false" />
```

## Results

You have completed importing your project.

# Import a Project into Eclipse

Follow the steps to import your project into Eclipse using Gradle.

## Prerequisites

Make sure that you have Gradle installed on your machine. You need it to generate the Gradle Wrapper files. For information about installing Gradle, see <https://gradle.org/install/>

## Context

### → Tip

It may be useful in Eclipse to use the info log level for Gradle. It is available in Preferences > Gradle > Program Arguments > --info

The root directory of the Gradle project is `bin/platform`.

## Procedure

1. Declare extensions for your project in the project `localextensions.xml` file:

```
<hybrisconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="localextensions.xsd">
<extensions>
    <path dir="${HYBRIS_BIN_DIR}" />
    <extension name="deltadetection" />
</extensions>
</hybrisconfig>
```

2. Build your project:

```
ant clean all
```

3. Generate the `settings.gradle` and `build.gradle` files:

```
ant gradle
```

4. Generate the Gradle Wrapper files:

```
gradle wrapper
```

5. Prepare the Eclipse classpath:

```
gradlew eclipseClasspath -PprepareEclipseClasspath
```

This step cleans the entries from the `.classpath` files of the default Eclipse project delivered with SAP Commerce. Now that Gradle knows about all dependencies, it manages the content on the fly.

### i Note

If you want Gradle to search for libraries in the local Maven repository, use the following classpath:

```
gradlew eclipseClasspath -PprepareEclipseClasspath -PuseMaven=true
```

Run Gradle before importing Platform to Eclipse the first time, and then after you add a new extension.

As the Buildship plug-in manipulates with Eclipse configuration files such as `.classpath`, `.project`, and others, you can observe unwanted file changes reported by it in the git repository. There's a workaround for ignoring staged files. You can edit the `.git/info/exclude` file and add the following entries:

```
**/.classpath
**/.project
**/.settings/org.eclipse.jdt.core.prefs
```

Then run this command on your edited files:

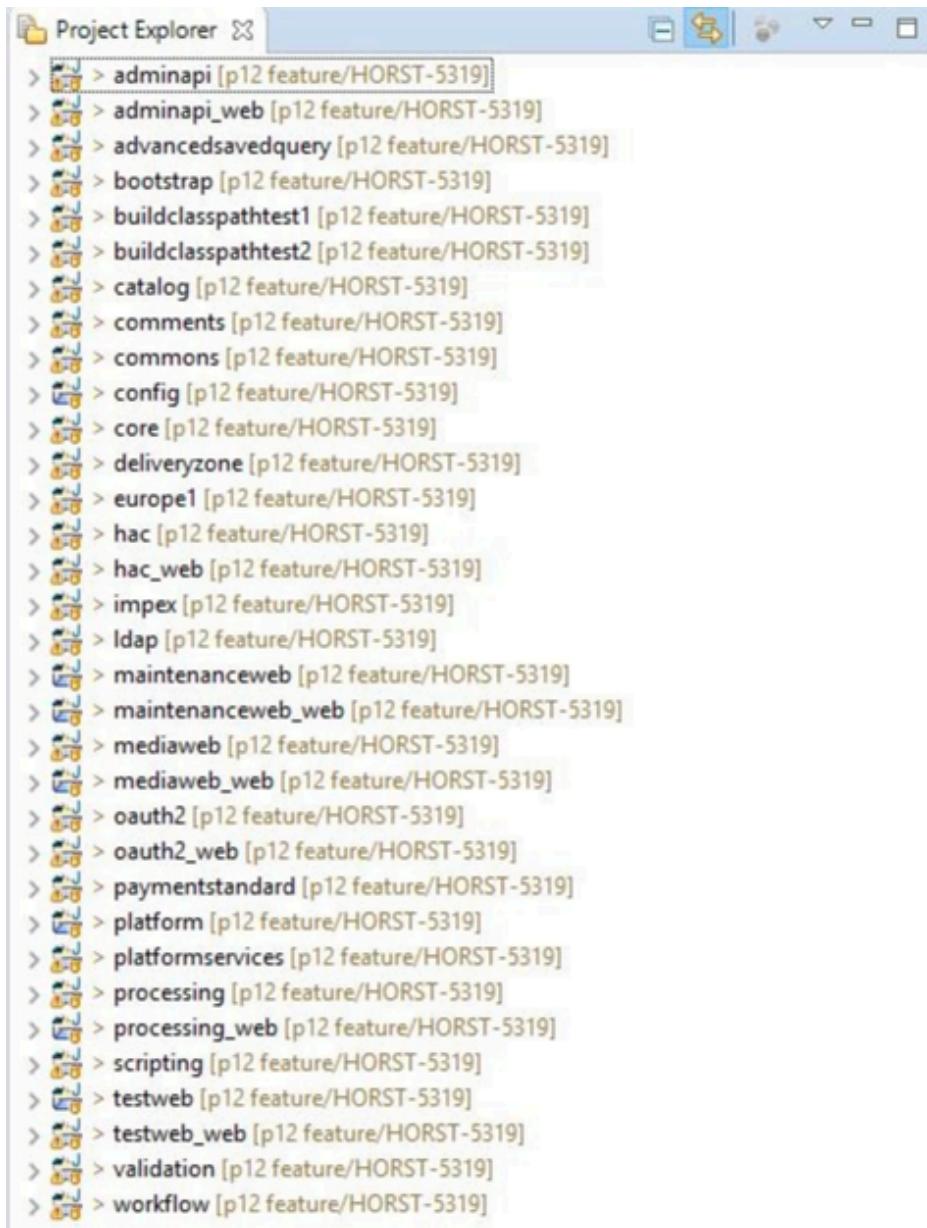
```
git update-index --assume-unchanged [<file>...]
```

For more information, see <https://stackoverflow.com/questions/1753070/how-do-i-configure-git-to-ignore-some-files-locally>

6. Use the Buildship plug-in for Gradle to import your project. Go through the configuration tabs and set up your configuration:

- a. Select  **Gradle**  Existing Gradle Project
- b. Specify the root directory of your Gradle project.
- c. **Optional:** Specify optional import options.
- d. Review your import configuration.
- e. Import your project.

You should get a similar project structure after importing and building the workspace:



## Synchronizing Project Changes

You may decide to change something in your project, for example add or remove an extension. After you make such a change in your `localextensions.xml` file, you must synchronize it with the project in your IDE.

### **! Restriction**

With the Buildship plug-in, you can only synchronize added extensions. Buildship doesn't support removed extensions.

**7. Optional:** To synchronize a change for your project:

- a. Run `ant gradle` to update `build.gradle` and `settings.gradle`.
- b. Choose `Gradle > Refresh Gradle Project` provided in the Buildship plug-in.

You've synchronized your changes.

## Related Information

<https://projects.eclipse.org/projects/tools.buildship>

## JUnit Tests

When you run JUnit tests from IDE, your results may differ from the results you get from Ant. Gradle prevents such situations thus making running tests more standardized.

The mentioned test result differences may occur because the runtime classpath is different from the compile classpath. During compilation, each extension sees only its own dependencies. During runtime, however, all extensions contribute to a single, flat classpath managed by `PlatformInPlaceClassLoader`. As a result, an extension can see classes and beans defined in dependent extensions. Ant reflects such behavior and runs tests correctly - you get the same results as you get when starting the server. An IDE doesn't know about the mentioned limitations and simply uses the compile dependencies. However, Gradle can reflect the correct behavior the same way Ant does it.

For more information, see:

#### [Running JUnit Tests in IntelliJ IDEA](#)

Follow the steps to run JUnit tests using Gradle.

#### [Running JUnit Tests in Eclipse](#)

Follow the steps to run JUnit tests using Gradle.

## Running JUnit Tests in IntelliJ IDEA

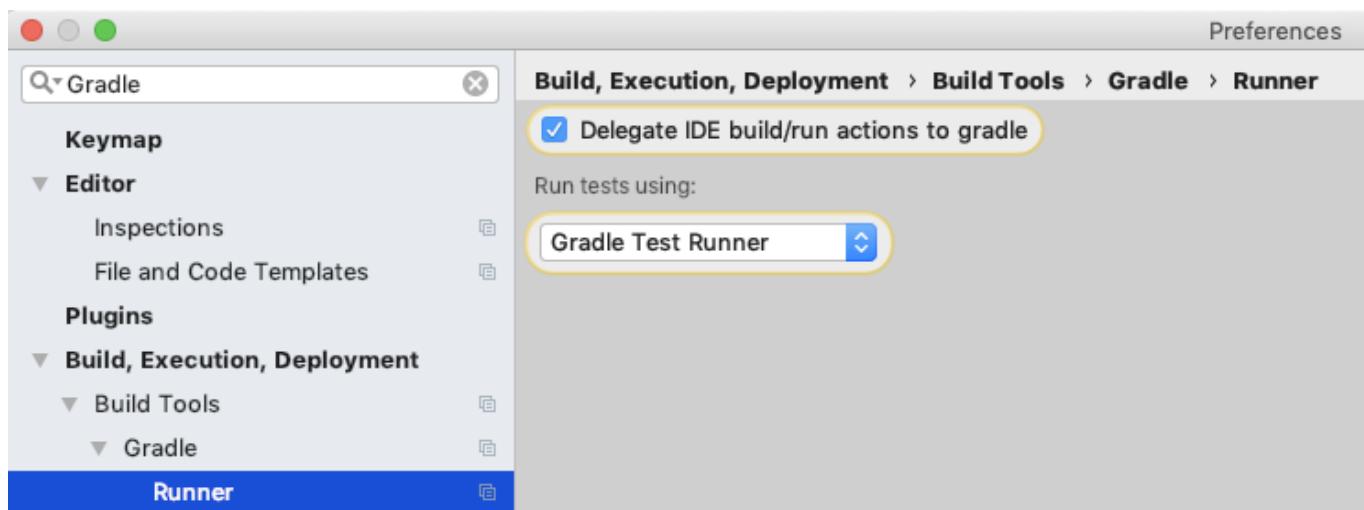
Follow the steps to run JUnit tests using Gradle.

### Context

To run an integration test using Gradle:

### Procedure

1. In IntelliJ IDEA, choose **Delegate IDE build/run actions to gradle**, and **Run tests using: Gradle Test Runner**.



2. Select a test and run it.

The test executes with correct classpath.

## Running JUnit Tests in Eclipse

Follow the steps to run JUnit tests using Gradle.

## Procedure

1. Right-click your test class, and choose **Run As > Gradle Test**.

You can see the execution status in the **Gradle Executions** view.

2. In the **Gradle Executions** view, find a particular test to see its execution result.

You can see executions details in the console, too.

## Debugging in Buildship

There is no support for debugging at the moment in the Buildship plug-in but you can use a workaround for it.

For more information see, <https://github.com/eclipse/buildship/issues/249>

The tested workaround is to run the following command from the console and debug it as a Remote Java Application, for localhost and the 5005 port:

```
gradlew processing:test --tests  
de.hybris.platform.task.impl.TaskScriptingIntegrationTest.testRunTaskForCorrectScript --rerun-  
tasks --debug-jvm --info
```

## Spock Tests

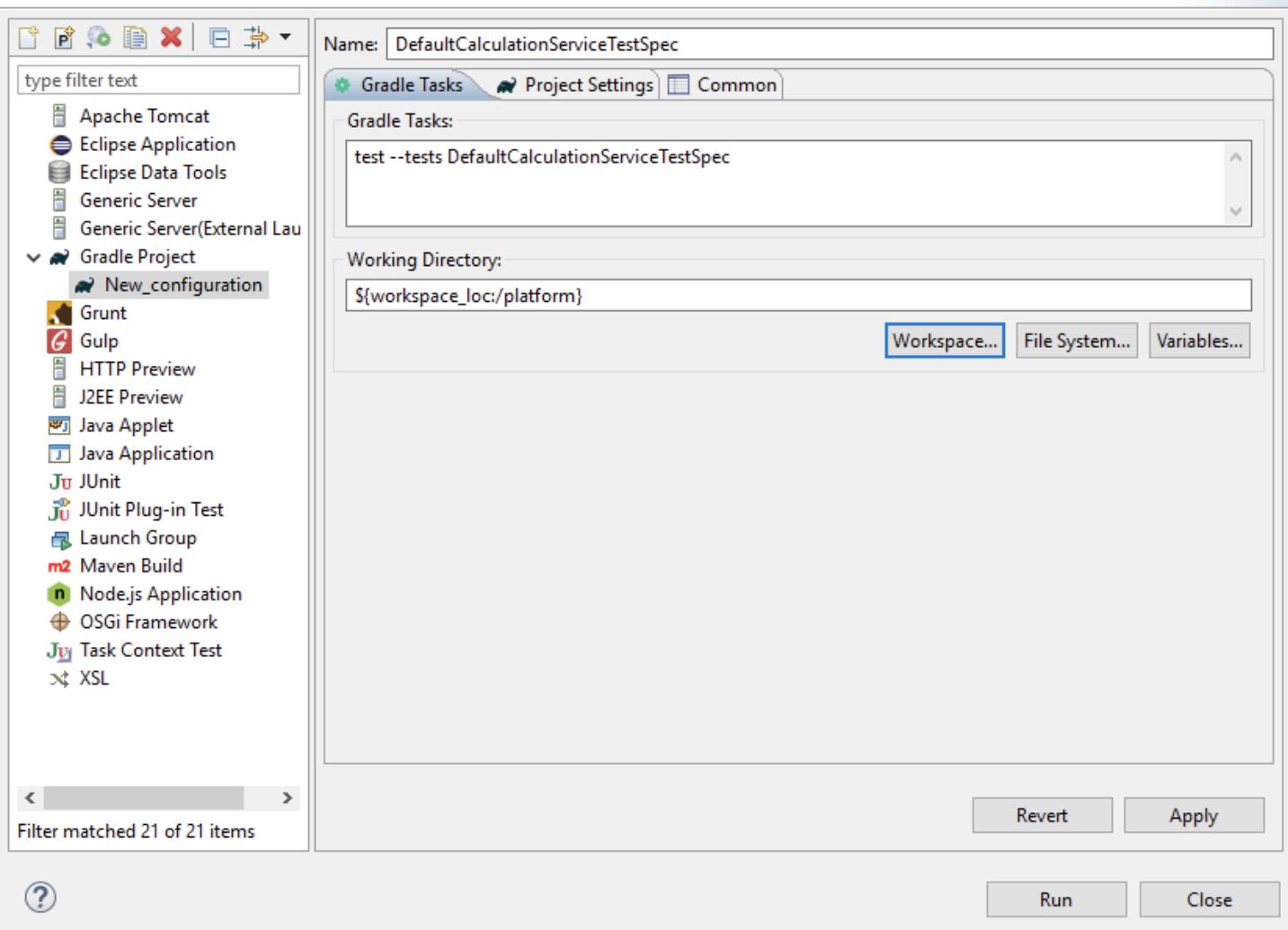
With Gradle, you can run Spock tests.

## Procedure

To run a Spock test, create and run a custom runner:

 Run Configurations

Create, manage, and run configurations



Name: DefaultCalculationServiceTestSpec

Gradle Tasks Project Settings Common

Gradle Tasks:

```
test --tests DefaultCalculationServiceTestSpec
```

Working Directory:

`\${workspace\_loc:/platform}`

Workspace... File System... Variables...

Revert Apply

Run Close

Filter matched 21 of 21 items

## Configurable Folders

You can set a custom Gradle folder configuration for a given Gradle project.

You set such a configuration through a property in `local.properties`. You can mark a folder as a resource, source, or test source. In IntelliJ, you can exclude a folder (mark as excluded), which allows you to get rid of unnecessary indexing.

The property pattern is:

```
gradle.sourceset.gradleProjectName.mark.type=paths
```

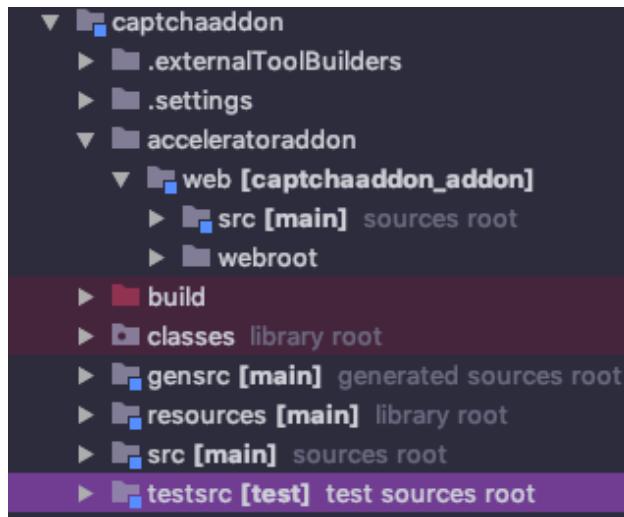
Where:

- `gradleProjectName` is a Gradle project name
- `type` can have these values: `resource`, `src`, `testsrc`, `excluded`

See the examples:

```
gradle.sourceset.captchaaddon.mark.src=src2
gradle.sourceset.captchaaddon_addon.mark.resource=webroot, webroot2
```

An extension may consist of a few Gradle projects - see the `captchaaddon` extension example:



It consists of one more Gradle project - `captchaaddonAddon`. If you want to mark `webroot` as a resource inside `captchaaddonAddon`, set the property to:

```
gradle.sourceset.captchaaddonAddon.mark.resource=webroot
```

## Business Process Management

Business process management in SAP Commerce extensively uses the task service, the cronjob service, and the process engine. Each of them are provided by the `processing` extension.

The topics include:

### [The SAP Commerce processengine](#)

The **processengine** enables you 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 then possible to manage these processes by using their own context.

### [The Task Service](#)

The SAP Commerce Task Service enables you to schedule time-based or event-based actions, even in a cluster environment. Compared to the existing **cron job** functionality, **tasks** should be seen as a simple and easy way of achieving the same outcome without the overhead of the additional cron job features.

### [The Cronjob Service](#)

The **cronjob** functionality is used for executing tasks, called cron jobs, regularly at a certain point of time. Typically cron jobs can be used for creating data for backups, updating catalog contents, or recalculating prices.

### [TenantAwareThreadFactory](#)

The `ThreadFactory` allows you to create threads with some additional implementation.

## The SAP Commerce processengine

The **processengine** enables you 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 then possible to manage these processes by using their own context.

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.

## i Note

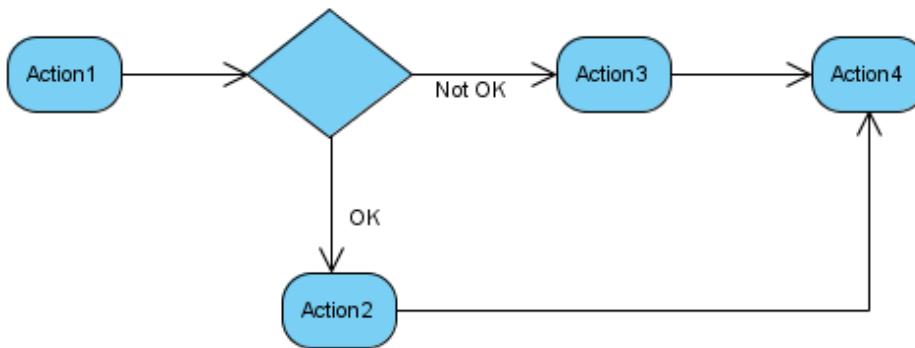
It is a good idea to study Java and XML files in the `yacceleratorfulfilmentprocess` extension. This is an illustration of the fulfillment process and neatly demonstrates how to create a process.

## 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>
    <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;  
    }  
}  
}
```

## 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 file that defines the process.

## 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>
```

## Handling Business Process Restart Requests

By default, the system doesn't restart business processes with nodes that are being run as. It creates a request before each restart to check whether there are any business processes with running nodes. If such processes exist, the system doesn't restart them to ensure that there are no processes with doubled instances of the same nodes running at the same time. As a result, the existing data model isn't broken by concurrent business processes.

Use the following properties to set the number of retries and milliseconds between attempts to request a business process restart:

```
processengine.process.restart.retries=3
processengine.process.restart.millis=500
```

The default values are 3 retries and 500 milliseconds.

The following property ensures that the system throws an exception when requests for restarting business processes fail:

```
processengine.process.restart.exception.if.failed=true
```

The property is set to `true` by default.

If you want the system to allow restarting business processes without any validation of currently running processes, use the following property with the value `true`:

```
processengine.process.restart.legacy=true
```

The property is disabled by default.

#### i Note

Enabling the `processengine.process.restart.legacy=true` property can lead to corrupt behavior caused by processes being run at the same time after being restarted.

## Disconnecting Task Instances from Business Process

To make business processes more resilient in case of database outages or node failures, use the following property:

```
mark.process.as.done.enabled=true
```

The property is enabled by default.

The property ensures that task instances are no longer connected to any business processes when business logic is successfully carried out. It also removes all conditions related to tasks and sets the value of the `TaskModel.RUNNERBEAN` property to `passthroughRunner`. If the process of removing a given task isn't successful, the `passthroughRunner` bean does nothing with it during any attempts to carry it out again, allowing the task to be removed smoothly by the task service.

## Process Definition Creation

The process definition defines a set of nodes that are connected with each other through their IDs.

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);
```

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.

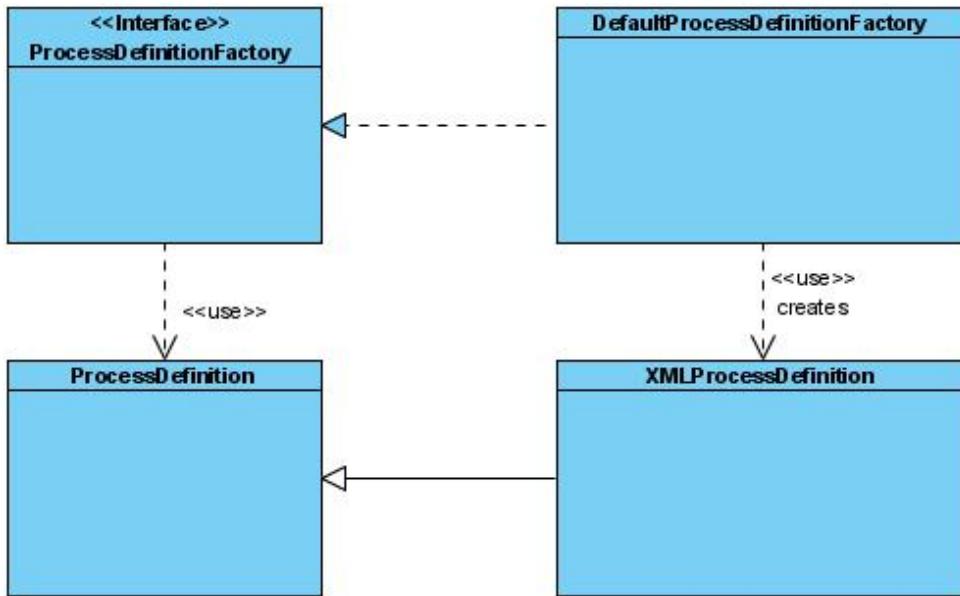


Fig. Process Definition

## 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"/>

```

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

Attribute	Description
<b>id</b>	This attribute needs to be unique in Spring
<b>class</b>	This attribute should always be as presented. It is the indicator for the <b>ProcessDefinitionFactory</b> ; 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
<b>name</b>	The <b>name</b> of the property is: <b>resources</b> . The <b>name</b> indicates the value of this property shows where in the file system the process definition XML file is located.
<b>value</b>	The <b>value</b> 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
<code>id</code>	This attribute is important as it relates the Spring bean with the action node.
<code>class</code>	This attribute points to the class name that realizes the action interface.
<code>parent</code>	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.C
  ...
</process>
```

Attribute	Description
<code>name</code>	This attribute tells the process' name.
<code>start</code>	This attribute tells the ID of the start node.
<code>onError</code>	This attribute gives the node that is called when an error occurs.
<code>processClass</code>	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.

### i Note

#### 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 the Actions section below.

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

Attribute	Description
<b>name</b>	Tells the result of an action.
<b>to</b>	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. There is an expression language that you can use in your event names. The process definition will look something like this:

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

### i Note

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

### i Note

`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

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='waitForEvent' 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 the `de.hybris.platform.processengine.BusinessProcessService.triggerEvent(BusinessProcessEvent)` method.

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

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

### Timeout

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>
```

### i Note

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

## Split Node

### i 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 <b>message</b> (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 <b>admingroup</b> has committed a message, a message for <b>othergroup</b> 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:

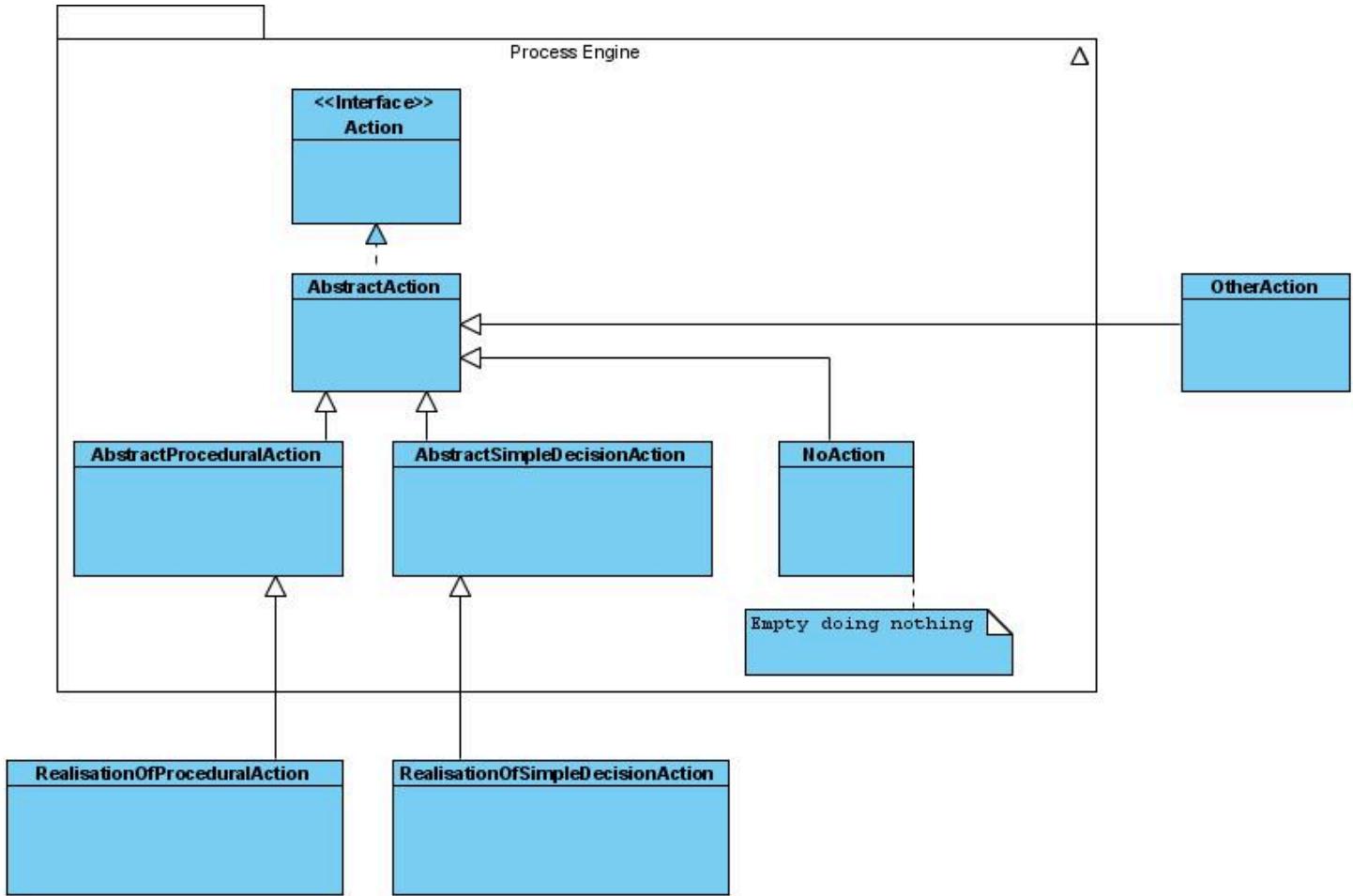
```
<end id="error" state="ERROR">All went wrong.</end>
<end id="success" state="SUCCEEDED">Everything was fine</end>
```

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 an action 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:

- `Set<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) throwsRetryLaterException, 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.

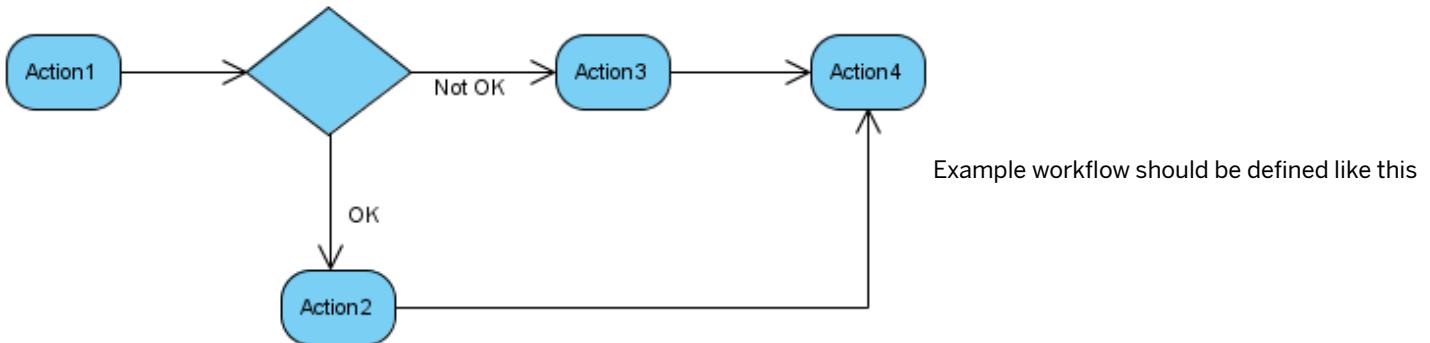
In addition, there are two codes that an action could return outside of the normal action procedure:

- `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:



```

<?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.

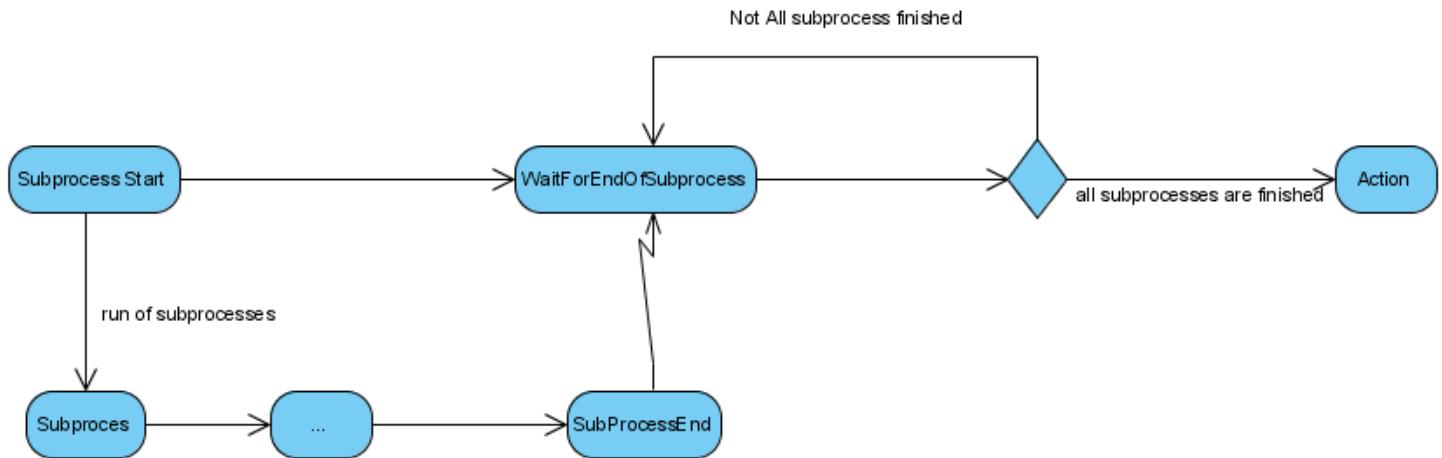
## Retries

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:



## 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.

### 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.value = 'modified';
            process.contextParameters.set(0, parameter);
        </script>
    </scriptAction>
</process>
  
```

```

        parameter.setValue('changedFromScript');
        modelService.save(parameter);
        'itworks'
    </script>
    <transition name='itworks' to='success' />
</scriptAction>
<end id='success' state='SUCCEEDED'>Everything was fine</end>
</process>

```

## Related Information

[Business Process Management](#)

# Starting a processengine Process with a Service Activator

From within a Spring Integration Message Channel you often need to call SAP Commerce services. Likewise, you need to be able to publish messages to the Spring Integration channel from a service. This tutorial shows you how to use the Messaging Gateway to start a pipeline, and how to use a Message Transformer and a Service Activator to start a process from within Spring Integration.

You should already be acquainted with:

- Spring Integration
- Enterprise Integration Patterns

## Objectives

Let us consider a simple interface with **sayHello** method:

**HelloWorldService.java**

```

public interface HelloWorldService
{
    void sayHello(final String toWhom);
}

```

If you call the method with the following parameter:

```
helloWorldService.sayHello("you");
```

the following message should be displayed:

Hello, you

Of course you can easily do this by just implementing the interface, but for the sake of this tutorial let us take a slight detour.

## Messaging Gateway

With Spring Integration you do not need to implement your own code to interface the messaging system. You may use the Messaging Gateway, which encapsulates messaging-specific code and separates it from the rest of the application code. You just need to define a dynamic proxy for the interface:

**tutorial-spring.xml**

```

<int:gateway
    id="helloWorldService"

```

```
service-interface="de.hybris.tutorial.HelloWorldService"
default-request-channel="sayHelloChannel"/>
```

It creates a dynamic proxy with the bean id **helloWorldService**, which implements the above interface. When calling the **sayHello** method, the proxy takes the parameter, creates a message with the method parameter as payload, and publishes the message to the **sayHelloChannel**.

The channel is defined like any Spring Integration channel:

#### **tutorial-spring.xml**

```
<int:channel id="sayHelloChannel"/>
```

You may use any type of channel supported by Spring Integration.

## Creating a Business Process with a Transformer

To get the result in form of the printed message, you should create a process using the **processengine** extension. To do it, follow the steps provided in [Creating a Business Process with a Transformer](#).

## Starting the Process with a Service Activator

Now that the process is created, you can start the process with the **BusinessProcessService**. To do this use a Service Activator that is an endpoint connecting the messages on the channel to the service being accessed.

In Spring Integration you may define the Service Activator by means of Spring Expression Language (SpEL). The Service Activator takes the message from the **sayHelloProcessChannel** and passes it over to the SpEL expression:

#### **tutorial-spring.xml**

```
<int:service-activator
  input-channel="sayHelloProcessChannel"
  expression="@businessProcessService.startProcess(payload)"/>
```

## Summary

#### **tutorial-spring.xml**

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:int="http://www.springframework.org/schema/integration"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

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

<bean
  id="sayHelloAction"
  class="de.hybris.tutorial.action.SayHelloAction"/>

<bean id="sayHelloProcessFactory"
```

```

<class="de.hybris.tutorial.SayHelloProcessFactory">
<property name="businessProcessService" ref="businessProcessService"/>
</bean>

<int:gateway
  id="helloWorldService" service-interface="de.hybris.tutorial.HelloWorldService"
  default-request-channel="sayHelloChannel"/>

<int:channel id="sayHelloChannel"/>

<int:transformer
  input-channel="sayHelloChannel"
  output-channel="sayHelloProcessChannel"
  expression="@sayHelloProcessFactory.createProcess(payload)"/>

<int:service-activator
  input-channel="sayHelloProcessChannel"
  expression="@businessProcessService.startProcess(payload)"/>

</beans>

```

As you can see, SAP Commerce services may be used out of the box through SpEL expressions or reflection. It is quite a common pattern to use a Transformer to convert the payload to some SAP Commerce-specific object and then pass it to the service. For simple cases you can use SpEL expression for the transformations, too.

## Related Information

<http://static.springsource.org/spring-integration/reference/htmlsingle/> ↗  
<http://www.eaipatterns.com/MessagingAdapter.html> ↗  
<http://www.eaipatterns.com/MessagingGateway.html> ↗

## Creating a Business Process with a Transformer

Follow the steps to create a business process with a transformer.

### Procedure

1. Create a new process type in the `items.xml` file in your extension. In our case its only attribute is called `toWhom`:

```

tutorial-items.xml

<itemtype generate="true"
  code="SayHelloProcess"
  jaloClass="de.hybris.tutorial.jalo.SayHelloProcess"
  extends="BusinessProcess"
  autocreate="true"
>
  <attributes>
    <attribute qualifier="toWhom" type="java.lang.String">
      <persistence type="property"/>
    </attribute>
  </attributes>
</itemtype>

```

2. Define the action bean in the `spring.xml` file in your extension:

```

tutorial-spring.xml

<bean id="sayHelloAction" class="de.hybris.tutorial.action.SayHelloAction"/>

```

3. Implement the action class:

```

SayHelloAction.java

```

```

public class SayHelloAction extends AbstractProceduralAction
{
    @Override
    public void executeAction(final BusinessProcessModel process) throws RetryLaterException, Exc
    {
        final SayHelloProcessModel sayHelloProcess = (SayHelloProcessModel) process;
        System.out.println(">>>>>> Hello " + sayHelloProcess.getToWhom());
    }
}

```

4. Create an XML file for the process definition and specify the process as follows:

`sayHelloProcess.xml`

```

<process
    xmlns="http://www.hybris.de/xsd/processdefinition"
    start="sayHello"
    name="sayHelloProcess"
    processClass="de.hybris.tutorial.model.SayHelloProcessModel">
    <action id="sayHello" bean="sayHelloAction">
        <transition name="OK" to="end"/>
    </action>
    <end id="end" state="SUCCEEDED">Success</end>
</process>

```

5. To create the process a Transformer is used to convert the payload of a message into the format that the next endpoint expects. In this case it is a String used as a value for the **toWhom** attribute of the business process:

`SayHelloProcessFactory.java`

```

public class SayHelloProcessFactory
{
    private BusinessProcessService businessProcessService;

    public SayHelloProcessModel createProcess(final String toWhom)
    {
        final SayHelloProcessModel process = (SayHelloProcessModel) businessProcessService.createPr
            "process_" + System.currentTimeMillis(), "sayHelloProcess");
        process.setToWhom(toWhom);
        return process;
    }

    @Required
    public void setBusinessProcessService(final BusinessProcessService businessProcessService)
    {
        this.businessProcessService = businessProcessService;
    }
}

```

6. Define the Spring bean for the **SayHelloProcessFactory** in the **spring.xml** file in your extension.

`tutorial-spring.xml`

```

<bean id="sayHelloProcessFactory" class="de.hybris.tutorial.SayHelloProcessFactory" >
    <property name="businessProcessService" ref="businessProcessService"/>
</bean>

```

7. Add the Transformer to Spring Integration definition. It takes the message from the **sayHelloChannel**, evaluates the expression that invokes the **createProcess** method on the **sayHelloProcessFactory**, and creates a new message with the result of the evaluation as payload.

`tutorial-spring.xml`

```

<int:transformer
    input-channel="sayHelloChannel"

```

```
output-channel="sayHelloProcessChannel"
expression="@sayHelloProcessFactory.createProcess(payload) " />
```

Note that you could also define the transformer using `ref` and `method` attributes. The former denotes the bean id of the service and the latter the method to invoke in order to transform the payload.

## Example Order Management Business Process

Based on the example provided in the `fulfilmentprocess` extension, here you will find a description of the creation of an order business process routine.

A business process describes a sequence of steps or activities that is followed repeatedly. To manage a process automatically or manually it needs to be identified and defined. This tutorial provides an example for how to create such a process in the scope of order management. As usual the business process is modelled as a flowchart with loops and parallel paths joining a number of actions.

## Defining the Process

In order to define an order process we need to know what the process looks like and what steps are necessary to complete it. This of course also includes the actions to take in case an error occurs during the process.

For the example two processes are defined, a **PlaceOrder** process and a **ConsignmentFulfilment** subprocess. The **PlaceOrder** process represents the overall process, for every consignment created out of an order a **ConsignmentFulfilment** subprocess is activated. The example is based on the template realized in the [yacceleratorfulfilmentprocess Extension](#).

### Place Order

The PlaceOrder process starts with checking an incoming order for payment authorization and fraud. When these are passed, a notification is sent that the order has been placed. Next the order is split (see [Order Splitting](#) for details) into one or more consigments, starting a ConsignmentFulfilment subprocess for every consignment created. The PlaceOrder process then waits for all the subprocesses to report back in order to decide if and when the order process is completed and an according notification can be sent.

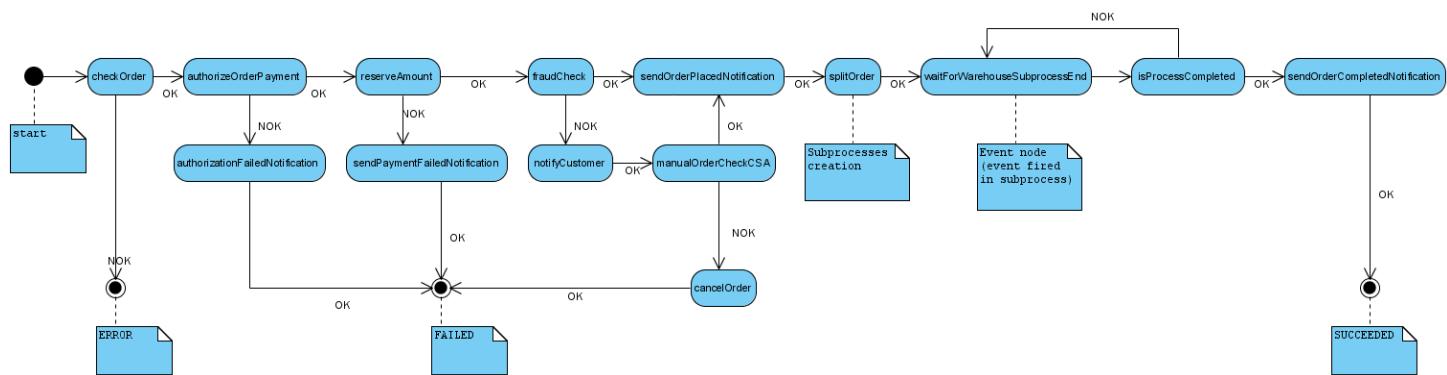


Fig.1 PlaceOrder process diagram

### Consignment Fulfilment

The ConsignmentFulfilment subprocess starts when the consignment is sent to the warehouse through an [interface](#). This can happen immediately an order reaches this state or after certain conditions (amount of consignments, certain time, etc.) have been met, as defined through the `waitBeforeTransmission` step. The subprocess now waits for the warehouse to report back on the consignment's status (cancel/partial/ok) before it takes according action. If the payment succeeds, shipping is allowed and a delivery message is sent. This ends a single subprocess and returns it to the PlaceOrder process.

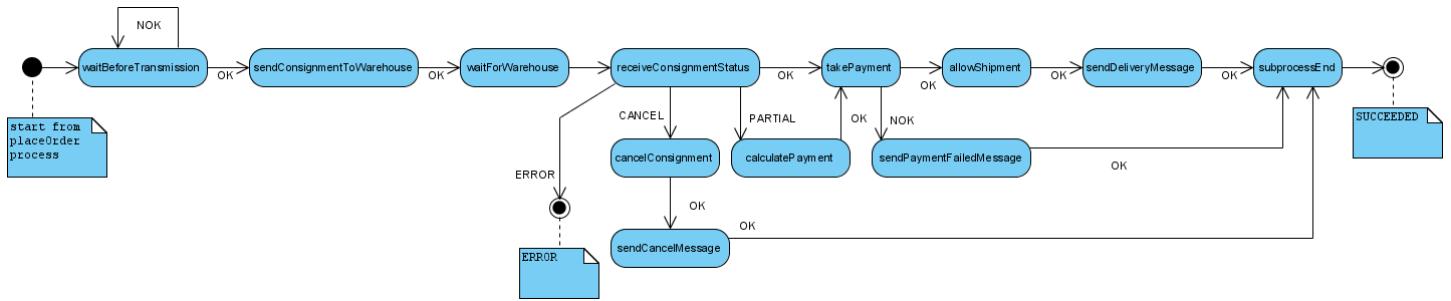


Fig2. ConsignmentFulfilment subprocess diagram

## Process definition in xml

The first step in the process creation is to define it in an xml file. In our case we need to define two processes, the PlaceOrder and the ConsignmentFulfilment process. Note how all the nodes in the process diagrams above are represented in the process definition and indicate the next process steps subject to their results. Detailed information about how a process should be defined is available in the [ProcessEngine documentation](#).

### PlaceOrder Process

```
<?xml version="1.0" encoding="utf-8"?>
<process xmlns="http://www.hybris.de/xsd/processdefinition" >

    <action id="checkOrder" bean="checkOrder">
        <transition name="OK" to="authorizeOrderPayment"/>
        <transition name="NOK" to="error"/>
    </action>

    <action id="authorizeOrderPayment" bean="authorizeOrderPaymer">
        <transition name="OK" to="reserveAmount"/>
        <transition name="NOK" to="authorizationFailedNotification"/>
    </action>

    <action id="reserveAmount" bean="reserveOrderAmount">
        <transition name="OK" to="fraudCheck"/>
        <transition name="NOK" to="sendPaymentFailedNotification"/>
    </action>

    <action id="fraudCheck" bean="fraudCheckOrderInternal">
        <transition name="OK" to="sendOrderPlacedNotification"/>
        <transition name="NOK" to="notifyCustomer"/>
    </action>

    <action id="notifyCustomer" bean="sendFraudErrorNotification'>
        <transition name="OK" to="manualOrderCheckCSA"/>
    </action>

    <action id="manualOrderCheckCSA" bean="fraudCheckOrder">
        <transition name="OK" to="sendOrderPlacedNotification"/>
        <transition name="NOK" to="cancelOrder"/>
    </action>

    <action id="sendOrderPlacedNotification" bean="sendOrderPlace">
        <transition name="OK" to="splitOrder"/>
    </action>

    <action id="cancelOrder" bean="cancelWholeOrderAuthorization'>
        <transition name="OK" to="failed"/>
    </action>

    <action id="authorizationFailedNotification" bean="sendAutho">
        <transition name="OK" to="failed"/>
    </action>

    <action id="sendPaymentFailedNotification" bean="sendPaymentF">
        <transition name="OK" to="failed"/>
    </action>

</process>
```

```

<action id="splitOrder" bean="splitOrder">
<transition name="OK" to="waitForWarehouseSubprocessEnd"/>
</action>

<wait id="waitForWarehouseSubprocessEnd" then="isProcessCompl
<event>ConsignmentSubprocessEnd</event>
</wait>

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

<action id="sendOrderCompletedNotification" bean="sendOrderCo
<transition name="OK" to="success"/>
</action>

<end id="error" state="ERROR">All went wrong.</end>
<end id="failed" state="FAILED">Order not placed.</end>
<end id="success" state="SUCCEEDED">Order placed.</end>

</process>

```

## ConsignmentFulfilment Subprocess

```

<?xml version="1.0" encoding="utf-8"?>
<process xmlns="http://www.hybris.de/xsd/processdefinition" s
processClass="de.hybris.platform.fulfilment.model.Consignment

<action id="waitBeforeTransmission" bean="waitBeforeTransmiss
<transition name="NOK" to="waitForWarehouse"/>
<transition name="OK" to="sendConsignmentToWarehouse"/>
</action>

<action id="sendConsignmentToWarehouse" bean="sendConsignment
<transition name="OK" to="waitForWarehouse"/>
</action>

<wait id="waitForWarehouse" then="receiveConsignmentStatus">
<event>WaitForWarehouse</event>
</wait>

<action id="receiveConsignmentStatus" bean="receiveConsignment
<transition name="OK" to="takePayment"/>
<transition name="PARTIAL" to="calculatePayment"/>
<transition name="CANCEL" to="cancelConsignment"/>
<transition name="ERROR" to="error"/>
</action>

<action id="calculatePayment" bean="calculatePayment">
<transition name="OK" to="takePayment"/>
</action>

<action id="takePayment" bean="takePayment">
<transition name="OK" to="allowShipment"/>
<transition name="NOK" to="sendPaymentFailedMessage"/>
</action>

<action id="allowShipment" bean="allowShipment">
<transition name="OK" to="sendDeliveryMessage"/>
</action>

<action id="sendDeliveryMessage" bean="sendDeliveryMessage">
<transition name="OK" to="subprocessEnd"/>
</action>

<action id="sendPaymentFailedMessage" bean="sendPaymentFailed
<transition name="OK" to="subprocessEnd"/>
</action>

<action id="cancelConsignment" bean="cancelConsignment">
<transition name="OK" to="sendCancelMessage"/>
</action>

```

```

<action id="sendCancelMessage" bean="sendCancelMessage">
<transition name="OK" to="subprocessEnd"/>
</action>

<action id="subprocessEnd" bean="subprocessEnd">
<transition name="OK" to="success"/>
</action>

<end id="error" state="ERROR">All went wrong.</end>
<end id="failed" state="FAILED">Order not placed.</end>
<end id="success" state="SUCCEEDED">Order placed.</end>
</process>

```

## Defining Actions

The next step defines the actions that are specified in the bean attribute of the action nodes. How to design an action foremost depends on what result(s) an action has to be able to provide.

The ProcessEngine extension offers three standard abstract [actions](#) as outlined below. When you have to implement action behavior you can simply create a class that implements the [Action Interface](#), but if you inherit from one of the actions specified below, you will have a set of useful functionality at hand. In our example, all three types of actions are included.

### AbstractProceduralAction (example: SendOrderPlacedNotification)

For some actions there exists only one possible result, on the advent of which the next event in the process chain is triggered. For such an action, the best solution is to define it to inherit from **AbstractProceduralAction**. All you need to do is to implement:

```

@Override
public void executeAction(final BusinessProcessModel process)
{
    //some logic
}

```

### AbstractSimpleDecisionAction (example: FraudCheckOrder)

Perhaps the most common type is the action that has two possible results. Such an action should inherit from the **AbstractSimpleDecisionAction**

```

public Transition executeAction(final BusinessProcessModel process)
{
    //some logic
    return Transition.NOK;
    // or return Transition.OK;
}

```

### AbstractAction (example ReceiveConsignmentStatus)

In this action we have four possible results. Since there is no defined action template for such a case, best practice is to define an enumeration inside the class that defines all possible codes.

```

public enum Transition
{
    OK, PARTIAL, CANCEL, ERROR;

    public static Set<String> getStringValues()
    {
        final Set<String> res = new HashSet<String>();
        for (final Transition t : Transition.values())

```

```
{  
    res.add(t.toString());  
}  
return res;  
}  
}
```

The method `getStringValues()` is a useful method defined in the Action interface and its implementation looks like this

```
@Override  
    public Set<String> getTransitions()  
    {  
        return Transition.getStringValues();  
    }
```

Its implementation is generic, so if you need to define your own transition you can simply copy these lines, redefining only the possible result codes for the action.

All you need to do in action class implementation is to fill in the execute method. Simply put your logic in this method and return one of the results specified in Transition enum.

## Spring integration

Finally you make the process work by defining the spring beans with the names specified in the process definition, and join them with the classes defined in the step before.

```
?xml version="1.0" encoding="UTF-8"?>
<!--
[y] hybris Platform

Copyright (c) 2000-2009 hybris AG
All rights reserved.

This software is the confidential and proprietary information of hybris
("Confidential Information"). You shall not disclose such Confidential
Information and shall use it only in accordance with the terms of the
license agreement you entered into with hybris.
-->

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <!-- Process resourcess definition -->

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

    <bean id="consignmentFulfilmentsubprocess" class="de.hybris.platform.processe...
        <property name="resource" value="classpath:/processdemo/consignmentFulfilment...
    </bean>

    <!-- Actions -->

    <bean id="checkOrderService" class="de.hybris.platform.fulfilment.impl.Default...
        <property name="modelService" ref="modelService"/>
        <property name="processParameterHelper" ref="processParameterHelper"/>
    </bean>

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

    <bean id="cancelOrder" class="de.hybris.platform.fulfilment.actions.CancelOrder...
        <property name="checkOrderService" ref="checkOrderService"/>
    </bean>

    <bean id="cancelOrderAction" class="de.hybris.platform.processengine.action.Abs...
        <property name="modelService" ref="modelService"/>
        <property name="processParameterHelper" ref="processParameterHelper"/>
    </bean>
```

```

<property name="checkOrderService" ref="checkOrderService"/>
</bean>

<bean id="cancelWholeOrderAuthorization" class="de.hybris.platform.fulfilment.parent="abstractAction"/>

<bean id="sendOrderPlacedNotification" class="de.hybris.platform.fulfilment.parent="abstractAction"/>

<bean id="sendPaymentFailedNotification" class="de.hybris.platform.fulfilment.parent="abstractAction"/>

<bean id="fraudCheckOrder" class="de.hybris.platform.fulfilment.actions.Fraud.parent="abstractAction">
<property name="fraudService" ref="fraudService"/>
<property name="providerName" value="Mockup_3rdPartyProvider"/>
</bean>

<bean id="sendFraudErrorNotification" class="de.hybris.platform.fulfilment.parent="abstractAction"/>

<bean id="authorizeOrderPayment" class="de.hybris.platform.fulfilment.actions.parent="abstractAction"/>

<bean id="sendAuthorizationFailedNotification" class="de.hybris.platform.fulfilment.parent="abstractAction"/>

<bean id="reserveOrderAmount" class="de.hybris.platform.fulfilment.actions.Reserve.parent="abstractAction"/>

<bean id="fraudCheckOrderInternal" class="de.hybris.platform.fulfilment.actions.Fraud.parent="abstractAction">
<property name="fraudService" ref="fraudService"/>
<property name="providerName" value="Hybris"/>
</bean>

<bean id="splitOrder" class="de.hybris.platform.fulfilment.actions.SplitOrder">
<property name="orderSplittingService" ref="orderSplittingService"/>
</bean>

<bean id="subprocessesCompleted" class="de.hybris.platform.fulfilment.actions.SubprocessesCompleted.parent="abstractAction"/>

<bean id="sendOrderCompletedNotification" class="de.hybris.platform.fulfilment.parent="abstractAction"/>

<!-- consignmentfulfilment process -->

<bean id="waitBeforeTransmission" class="de.hybris.platform.fulfilment.actions.WaitBeforeTransmission.parent="abstractAction"/>

<bean id="sendConsignmentToWarehouse" class="de.hybris.platform.fulfilment.actions.SendConsignmentToWarehouse.parent="abstractAction">
<property name="process2WarehouseAdapter" ref="process2WarehouseAdapter"/>
</bean>

<bean id="receiveConsignmentStatus" class="de.hybris.platform.fulfilment.actions.ReceiveConsignmentStatus.parent="abstractAction"/>

<bean id="calculatePayment" class="de.hybris.platform.fulfilment.actions.CalculatePayment.parent="abstractAction"/>

<bean id="takePayment" class="de.hybris.platform.fulfilment.actions.ConsignmentTakePayment.parent="abstractAction"/>

<bean id="sendDeliveryMessage" class="de.hybris.platform.fulfilment.actions.ConsignmentSendDeliveryMessage.parent="abstractAction"/>

<bean id="sendPaymentFailedMessage" class="de.hybris.platform.fulfilment.actions.ConsignmentSendPaymentFailedMessage.parent="abstractAction"/>

<bean id="cancelConsignment" class="de.hybris.platform.fulfilment.actions.ConsignmentCancelConsignment.parent="abstractAction"/>

<bean id="sendCancelMessage" class="de.hybris.platform.fulfilment.actions.ConsignmentSendCancelMessage.parent="abstractAction"/>

```

```

<bean id="subprocessEnd" class="de.hybris.platform.fulfilment.actions.consign"
parent="abstractAction"/>

<bean id="allowShipment" class="de.hybris.platform.fulfilment.actions.consign"
parent="abstractAction">
<property name="process2WarehouseAdapter" ref="process2WarehouseAdapter"/>
</bean>

</beans>

```

## Related Information

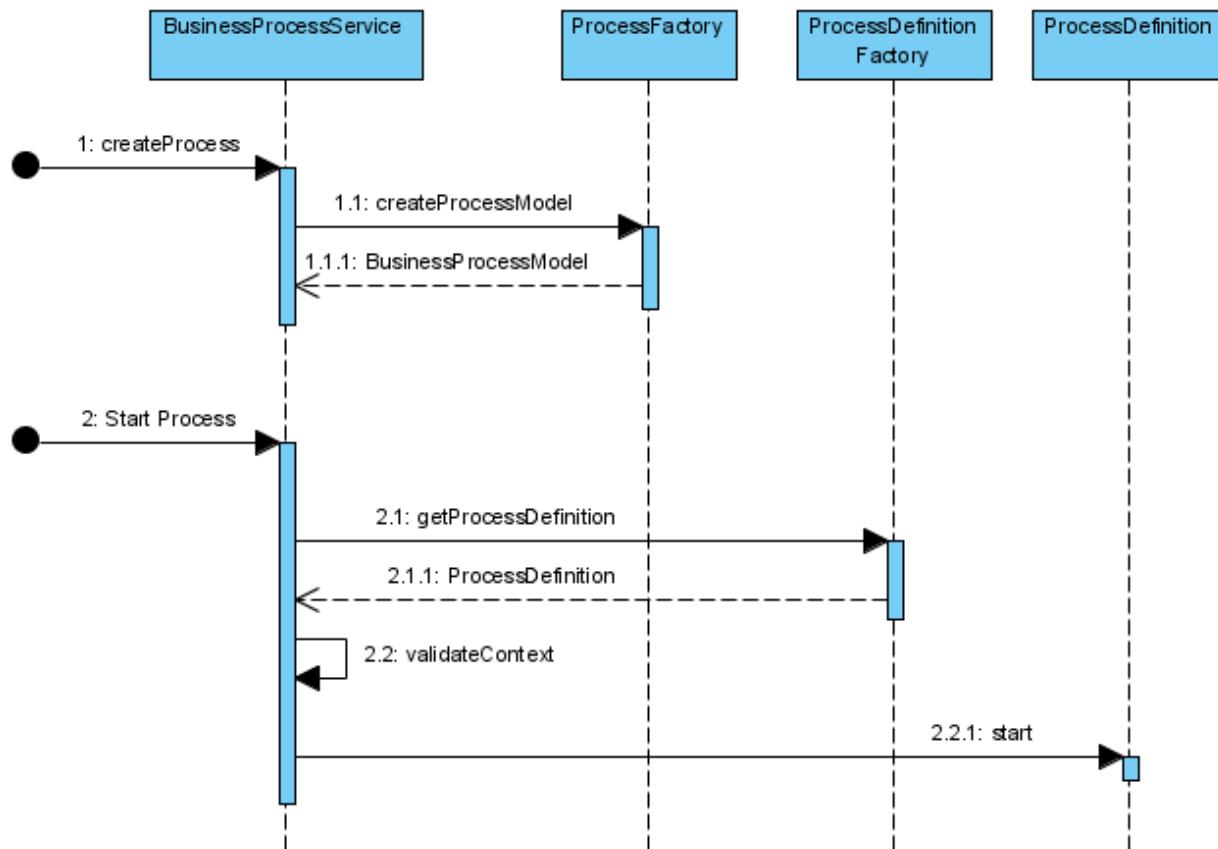
[Order Splitting](#)

[yacceleratorfulfilmentprocess Extension](#)

[The SAP Commerce processengine](#)

## Process Routine of processengine

A new process instance is created by calling the createProcess method from the BusinessProcessService service. You can then run this service using the startProcess method. ProcessDefinitionFactory will create a process definition in this step if not created before.



Process creation.

After a process is created and the startProcess routine invoked (1), we invoke the process definition (1.2.1).

Then (1.2.1.1) getStartNode gives the possibility to get the start node from the process definition by its id. At this the processing of a given process is fired.

First the trigger (1.2.1.1.1) method of the new node is executed. This method invokes scheduleTask from TaskService with proper constraints (date or event name, see [The Task Service](#)). This stores a new Task in a queue and causes the Task Module (depending on specified scheduleTask conditions) to invoke the method run (2.1) on TaskRunner.

The next step begins with run (2.1) on TaskRunner. TaskRunner reads ProcessDefinition (2.1.1), gets from its nodes the one that has to be executed (2.1.2) and then perform the execute routine (2.1.3) on it.

Realization of the execute method depends on the node. For example with an **action** node first the action is performed and then the action result is evaluated to choose one of the transition routes specified in the workflow. **wait** does not simply invoke the trigger method on the node whose id is stated in the **then** attribute but rather waits for a specified event to take place. **end** simply ends the process and does not trigger another node as it is the last in a process.

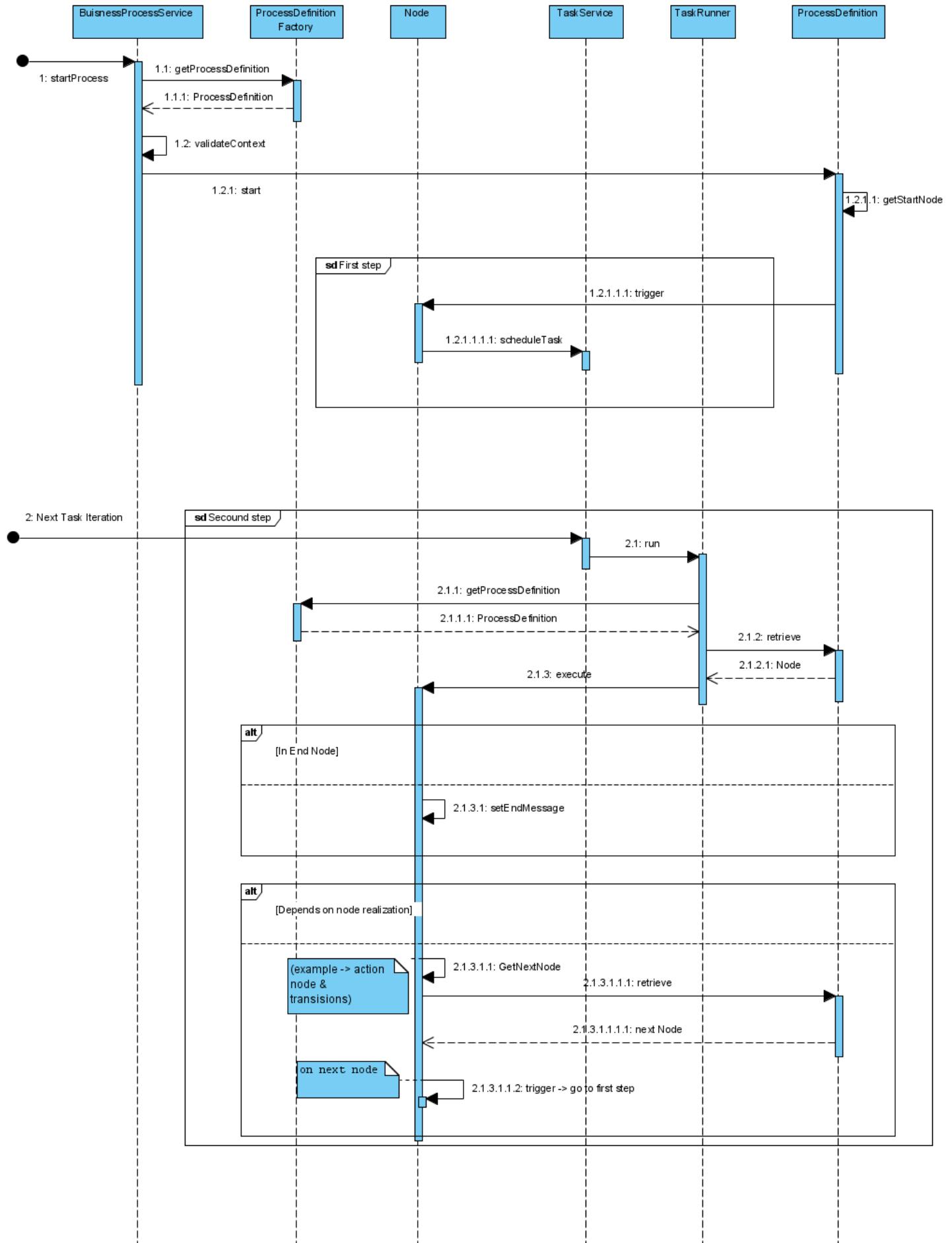


Figure: Process routine.

## Related Information

### [The Task Service](#)

# Executing Business Process Action in a Synchronous Way

Platform runs action nodes in a synchronous manner.

Action nodes in a business process definition can be run asynchronously or synchronously. In the asynchronous mode, an action node is treated as a single and separate task when run by the task engine. In the synchronous mode, an action node is run in the same task as the preceding node.

By default, action nodes are run synchronously. If you wish to disable the synchronous mode, add the following property to your `local.properties` file:

```
processengine.process.canjoinpreviousnode.default=false
```

Running action nodes synchronously meets the following requirements:

- Steps are processed within separate transactions so that any failed steps don't affect previous successful steps.
- After each step, the process and task items are updated in the same way as if they were processed asynchronously. As a result, when the engine crashes, you know what the last successful step was.

For both the synchronous and asynchronous mode, you can decide for any step whether it should be processed synchronously or asynchronously. To run a given step asynchronously, add the `canJoinPreviousNode` element attribute in the process definition in XML, for example:

```
<action id="actionId" canJoinPreviousNode="false">
```

To run a given step synchronously, add the `canJoinPreviousNode` element attribute with the value `true`:

```
<action id="actionId" canJoinPreviousNode="true">
```

Advantages:

- The time delay between performing two actions is small.
- The total time for processing business model is reduced. There is no additional effort required to run the task in the task engine for a synchronous job. This has mainly positive effect when you have a business process model with many actions and the time of a single execution of an action is small.
- On each node, you have the possibility to decide whether you want to perform a node synchronously or asynchronously.

## i Note

### Asynchronous Node Performing Restrictions

- Only action nodes and their script action node extension have the possibility to synchronously join to other node.
- Synchronous execution doesn't apply to the starting action.
- Synchronous execution doesn't apply to performing the same action a few times in a row
- If an error occurs, the process is performed as usual - in an asynchronous way.
- A node with the `canJoinPreviousNode` flag set to `true` is run synchronously only if the previous node is of the action node type.

## Simple Scenario Example

Here is a simple scenario example:

```

<process xmlns="http://www.hybris.de/xsd/processdefinition" start="start" name="example">
    <action id="start" bean="SomeActionBean" canJoinPreviousNode="true">
        <transition name="OK" to="secondStep"/>
    </action>
    <action id="secondStep" canJoinPreviousNode="true">
        <transition name="OK" to="thirdStep"/>
    </action>
    <action id="thirdStep" >
        <transition name="OK" to="fourthStep"/>
    </action>
    <action id="fourthStep" canJoinPreviousNode="true">
        <transition name="OK" to="fifthStep"/>
    </action>
    <action id="fifthStep" canJoinPreviousNode="true">
        <transition name="OK" to="sixthStep"/>
    </action>
    <end id="success" state="SUCCEEDED">Everything was fine</end>
</process>

```

After the process is started, the flow is as follows:

1. The start node has the `canJoinPreviousNode` flag set but according to restriction it is processed asynchronously.
2. `secondStep` is processed synchronously.
3. `thirdStep` is processed synchronously or asynchronously depending on the default configuration set: synchronously for `processengine.process.canjoinpreviousnode.default=true`, asynchronously for `processengine.process.canjoinpreviousnode.default=false`.
4. `fourthStep` is processed synchronously.
5. `fifthStep` is processed synchronously.
6. The next node is the end node.

The process is finished inside the caller thread and returned.

## Execution Logs

Below you can see some execution logs (jdbc statements are enabled) for a business process defined for two actions with synchronous and asynchronous execution between them.

This is the version with a synchronous execution:

### Process definition

```

<process xmlns="http://www.hybris.de/xsd/processdefinition" start="start" name="simpleProcess">
    <action id="start" bean="TestActionBean">
        <transition name="OK" to="secondStep"/>
    </action>
    <action id="secondStep" bean="TestActionBean" canJoinPreviousNode="true">
        <transition name="OK" to="success"/>
    </action>
    <end id="success" state="SUCCEEDED">Everything was fine</end>
</process>

```

### Log

```

1|master|171218-09:25:07:908|1 ms|statement|SELECT * FROM junit_composedtypes WHERE InheritancePathS
1|master|171218-09:25:07:910|1 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_t0
1|master|171218-09:25:07:911|0 ms|statement|SELECT item_t0.PK FROM junit_processes item_t0 WHERE (
1|master|171218-09:25:07:915|1 ms|statement|SELECT * FROM junit_composedtypes WHERE InheritancePathS
1|master|171218-09:25:07:916|0 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_t0
1|master|171218-09:25:07:918|1 ms|statement|SELECT item_t0.PK FROM junit_dynamiccontent item_t0 WHE
1|master|171218-09:25:07:923|1 ms|statement|SELECT * FROM junit_composedtypes WHERE InheritancePathS
1|master|171218-09:25:07:924|1 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_t0
1|master|171218-09:25:07:925|1 ms|statement|SELECT item_t0.p_action FROM junit_tasks item_t0 WHERE

```

```

1|master|171218-09:25:07:929|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM jur
...
1|master|171218-09:25:07:953|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM jur
1|master|171218-09:25:07:954|1 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
1|master|171218-09:25:07:955|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM jur
1|master|171218-09:25:07:955|0 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
1|master|171218-09:25:07:964|1 ms|statement|SELECT * FROM junit_atomictypes WHERE PK=? |SELECT * FRON
1|master|171218-09:25:07:965|1 ms|statement|SELECT * FROM junit_atomictypes WHERE PK=? |SELECT * FRON
1|master|171218-09:25:07:966|1 ms|statement|SELECT * FROM junit_atomictypes WHERE PK=? |SELECT * FRON
1|master|171218-09:25:07:967|1 ms|statement|SELECT * FROM junit_atomictypes WHERE PK=? |SELECT * FRON
1|master|171218-09:25:07:976|1 ms|statement|SELECT currentValue,seriestype,template FROM junit_number
1|master|171218-09:25:07:978|2 ms|statement|UPDATE junit_numberseries SET currentValue = ? WHERE seri
1|master|171218-09:25:07:983|1 ms|statement|INSERT INTO junit_tasks ( hjmpTS,PK,createdTS,modifiedTS,
1|master|171218-09:25:07:986|1 ms|statement|SELECT * FROM junit_enumerationvalues WHERE TypePkString
1|master|171218-09:25:07:990|1 ms|statement|UPDATE junit_processes SET hjmpTS = ? ,modifiedTS=?,p_pr
1|master|171218-09:25:07:992|2 ms|commit||

1|master|171218-09:25:07:997|1 ms|statement|SELECT * FROM junit_processes WHERE PK=?|SELECT * FROM ju
24|master|171218-09:25:07:998|1 ms|statement|SELECT isInitialized FROM junit_metainformations WHERE F
24|master|171218-09:25:08:00|1 ms|statement|SELECT item_t0.PK , hjmpTS FROM junit_tasks item_t0 WHE
24|master|171218-09:25:08:02|1 ms|statement|SELECT item_t0.PK , hjmpTS FROM junit_tasks item_t0 WHE
24|master|171218-09:25:08:03|1 ms|statement|SELECT item_t0.PK , hjmpTS FROM junit_taskconditions ite
28|master|171218-09:25:08:06|0 ms|statement|SELECT * FROM junit_tasks WHERE PK=?|SELECT * FROM junit_
28|master|171218-09:25:08:11|2 ms|statement|UPDATE junit_tasks SET p_runningonclusternode = ? WHERE p
28|master|171218-09:25:08:13|2 ms|statement|SELECT * FROM junit_composedtypes WHERE jaloClassName =
28|master|171218-09:25:08:14|1 ms|statement|UPDATE junit_taskconditions SET p_consumed = ? WHERE p_ta
[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798878335926]] (junit) [ProcessengineTaskRunner] Runni
[m[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798878335926]] (junit) [ProcessengineTaskRunner] Rur
[m28|master|171218-09:25:08:38|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM j
...
28|master|171218-09:25:08:147|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:148|1 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
28|master|171218-09:25:08:149|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:149|0 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
28|master|171218-09:25:08:150|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:151|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:152|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:153|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:155|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:156|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:157|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:158|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
Executing node with ID start for process with code test-proc
[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798878335926]] (junit) [ProcessengineTaskRunner] Execu
[m[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798878335926]] (junit) [ProcessengineTaskRunner] Exe
[m28|master|171218-09:25:08:166|1 ms|statement|SELECT * FROM junit_composedtypes WHERE InheritancePa
28|master|171218-09:25:08:168|0 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_1
28|master|171218-09:25:08:170|1 ms|statement|SELECT item_t0.PK FROM junit_taskconditions item_t0 W
28|master|171218-09:25:08:181|1 ms|statement|UPDATE junit_tasks SET hjmpTS = ? ,modifiedTS=?,p_actio
28|master|171218-09:25:08:194|12 ms|commit||

28|master|171218-09:25:08:197|1 ms|statement|SELECT * FROM junit_tasks WHERE PK=?|SELECT * FROM ju
Executing node with ID secondStep for process with code test-proc
[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798878335926]] (junit) [ProcessengineTaskRunner] Execu
[m[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798878335926]] (junit) [ProcessengineTaskRunner] Exe
[m28|master|171218-09:25:08:204|1 ms|statement|SELECT * FROM junit_enumerationvalues WHERE TypePkSti
28|master|171218-09:25:08:207|1 ms|statement|UPDATE junit_processes SET hjmpTS = ? ,modifiedTS=?,p_er
28|master|171218-09:25:08:210|2 ms|commit||

28|master|171218-09:25:08:213|1 ms|statement|SELECT * FROM junit_processes WHERE PK=?|SELECT * FROM j
28|master|171218-09:25:08:220|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:221|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:222|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:223|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:224|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:226|2 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:227|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:228|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:229|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:230|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:231|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:232|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:232|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:233|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:234|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:235|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:236|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:25:08:237|1 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
28|master|171218-09:25:08:238|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju

```

```

28|master|171218-09:25:08:239|0 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
28|master|171218-09:25:08:250|1 ms|statement|SELECT currentValue,seriestype,template FROM junit_number
28|master|171218-09:25:08:251|1 ms|statement|UPDATE junit_numberseries SET currentValue = ? WHERE seri
28|master|171218-09:25:08:253|1 ms|statement|INSERT INTO junit_tasklogs ( hjmpTS,PK,createdTS,modifiedT
28|master|171218-09:25:08:255|2 ms|commit||
28|master|171218-09:25:08:257|0 ms|statement|SELECT * FROM junit_tasklogs WHERE PK=?|SELECT * FROM ju
28|master|171218-09:25:08:267|1 ms|statement|SELECT item_t0.PK FROM junit_taskconditions item_t0 WHE
28|master|171218-09:25:08:269|1 ms|statement|SELECT tbl_pk FROM (SELECT item_t0.PK as pk FROM junit
28|master|171218-09:25:08:293|1 ms|statement|SELECT tbl_pk FROM (SELECT item_t0.PK as pk FROM junit
28|master|171218-09:25:08:298|1 ms|statement|SELECT * FROM junit_composedtypes WHERE InheritancePath
28|master|171218-09:25:08:300|1 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_1
28|master|171218-09:25:08:301|1 ms|statement|SELECT item_t0.PK FROM junit_medias item_t0 WHERE ( i
28|master|171218-09:25:08:302|1 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_1
28|master|171218-09:25:08:303|1 ms|statement|SELECT item_t0.PK FROM junit_taskconditions item_t0 W
28|master|171218-09:25:08:304|1 ms|commit||
28|master|171218-09:25:08:307|2 ms|statement|SELECT * FROM junit_composedtypes WHERE jaloClassName =
28|master|171218-09:25:08:311|1 ms|statement|SELECT * FROM junit_composedtypes WHERE jaloClassName =
28|master|171218-09:25:08:317|4 ms|statement|SELECT * FROM junit_composedtypes WHERE SuperTypePK=?|SE
28|master|171218-09:25:08:319|2 ms|statement|SELECT item_t0.PK FROM junit_composedtypes item_t0 WHE
28|master|171218-09:25:08:320|0 ms|statement|SELECT * FROM junit_composedtypes WHERE PK=? |SELECT * F
28|master|171218-09:25:08:322|1 ms|statement|SELECT * FROM junit_composedtypes WHERE PK=? |SELECT * F
28|master|171218-09:25:08:323|1 ms|statement|SELECT * FROM junit_composedtypes WHERE PK=? |SELECT * F
28|master|171218-09:25:08:324|1 ms|statement|SELECT * FROM junit_composedtypes WHERE PK=? |SELECT * F
28|master|171218-09:25:08:325|1 ms|statement|SELECT * FROM junit_composedtypes WHERE PK=? |SELECT * F
28|master|171218-09:25:08:326|1 ms|statement|SELECT * FROM junit_composedtypes WHERE PK=? |SELECT * F
28|master|171218-09:25:08:327|1 ms|statement|SELECT * FROM junit_composedtypes WHERE PK=? |SELECT * F
28|master|171218-09:25:08:328|1 ms|statement|SELECT * FROM junit_composedtypes WHERE PK=? |SELECT * F
28|master|171218-09:25:08:331|1 ms|statement|SELECT res.linkPK , res.src, res.tgt FROM ( SELECT ite
28|master|171218-09:25:08:334|1 ms|statement|DELETE FROM junit_props WHERE ITEMPK=?|DELETE FROM junit
28|master|171218-09:25:08:335|1 ms|statement|DELETE FROM junit_props WHERE ITEMPK=?|DELETE FROM junit
28|master|171218-09:25:08:340|1 ms|statement|DELETE FROM junit_aclentries WHERE ItemPK=?|DELETE FROM
28|master|171218-09:25:08:341|1 ms|statement|DELETE FROM junit_tasks WHERE PK = ? AND (sealed IS NULL

```

This is the version with an asynchronous execution:

## Process definition

```

<process xmlns="http://www.hybris.de/xsd/processdefinition" start="start" name="simpleProcess">
    <action id="start" bean="TestActionBean">
        <transition name="OK" to="secondStep"/>
    </action>
    <action id="secondStep" bean="TestActionBean" canJoinPreviousNode="false">
        <transition name="OK" to="success"/>
    </action>
    <end id="success" state="SUCCEEDED">Everything was fine</end>
</process>

```

## Log

```

1|master|171218-09:27:14:532|0 ms|statement|SELECT * FROM junit_composedtypes WHERE InheritancePathS
1|master|171218-09:27:14:534|1 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_t0
1|master|171218-09:27:14:535|1 ms|statement|SELECT item_t0.PK FROM junit_processes item_t0 WHERE (
1|master|171218-09:27:14:538|1 ms|statement|SELECT * FROM junit_composedtypes WHERE InheritancePathS
1|master|171218-09:27:14:539|0 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_t0
1|master|171218-09:27:14:541|1 ms|statement|SELECT item_t0.PK FROM junit_dynamiccontent item_t0 WHE
1|master|171218-09:27:14:545|1 ms|statement|SELECT * FROM junit_composedtypes WHERE InheritancePathS
1|master|171218-09:27:14:546|1 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_t0
1|master|171218-09:27:14:547|1 ms|statement|SELECT item_t0.p_action FROM junit_tasks item_t0 WHERE
1|master|171218-09:27:14:551|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM jur
...
1|master|171218-09:27:14:570|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM jur
1|master|171218-09:27:14:571|0 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
1|master|171218-09:27:14:572|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM jur
1|master|171218-09:27:14:573|1 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
1|master|171218-09:27:14:580|1 ms|statement|SELECT * FROM junit_atomictypes WHERE PK=? |SELECT * FRON
1|master|171218-09:27:14:580|0 ms|statement|SELECT * FROM junit_atomictypes WHERE PK=? |SELECT * FRON
1|master|171218-09:27:14:581|0 ms|statement|SELECT * FROM junit_atomictypes WHERE PK=? |SELECT * FRON
1|master|171218-09:27:14:582|1 ms|statement|SELECT * FROM junit_atomictypes WHERE PK=? |SELECT * FRON
1|master|171218-09:27:14:588|0 ms|statement|SELECT currentValue,seriestype,template FROM junit_number
1|master|171218-09:27:14:590|2 ms|statement|UPDATE junit_numberseries SET currentValue = ? WHERE seri
1|master|171218-09:27:14:594|1 ms|statement|INSERT INTO junit_tasks ( hjmpTS,PK,createdTS,modifiedTS,

```

```

1|master|171218-09:27:14:597|0 ms|statement|SELECT * FROM junit_enumerationvalues WHERE TypePkString
1|master|171218-09:27:14:602|1 ms|statement|UPDATE junit_processes SET hjmpTS = ? ,modifiedTS=? ,p_pr
1|master|171218-09:27:14:604|1 ms|commit||
1|master|171218-09:27:14:609|2 ms|statement|SELECT * FROM junit_processes WHERE PK=?|SELECT * FROM ju
24|master|171218-09:27:14:611|2 ms|statement|SELECT isInitialized FROM junit_metainformations WHERE F
24|master|171218-09:27:14:612|1 ms|statement|SELECT item_t0.PK , hjmpTS FROM junit_tasks item_t0 WHE
24|master|171218-09:27:14:614|1 ms|statement|SELECT item_t0.PK , hjmpTS FROM junit_tasks item_t0 WHE
24|master|171218-09:27:14:615|0 ms|statement|SELECT item_t0.PK , hjmpTS FROM junit_taskconditions it
28|master|171218-09:27:14:619|1 ms|statement|SELECT * FROM junit_tasks WHERE PK=?|SELECT * FROM junit
28|master|171218-09:27:14:623|2 ms|statement|UPDATE junit_tasks SET p_runningonclusternode = ? WHERE
28|master|171218-09:27:14:626|2 ms|statement|SELECT * FROM junit_composedtypes WHERE jaloClassName =
28|master|171218-09:27:14:627|1 ms|statement|UPDATE junit_taskconditions SET p_consumed = ? WHERE p_1
[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798911103926]] (junit) [ProcessengineTaskRunner] Runni
[m[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798911103926]] (junit) [ProcessengineTaskRunner] Runni
[m28|master|171218-09:27:14:648|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM
...
28|master|171218-09:27:14:707|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:707|0 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
28|master|171218-09:27:14:708|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:709|1 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
28|master|171218-09:27:14:710|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:711|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:712|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:713|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:714|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:715|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:717|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:717|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:717|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
Executing node with ID start for process with code test-proc
[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798911103926]] (junit) [ProcessengineTaskRunner] Execu
[m[36mDEBUG [TaskExecutor-junit-28-ProcessTask [8798911103926]] (junit) [ProcessengineTaskRunner] Execu
[m28|master|171218-09:27:14:729|1 ms|statement|SELECT * FROM junit_composedtypes WHERE InheritancePa
28|master|171218-09:27:14:731|2 ms|statement|SELECT item_t0.p_code , item_t0.RestrictedType , item_1
28|master|171218-09:27:14:732|1 ms|statement|SELECT item_t0.PK FROM junit_taskconditions item_t0 W
28|master|171218-09:27:14:739|1 ms|statement|INSERT INTO junit_tasks ( hjmpTS,PK,createdTS,modifiedTS
28|master|171218-09:27:14:742|1 ms|commit||
24|master|171218-09:27:14:746|1 ms|statement|SELECT isInitialized FROM junit_metainformations WHERE F
24|master|171218-09:27:14:748|1 ms|statement|SELECT item_t0.PK , hjmpTS FROM junit_tasks item_t0 WHE
24|master|171218-09:27:14:750|1 ms|statement|SELECT item_t0.PK , hjmpTS FROM junit_tasks item_t0 WHE
28|master|171218-09:27:14:751|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
24|master|171218-09:27:14:751|1 ms|statement|SELECT item_t0.PK , hjmpTS FROM junit_taskconditions it
28|master|171218-09:27:14:752|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
31|master|171218-09:27:14:752|0 ms|statement|SELECT * FROM junit_tasks WHERE PK=?|SELECT * FROM ju
28|master|171218-09:27:14:753|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:754|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
31|master|171218-09:27:14:754|1 ms|statement|UPDATE junit_tasks SET p_runningonclusternode = ? WHERE
28|master|171218-09:27:14:755|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:755|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
31|master|171218-09:27:14:755|0 ms|statement|UPDATE junit_taskconditions SET p_consumed = ? WHERE p_1
[36mDEBUG [TaskExecutor-junit-31-ProcessTask [8798911136694]] (junit) [ProcessengineTaskRunner] Runni
[m[36mDEBUG [TaskExecutor-junit-31-ProcessTask [8798911136694]] (junit) [ProcessengineTaskRunner] Runni
[m28|master|171218-09:27:14:756|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM
28|master|171218-09:27:14:757|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:758|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:759|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
Executing node with ID secondStep for process with code test-proc
[36mDEBUG [TaskExecutor-junit-31-ProcessTask [8798911136694]] (junit) [ProcessengineTaskRunner] Execu
[m[36mDEBUG [TaskExecutor-junit-31-ProcessTask [8798911136694]] (junit) [ProcessengineTaskRunner] Execu
[m28|master|171218-09:27:14:760|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM
31|master|171218-09:27:14:761|1 ms|statement|SELECT item_t0.PK FROM junit_taskconditions item_t0 W
28|master|171218-09:27:14:762|2 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:763|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
31|master|171218-09:27:14:764|1 ms|statement|SELECT * FROM junit_enumerationvalues WHERE TypePkStrir
28|master|171218-09:27:14:764|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:765|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:766|0 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
31|master|171218-09:27:14:766|0 ms|statement|UPDATE junit_processes SET hjmpTS = ? ,modifiedTS=? ,p_er
28|master|171218-09:27:14:768|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
31|master|171218-09:27:14:768|1 ms|commit||
28|master|171218-09:27:14:769|1 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
28|master|171218-09:27:14:770|1 ms|statement|SELECT REALNAME,LANGPK,TYPE1,VALUESTRING1,VALUE1 FROM ju
28|master|171218-09:27:14:771|1 ms|statement|SELECT * FROM junit_attributedescriptors WHERE PK=? |SEL
31|master|171218-09:27:14:771|1 ms|statement|SELECT * FROM junit_processes WHERE PK=?|SELECT * FROM j
28|master|171218-09:27:14:781|0 ms|statement|SELECT currentValue,seriesType,template FROM junit_number
28|master|171218-09:27:14:783|1 ms|statement|UPDATE junit_numberseries SET currentValue = ? WHERE seri
31|master|171218-09:27:14:786|2 ms|statement|INSERT INTO junit_tasklogs ( hjmpTS,PK,createdTS,modifiedT

```

# The Task Service

The SAP Commerce Task Service enables you to schedule time-based or event-based actions, even in a cluster environment. Compared to the existing **cron job** functionality, **tasks** should be seen as a simple and easy way of achieving the same outcome without the overhead of the additional cron job features.

Tasks are a method of scheduling actions. With tasks, it is possible to define Spring-based actions which are triggered at defined times or upon triggering of certain events. Compared to CronJobs, tasks are a more lightweight scheduling framework, offering only a minimal set of functionality, but greatly reducing the complexity of implementing actions.

# Persistence

Tasks provide **Task** and **TaskCondition** item types, which hold all necessary information about one scheduled execution of the specified action bean. This way all actions to be executed are stored inside the database.

## Scheduling

Tasks offer two ways of scheduling:

- With the **time-based** scheduling, the task item holds the **execution time** at which the **action bean**, is triggered.
- With the **event-based** scheduling, the task holds at least one task condition item to specify the events required to be fulfilled before triggering the action.

Each task may further specify an **expiration time**. This way the system automatically fails the scheduled action if the called for events have not arrived yet.

## Cluster Aware

Tasks process scheduled actions on any available Platform

By spreading action execution across the whole cluster, the task functionality provides a robust and reliant way of action processing. Even if a single node crashes, the overall processing will still keep working. It is also possible to pin a scheduled action down to a specific cluster node. If that is the case, only this specific node will execute it. cluster node. This means that due actions are executed by the first node able to fetch and lock it.

Normally, all Platform cluster nodes automatically start processing due actions. In some cases it may be necessary to exclude some of these nodes from doing so, which is also possible with the task functionality.

There is also a way to disable task processing on per node basis. It is the only supported way to block task processing when using id autodiscovery. For more information about id autodiscovery, see [Cluster Improvements and ID Autodiscovery](#).

### Details

To disable task processing on a node, set the `task.engine.loadonstartup` property to `false` (it is `true` by default).

```
# disable task processing on a given node
task.engine.loadonstartup=false
```

In a most common scenarios this property will be put in `HYBRIS_RUNTIME_PROPERTIES` file, or in `HYBRIS_OPT_CONFIG_DIR`.

### Deprecations

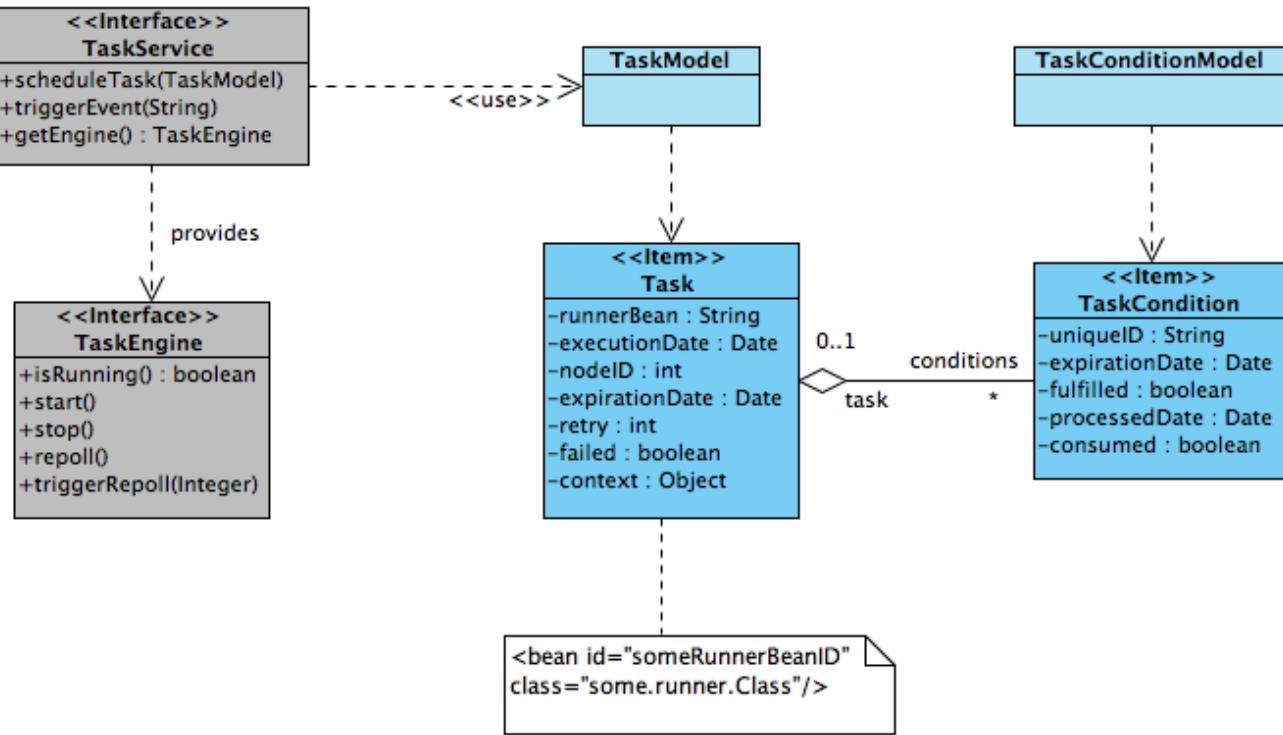
The `task.excluded.cluster.ids` property was the previous way to block task execution on specific nodes. It is still supported provided that you **do not** use new id autodiscovery. If you try to use both, id autodiscovery and `task.excluded.cluster.ids`, you will get a warning and the ids you want to exclude will not be taken into account - tasks will run on all nodes that have `task.processing.enabled` or `task.engine.loadonstartup` set to `true`.

## Spring and ServiceLayer Based

The task functionality was designed to be used in a ServiceLayer-based application right from the start. It provides a service for scheduling new actions and triggering events. Actions are defined as Spring beans inside the usual (core) Spring configuration files, which are provided by each extension.

## Types And Services

The diagram shows all available task types and services.



Note that model classes are not complete since their attributes completely match the underlying item attributes.

## Basic Usage

Actions include your business logic. Learn how to define and schedule them.

### Enabling and Disabling Tasks

To enable or disable the task functionality, use the `task.engine.loadonstartup` property instead of `task.processing.enabled`, which you may have used so far. By default, `task.engine.loadonstartup` is set to `true`, which enables the task functionality. Remember that `task.engine.loadonstartup` takes precedence, so make sure it is set to the same value `task.processing.enabled` was set to earlier.

### Actions

To define an action, it is sufficient to create a class implementing the `TaskRunner` interface. It defines two methods for being triggered and for error handling. The task item model is always provided as invocation context.

```

public class MyTaskRunner implements TaskRunner<TaskModel>
{
    public void run(TaskService taskService, TaskModel task) throws RetryLaterException
    {
        // business code goes here
    }

    public void handleError( TaskService taskService, TaskModel task, Throwable error)
    {
        // this is called if a error occurred or a scheduled action could not be executed
        // in time
    }
}
  
```

This action must be defined inside one extension core Spring configuration file.

```
<bean id="MyRunner" class="MyTaskRunner" >
  <property .../>
</bean>
```

### i Note

Note that putting actions inside a web Spring configuration will **not** work! It is necessary to put them into the core application context (one for each tenant) defined by core Spring configuration files. Of course it is also very easy to wire other required services to these actions simply using Spring.

## Scheduling

Time-based actions are simply scheduled by creating a new TaskModel that points to the requested action bean and holds the correct execution time.

```
ModelService modelService = ...
TaskService taskService = ...

// create model
TaskModel task = modelService.create(TaskModel.class);

// configure it
task.setRunnerBean("MyRunner"); // the action bean name
task.setExecutionDate( new Date() ); // the execution time - here asap

// schedule
taskService.scheduleTask(task);
```

With event-based scheduling, additional item models must be created to specify these events.

```
// create models
TaskModel task = modelService.create(TaskModel.class);
TaskConditionModel cond = modelService.create(TaskConditionModel.class);

// configure them
task.setRunnerBean("MyRunner");
// define event name
cond.setUniqueID("MyEventArrived");
// add to task
task.setConditions( Collections.singleton( cond ) );

// schedule
taskService.scheduleTask(task);
```

The system will now wait until an event with the specified (unique) name has been triggered before the action is executed. In case an expiration time has been specified and the event has not arrived in time, the system will trigger the error handling method of the action bean to signal that this specific task cannot be executed properly any more.

## Triggering Events

Events are generally triggered externally by calling the specific method upon the task service. An event is identified by its unique name. Triggering it requires to pass this unique name - no condition or task item models need to be fetched for that.

```
TaskService taskService = ...

// trigger the event
taskService.triggerEvent( "MyEventArrived" );
```

### i Note

Note that this method will not check if there actually is a matching condition yet. For details, see the Premature Events section.

## Timeouts

Once a task has been scheduled, it resides inside the database until it is processed. Sometimes it may be required to limit the time frame within which it is allowed to be performed in order to avoid working with outdated data. Alternatively, it may be useful to be notified if a scheduled task has been on hold too long, especially if the task is not time based but waits for events to occur.

Therefore, with the task item model it is possible to specify an expiration time after which the system will automatically fetch the task, call the action bean error handling method, and mark the task as failed.

Instead of one timeout per scheduled task, an expiration time can be specified for each condition belonging to a task. This way it is possible to make some event conditions fail earlier than others, according to their expected arrival time.

```
ModelService modeService = ...
TaskService taskService = ...

// create models
TaskModel task = modelService.create(TaskModel.class);
TaskConditionModel cond1 = modelService.create(TaskConditionModel.class);
TaskConditionModel cond2 = modelService.create(TaskConditionModel.class);

// ...
// set single expiration date for the whole task: now + 10min
task.setExpirationDate(new Date( System.currentTimeMillis() + (10 * 60 * 1000) ))

// set expiration date for conditions
cond1.setExpirationDate(new Date( System.currentTimeMillis() + (2 * 60 * 1000) ))
cond2.setExpirationDate(new Date( System.currentTimeMillis() + (6 * 60 * 1000) ))

// ...
```

Have a look at the Error Handling section for instruction on how to get notified that a timeout has occurred.

## Passing Context Data

In previous sections we described actions that were triggered because we specified an execution time or event conditions. This may be sufficient for actions that don't require any information (like performing cleanup jobs) but is not adequate where an action should actually process some previously defined information.

To pass information, the task item model allows us to attach a context object that is serialized and stored inside the database, too.

```
// create model
TaskModel task = modelService.create(TaskModel.class);

// configure it
task.setRunnerBean("MyRunner"); // the action bean name
task.setExecutionDate( new Date() ); // the execution time - here asap
task.setContext( new MyContext(...) );

// schedule
taskService.scheduleTask(task);

// -----
// In action bean: retrieve it
// -----
public void run( TaskService taskService, TaskModel task) throws RetryLaterException
{
    MyContext ctx = task.getContext();
    // business logic
}
```

Often, required data is stored inside an item model. Again, it is possible to attach such an item to the task item model.

```
UserModel requiredUser = ...
// ...
task.setContextItem( requiredUser );
// ...

// -----
// In action bean: retrieve it
// -----
public void run(TaskService taskService, TaskModel task) throws RetryLaterException
{
    UserModel user = (UserModel)task.getContextItem();
    // ...
}
```

## Advanced Concepts

Learn about advanced concepts related to the task functionality, such as transactions, task life cycle, error handling, premature events, and other concepts.

### Transactions

All actions are performed inside their own transaction. This means that changes made inside the action bean `run` method are rolled back in case of an error.

However, in some circumstances it may be required to let a business exception reach the outside but also commit the transaction and deal with the exception outside. Therefore, it is possible to make the task engine not roll back the changes made during a task which failed.

To implement this, add an execute method which allows you to define a set of permitted exceptions that are passed without rolling back the transaction.

```
public void run(TaskService taskService, TaskModel task) throws RetryLaterException
{
    if( ... cannot perform ... )
    {
        ... save some crucial data - we *don't* want to lose them ...
        RetryLaterException ex = new RetryLaterException("cannot perform");
        ex.setRollBack(false); // this tells the engine to commit regardless of the exception
        throw ex;
    }
    else
    {
        // perform...
    }
}
```

### Task Life Cycle

Scheduled actions are stored in the database as task items. These items remain in the database as long as their target action hasn't been performed yet or Retry has been requested (see below).

Once an action has been performed, either successfully or even abnormally, the backing task item is removed from the database.

### Customize Context Data

Although the task item model already enables the user to store a context object or a context item reference, it may sometimes be preferable to declare all required context information as attributes. Therefore, the most flexible solution is to declare a custom task

type.

The main advantage is type safety for reading and writing context data. Also, no separate item is required to carry this information.

```
<itemtype code="NewCustomerTask" jaloclass="NewCustomerTask" extends="Task">
  <attributes>
    <attribute qualifier="firstName" type="java.lang.String" >
      <persistence type="property"/>
    </attribute>
    <attribute qualifier="lastName" type="java.lang.String" >
      <persistence type="property"/>
    </attribute>
    <attribute qualifier="email" type="java.lang.String" >
      <persistence type="property"/>
    </attribute>
    ...
  </attributes>
</itemtype>
```

Scheduling is still simple - only the task model class needs to be changed:

```
ModelService modelService = ...
TaskService taskService = ...

NewCustomerTaskModel task = modelService.create(NewCustomerTaskModel.class);
task.setRunnerBean("NewCustomerTaskRunner");
task.setExecutionDate( new Date() );

// set context information directly
task.setFirstName("foo");
task.setLastName("bar");
task.setLastName("foo.bar@xyz.com");

taskService.scheduleTask(task);

// -----
// In action bean: retrieve it
// -----
public void run(TaskService taskService, TaskModel task) throws RetryLaterException
{
    String firstName = ((NewCustomerTaskModel)task).getFirstName();
    // ...
}
```

## Retry

Sometimes when an action is triggered, it may decide that it cannot perform or complete its work due to temporary problems (for example webservice unreachable). Throwing an error here means that the whole scheduled action is not performed at all and has to be handled (for example restarted) somewhere else.

For such occasions the task functionality provides the **retry** option. By throwing `RetryLaterException`, the action bean is allowed to signal that the current task should be put back into the queue of due tasks to be processed again later. It is also possible to specify the time to wait before the task is fetched again.

```
private static final int MAX_RETRIES = 10;

public void run(TaskService taskService, TaskModel task) throws RetryLaterException
{
    if( ... cannot perform ... )
    {
        if( task.getRetry().intValue() <= MAX_RETRIES )
        {
            RetryLaterException ex = new RetryLaterException("cannot perform");
            ex.setDelay(24 * 60 * 60 * 1000 ); // delay for 24h
            throw ex;
        }
    }
}
```

```

        else
        {
            throw new IllegalStateException("finally cannot perform after "+task.getRetry()+" retries");
        }
    }else
    {
        // perform...
    }
}

```

The sample also illustrates how to use the number of retries in order to avoid endless cycles.

### i Note

Be aware that requesting a retry **does not** extend a task's expiration date (if specified). If it expires while waiting for a retry, it will be processed as failed in the normal way.

## Error Handling

There are a number of errors which may occur and need to be handled:

- A scheduled task has expired before being processed. That may happen if there are still unfulfilled event conditions, or previous attempts to trigger the action ended with `RetryLaterException` and eventually the expiration time has been met. After the expiration time has passed, the system will fetch the task, call the error handling method, and mark it as failed.
- The cluster node shuts down abnormally while processing an action. When that cluster node starts up again, the system will automatically detect which actions have not ended normally, call the error handling method, and mark their task items as failed.
- Of course the action itself might throw an exception. Again, the system will call the error handling method and mark the scheduled task as failed.

Once a task item has been marked as failed, it will never be processed again.

```

public class MyTaskRunner implements TaskRunner<TaskModel>
{
    public void run(TaskService taskService, TaskModel task) throws RetryLaterException
    {
        // ...
    }

    public void handleError(TaskService taskService, TaskModel task, Throwable error)
    {
        if( error instanceof TaskTimeoutException )
        {
            // handle expiration here
        }
        else if( error instanceof InvalidTaskStateError )
        {
            // previous task execution ended due to system failure
        }
        else
        {
            // all other exceptions are coming from action itself !
        }
    }
}

```

It is also possible to add a method which will allow execution of changes made during a task which failed. For details, see [Transactions](#).

## Premature Events

Events are generally triggered from the outside. This may happen by polling some data regularly using CronJobs or by receiving a web service call. Most likely the data is fetched or arrives from an external system.

Occasionally, it may be the case that external data is available and is used for triggering an event **before** there is a matching scheduled action in form of a task item inside the database. In case such a 'premature' event is triggered while there is no task condition waiting for it, the Task stores the incoming event for an unlimited time. The first to-be-scheduled task holding a condition matching the event's unique name consumes the premature event immediately.

### i Note

Note that triggering the same (by unique ID) premature event multiple times before a task is scheduled has no effect other than triggering it only once.

## Engine Management

Normally, the task engine is started and shut down automatically during Platform startup or shutdown. In addition, there is also an administrative API for the task engine.

With this API, it is possible to check whether the engine is currently running, to start or stop it, and even to request to re-poll tasks from the database. Of course these actions should be used with care.

```
TaskService taskService = ...  
  
TaskEngine engine = taskService.getEngine();  
  
// check state  
boolean running = engine.isRunning();  
  
// stop  
engine.stop()  
  
// start  
engine.start()  
  
// repoll task queue on current cluster node  
engine.repoll()  
  
// repoll task queue on a specific cluster node (12)  
engine.triggerRepoll( Integer.valueOf(12) );
```

## Task Scripting

To find out more about task scripting, see [Task Scripting](#).

## Related Information

[Task Scripting](#)

[The Cronjob Service](#)

[The SAP Commerce processengine](#)

## Task Service Properties

There are some parameters that you can use to increase the task service performance or tune up its behavior.

## Optimizing Database Polling

Platform may process tasks very fast and poll the database to retrieve new tasks immediately. Since the task engine queues only a small portion of retrieved tasks, the number of processed tasks per poll is small too. As a result, the task engine performance may drop drastically because it spends most of the time on retrieving another batch of tasks.

You can remedy this by setting the `task.polling.interval.min` property to an appropriate value. This value defines the minimum time interval (in seconds) between subsequent polls.

As a result, within one interval you can process both the queued tasks and then the tasks buffered in the latest poll until the buffer expires. The buffer includes tasks that were retrieved from the database but have not been queued. The size of the buffer is defined by the `task.engine.query.tasks.count` and `task.engine.query.conditions.count` properties. The buffer expiration time is calculated during a database poll. The calculation is based on the `task.polling.interval.min` property value. The default value is `task.polling.interval.min=10`.

### i Note

The value of the `task.polling.interval.min` property can be overridden by the value of `task.queue.size.reporting.interval`. Since every count query requires a queue query, setting `task.queue.size.reporting.interval` to 0 results in a queue query being run at intervals specified by this property.

## Postponing Database Polling

Platform may start to poll the database to retrieve tasks during Tomcat startup even when not all servlet contexts are initialized. To postpone polling the database for tasks until all servlet contexts are started, set the `task.polling.startup.delay.enabled` property to true. The default value of this property is false.

## Querying for Old and New Tasks

There is a distinction introduced between old and new tasks. It allows you to maintain performance by adapting to a situation when there are many frozen tasks (tasks waiting for some condition to be met). If frozen tasks haven't been executed for a considerably long period of time, you could consider them to be old and decide it is not efficient for the task engine to search for them in each query. In such a case, you can use the `task.engine.query.full.interval` property to tune the task engine to search for new tasks more often than for old tasks. The `task.engine.query.full.executiontime.threshold` property serves as a threshold dividing tasks into old and new.

### Threshold Between Old and New Tasks

The `task.engine.query.full.executiontime.threshold` property defines a threshold that divides tasks into new and old. The value of the property is a number of hours counted into the past from now. Platform considers tasks to be old when their execution start time is before the current time, minus the property value. Tasks whose execution start time is after the current time minus the property value are considered to be new. For example, if you set the property to 10, all tasks with execution start time before now, minus 10 hours are considered **old**. Tasks with execution start time scheduled after now, minus 10 hours are considered **new**. If the property value is 0 (default) or negative, all tasks are considered to be new and are always queried:

### Property default value

```
task.engine.query.full.executiontime.threshold=0
```

You can change the value at runtime. It is not necessary to restart the application to apply the new value.

### Interval for Querying Old and New Task

The `task.engine.query.full.interval` property defines how often the task engine queries the full table instead of querying only new tasks.

If you set the property to 0 (default), every query searches for old and new tasks:

#### Property default value

```
task.engine.query.full.interval=0
```

If you set the property to 5, for example, only every 5th query searches for old and new tasks. The other 4 queries search for the new tasks only.

You can change the value at runtime. It is not necessary to restart the application to apply the new value.

## Gathering Statistics on Tasks to be Run

Use the `task.queue.size.reporting.interval` property to gather statistics on the number of queued tasks. As a value of this property, specify the interval of count queries that are run to calculate the number of queued tasks.

No count queries are run if this property is set to the default value of -1. If you want to run a count query with every polling query, set the property to 0.

As count queries require full queue queries, the value of the `task.queue.size.reporting.interval` can override the settings provided by the `task.engine.query.full.interval` property.

## Storing Processing Logs in the Database

With the `processengine.process.log.dbstore.enabled` property set to `true`, the task engine task logger persists a log file in the database. The file contains logs gathered as a result of task engine processing tasks.

By default, the property is set to `true`:

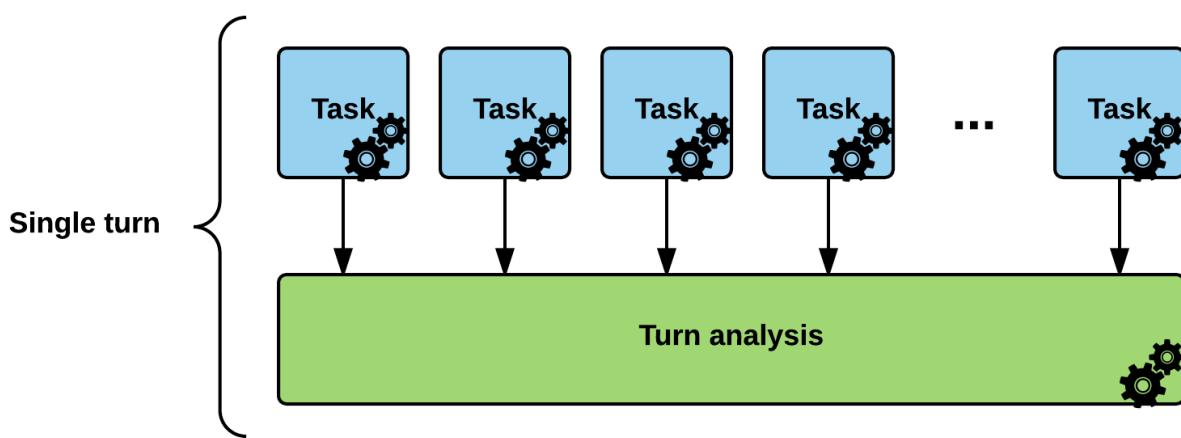
```
processengine.process.log.dbstore.enabled=true
```

You can switch off storing processing logs by setting `processengine.process.log.dbstore.enabled` to `false`.

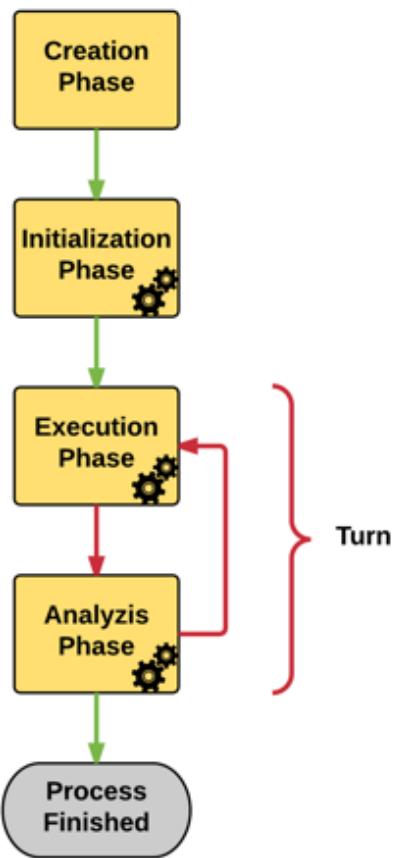
## The Distributed Process Framework

Distributed processing is a solution for leveraging the whole cluster to perform some tasks. The idea behind distributed processing is to split one huge task into smaller tasks called batches and use the task engine to execute them.

To execute operations in parallel, a synchronization point is required to analyze the execution and decide what to do next. To allow those synchronization points, the Distributed Process works in a turn based manner. A single turn consists of multiple tasks executed in parallel and one task waiting for them. The waiting task is responsible for analyzing execution results and for making a decision about further processing. The following diagram shows an overview of a single turn:



The whole execution of a distributed process is divided into four phases. Here is an overview of all the phases and the transitions between them:



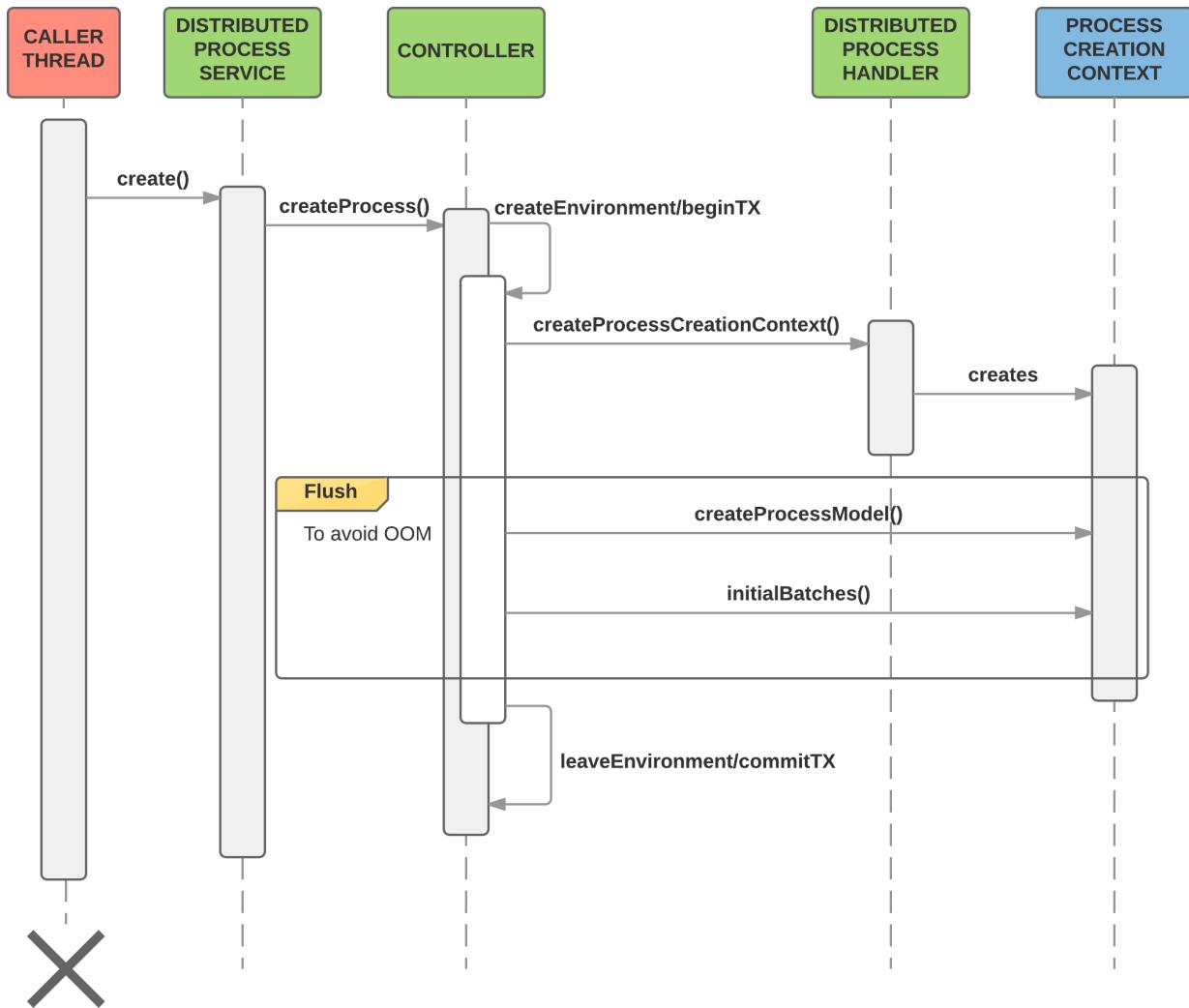
As you can see on the diagram all phases except the creation phase are executed by the task engine and so they can be distributed across cluster nodes. It is important to notice that the execution phase might reoccur. The decision whether to finish or continue execution is taken during the analysis phase.

The Distributed Process is based on SAP Commerce items. It uses the task engine that is also persisted. It allows a distributed process to be continued even after a node failure.

If you want to use distributed processing, all you need to do is to implement `de.hybris.platform.processing.distributed.defaultimpl.DistributedProcessHandler`. This interface allows you to fully control a distributed process. Also see [Simple Template for Distributed Process](#) to find out how to easily implement and set up processing of workload split into batches and distributed across nodes.

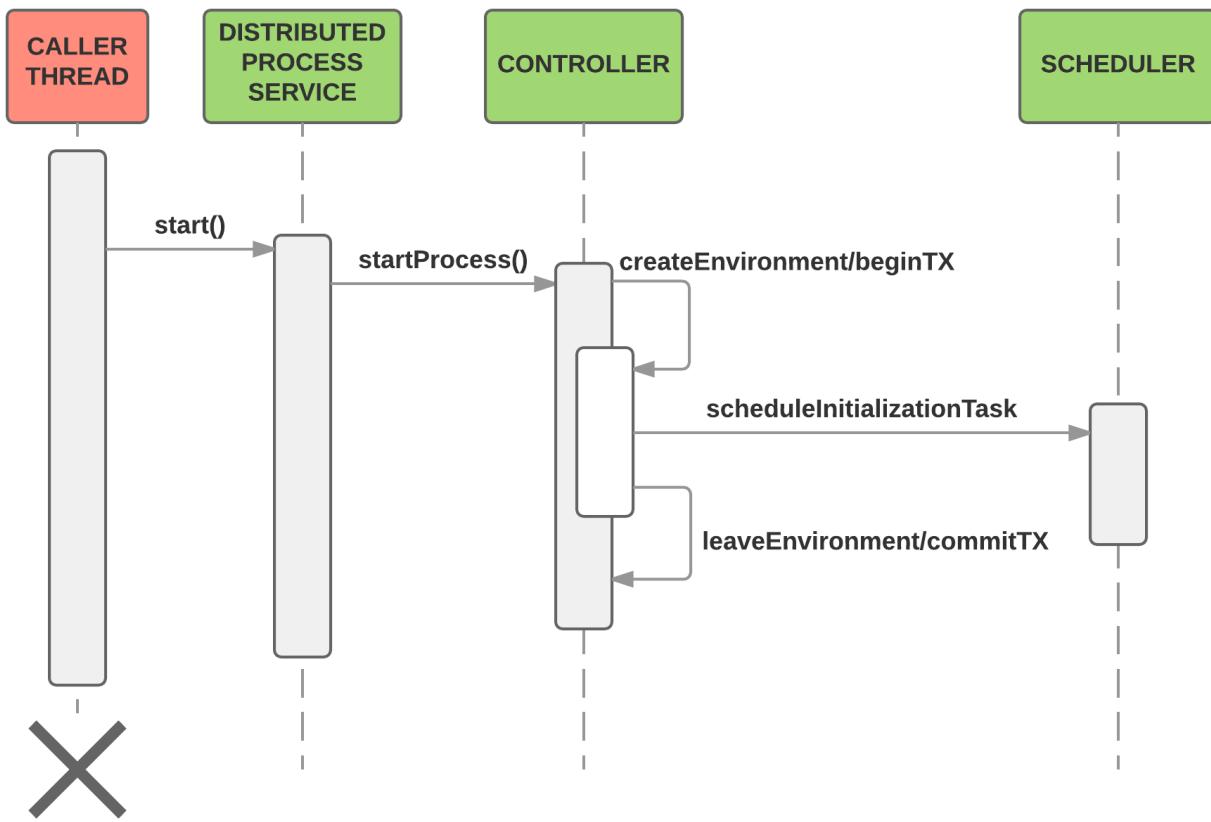
## Creation Phase

The creation phase is the only phase that is executed on the caller thread. It is important (of course if possible) to make it really quick to not block the user thread. Because it is blocking, it is the only phase that can access node specific local environment such as the file system or environment variables. Here is a sequence diagram showing how the process is created:



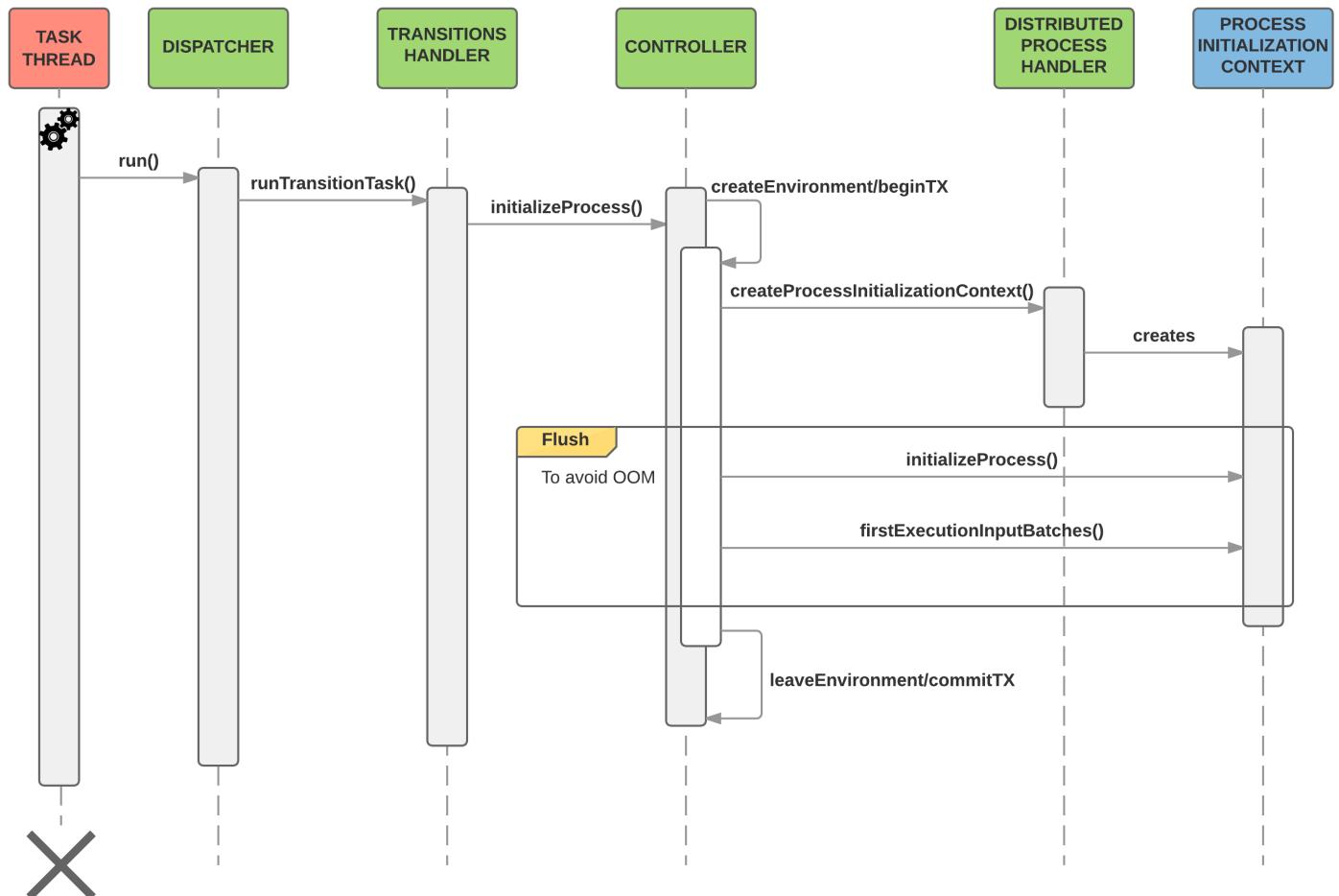
## Starting Process

The starting process is not a separate phase nevertheless it is also important to know how it works.



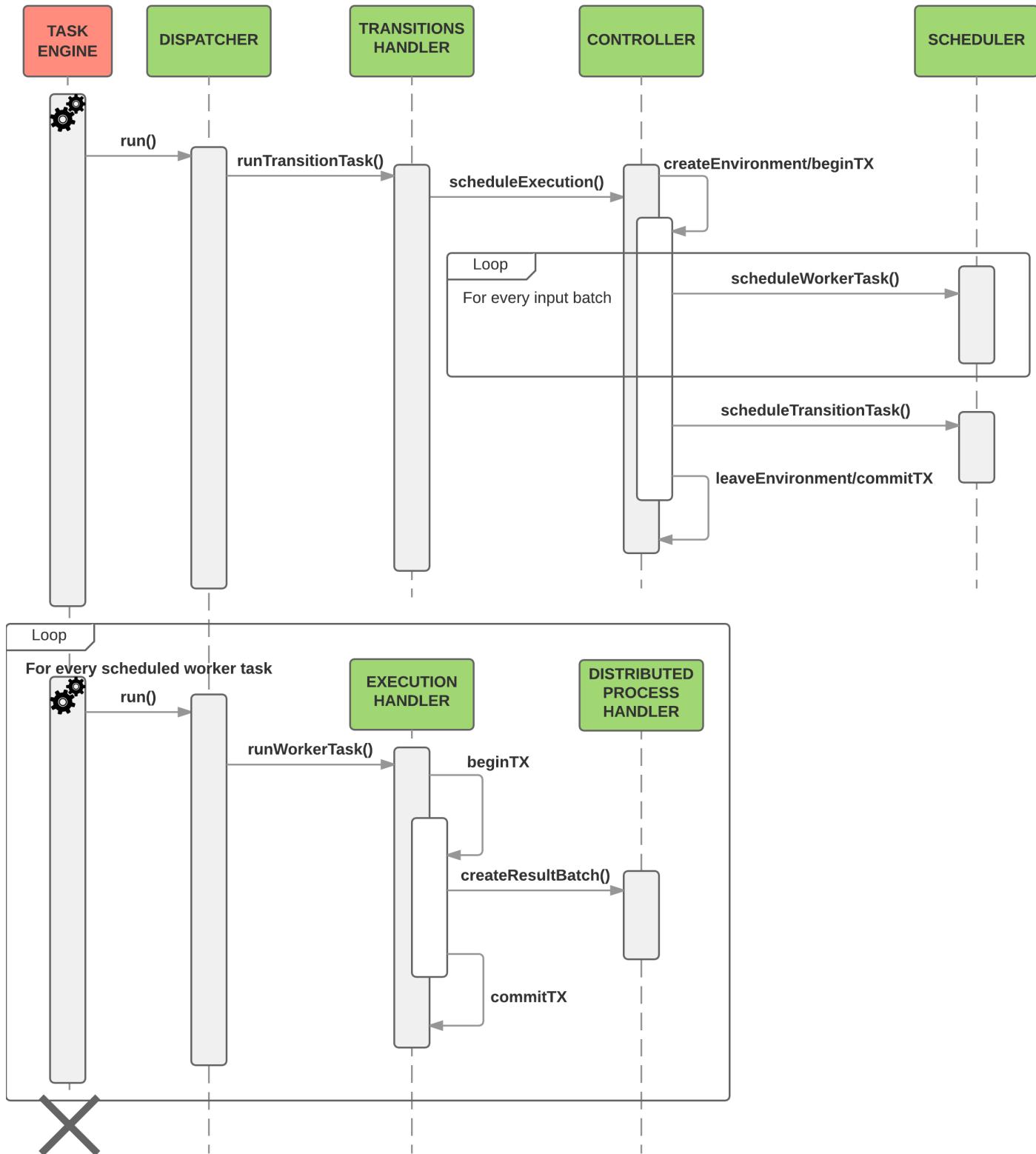
## Initialization Phase

The initialization phase is responsible for creating batches for the first turn of processing.



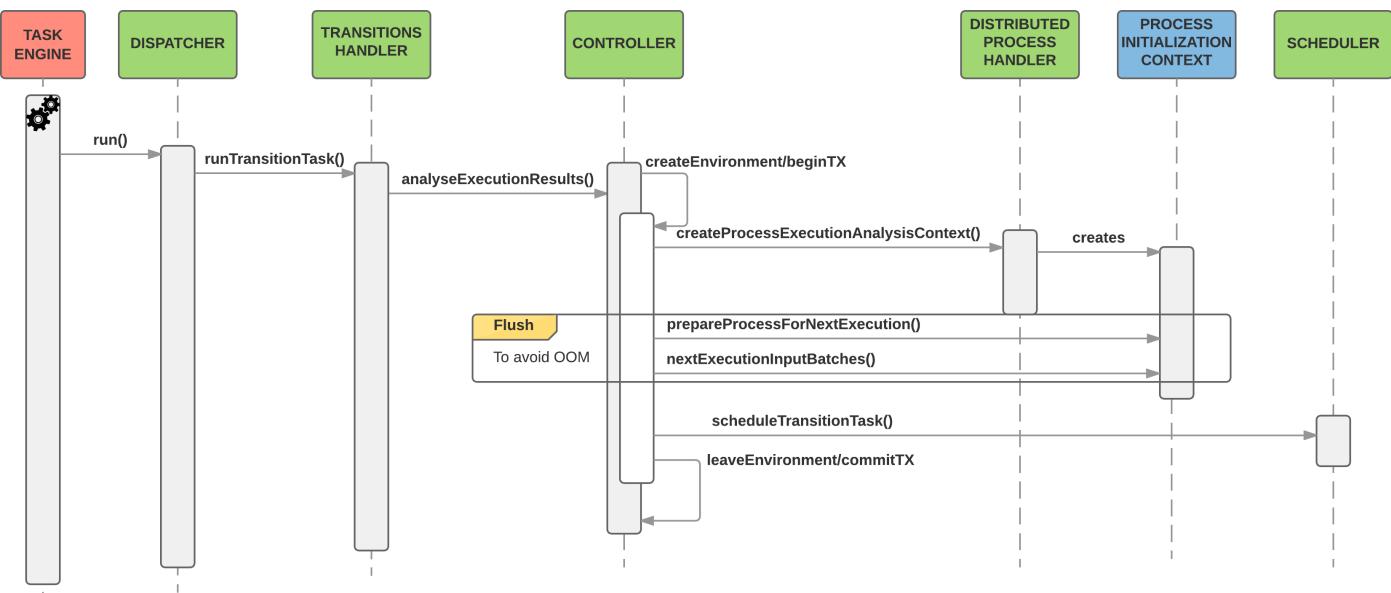
## Execution Phase

The execution phase is responsible for taking all input batches for current turn and producing result batches from them. Each batch is processed in separate task.



## Analysis Phase

Analysis takes place after each turn. It must take a decision whether the process is finished or whether it should be continued in the next turn. If it must be continued, the analysis phase should create, based on the results, input batches for the next execution turn.



## Simple Template for Distributed Process

The Distributed Process framework allows you to process some of your workload on the whole cluster. **Simple Template** for the Distributed Process framework allows you to easily implement and set up workload processing, and use it for your custom applications. It supports processing of workload that can be split into batches and may require possible retries for failed tasks.

### What Simple Template Supports

Simple Template provides a set of technical classes and interfaces that must be extended, implemented, or just declared in a Spring context to work. You can also control the Simple Template default behavior using properties it comes with.

#### Crucial Classes and Interfaces

These are key classes provided with Simple Template:

- **SimpleAbstractDistributedProcessHandler** is a main class that controls the whole flow of a distributed process execution. It is an abstract class that implements **DistributedProcessHandler**. Simple Template provides two concrete classes that extend **SimpleAbstractDistributedProcessHandler**:
  - **SimpleDistributedProcessHandler** provides implementation that operates directly on the **SimpleBatchProcessor** interface. **SimpleBatchProcessor** executes logic for individual batches. Register this class under a unique ID in a Spring context to set up a distributed process.
  - **SimpleScriptingDistributedProcessHandler** provides implementation that executes logic for individual batches from an existing Script via Scripting Engine. It doesn't require any interaction with a Spring context. For details, see [Scripting Engine](#)
- **SimpleBatchProcessor** is an interface that you must implement and inject into **SimpleDistributedProcessHandler** via Spring. It provides logic responsible for executing a single batch of workload.
- **SimpleAbstractDistributedProcessCreationData** is an abstract class and is the basis for the creation of the whole distributed process. It provides information on batch size, number of retries, and actual data for individual batches. Implement at least one method that returns a stream of **SimpleBatchCreationData** objects. Simple Template comes with two default concrete classes that extend **SimpleAbstractDistributedProcessCreationData**:
  - **QueryBasedCreationData** provides implementation that operates on a FlexibleSearch query and builds batches from the result.

- `CollectionBasedCreationData` provides implementation that operates on any Collection and builds batches from elements of that Collection.
- `SimpleBatchCreationData` is a simple POJO class that is a wrapper around the batch context Object. A context Object is a real input data required for a particular, single batch execution logic. Since that Object is persisted in a technical Batch item in a database, it is required that it implements Serializable interface.

## Items

Simple Template comes with two framework items extended from the Distributed Process framework:

- `SimpleDistributedProcess` provides the additional `batchSize` attribute
- `SimpleBatch` provides the additional `resultBatchId`, `retries`, and `context` attributes

## Spring Beans

These are Spring beans provided with Simple Template:

- `abstractSimpleDistributedProcessHandler`

By default, Simple Template has a registered abstract Bean of the `SimpleDistributedProcessHandler` class. It enables you to set concrete Beans of the `SimpleDistributedProcessHandler` class in the most convenient way. All concrete Beans must refer to this Bean as a parent Bean.

- `simpleScriptingDistributedProcessHandler` is responsible for handling execution of a distributed process with use of existing Script that contains business logic for individual batches.

## Configuration

Simple Template comes with two properties that can control the behaviour of a distributed process:

- The `distributed.process.simple.template.max.batch.retries` property sets the number of retries for single Batch execution logic. Each time execution throws an exception, a particular batch is scheduled for a next turn until the max number of retries is reached. After that, the whole process is considered as failed. The property default value is 3.
- The `distributed.process.simple.template.batch.size` property sets a batch size. The default value is 100.

## QueryBasedCreationData

For convenience, Simple Template comes with an easy to use implementation of `SimpleAbstractDistributedProcessCreationData` called `QueryBasedCreationData`. This implementation allows you to use FlexibleSearch to query the database for any result and then divide it into batches and build a Stream of `SimpleBatchCreationData` objects for further processing. By default, query to the database is executed without pagination, so the whole result is read into the memory and then divided into batches internally. However, you can use the database pagination mechanism. To enable it, use the `useDatabasePaging()` method as follows:

```
final QueryBasedCreationData processData = QueryBasedCreationData.builder(flexibleSearchService) //  
    .withQuery("SELECT {PK} FROM {Title} ORDER BY {code}") //  
    .useDatabasePaging() //  
    .withHandlerId("titlesProcessing") //  
    .build();
```

### Caution

Keep in mind that you are responsible for providing a proper `ORDER BY` clause to the query.

## Hook for QueryBasedCreationData

Sometimes it is useful to execute some logic before running a FlexibleSearch query. For instance, to set some settings into a session, provide a function into `QueryBasedCreationData`. It implements the `QueryHook` interface:

```
final QueryBasedCreationData processData = QueryBasedCreationData.builder(flexibleSearchService) //  
    .withQuery("SELECT {PK} FROM {Title}")  
    .withHandlerId("titlesProcessing") //  
    .withBeforeQueryHook(() -> sessionService.setAttribute("foo", "bar")) //  
    .build();
```

## CollectionBasedCreationData

If you have some Collection of elements you want to process, for instance a List of PKs, it is better to use another implementation called `CollectionBasedCreationData`. It provides a simple batching strategy for any Collection of data:

```
// assuming Title PKs are obtained somewhere earlier in a code  
final List<PK> myTitles;  
final CollectionBasedCreationData testData = CollectionBasedCreationData.builder() //  
    .withElements(myTitles) //  
    .withHandlerId("titlesProcessing") //  
    .build();
```

## Custom Process Model

By default, Simple Template creates and operates on a `SimpleDistributedProcessModel` item as the main process object. Sometimes it is useful to extend that item and enrich it with custom attributes. Both `QueryBasedCreationData` and `CollectionBasedCreationData` provide methods to set a custom process model class. The following example shows how to do it:

```
// Assuming CustomSimpleDistributedProcess is a name of a custom process item  
final QueryBasedCreationData testData = QueryBasedCreationData.builder() //  
    .withQuery("SELECT {PK} FROM {Title}") //  
    .withHandlerId("testSimpleDistributedProcessHandler") //  
    .withProcessModelClass(CustomSimpleDistributedProcessModel.class) //  
    .build();  
  
final CustomSimpleDistributedProcess process = distributedProcessService.create(processData);  
// now you have possibility to set some custom attributes before process start  
process.setCustomField("foo");  
// you need to save it  
modelService.save(process);  
// run as usual  
distributedProcessService.start(process.getCode());
```

# Using Simple Template

Learn how to use Simple Template for Distributed Process by following provided example.

In the example, we want to:

- Query the database for a list of some Items
- Execute some complicated computation on data that comes with each instance of that Item
- Remove all the Items we found

We should expect the query to return millions of records. Searching the database in our case would be an expensive process. This is why we'd like to leverage the whole cluster to do the work fast.

## Set Up Your Distributed Process

First, we decide on how to query an Item, and what is the safest and minimum set of data we want for a single batch execution. Remember that between turns, a distributed process stores data in a Blob object in a database, so it must be serializable. For simplicity, we use a List of PKs, and the title item. The `QueryBasedCreationData` class helps us to set up the whole process. By default, `QueryBasedCreationData` produces exactly a List of PKs as a result of the query. First, we implement the logic that is to be executed on a single instance of our `TitleModel`. Since `SimpleBatchModel` has a context attribute, we can get it and simply cast to the List of PKs, and finally convert it to the `TitleModel`. Then we can play with it. This is an example implementation of the `SimpleBatchProcessor` interface we have to provide:

```
public class ItemProcessingTestBatchProcessor implements SimpleBatchProcessor
{
    private ModelService modelService;

    @Override
    public void process(final SimpleBatchModel inputBatch)
    {
        final List<PK> pk = asListOfPk(inputBatch.getContext());
        pk.stream().map(pk -> modelService.<TitleModel> get(pk)).forEach(title -> {
            // Do some very complicated logic here with a TitleModel
            modelService.remove(title);
        });
    }

    private List<PK> asListOfPk(final Object ctx)
    {
        Preconditions.checkNotNull(ctx instanceof List, "ctx must be instance of a List");
        return (List<PK>) ctx;
    }

    @Required
    public void setModelService(final ModelService modelService)
    {
        this.modelService = modelService;
    }
}
```

## Register a SimpleDistributedProcessHandler Bean

Next, we register a new bean of the `SimpleDistributedProcessHandler` class in a Spring context, and inject our newly created implementation of the `SimpleBatchProcessor` interface into it:

```
<bean id="titlesProcessing" class="de.hybris.platform.processing.distributed.simple.SimpleDistributedProcessHandler">
    <constructor-arg>
        <bean class="de.hybris.example.ItemProcessingTestBatchProcessor" />
    </constructor-arg>

```

```

<property name="modelService" ref="modelService"/>
</bean>
</constructor-arg>
</bean>
```

## Prepare and Run the Process

To prepare the process, we provide a concrete instance of `SimpleAbstractDistributedProcessCreationData`. We use the `QueryBasedCreationData` class as our implementation. This class comes with a Builder that allows for an easy setup. Since the `QueryBasedCreationData` class needs `FlexibleSearchService` to work, we make sure we have it injected into our class. We start the whole process via `DistributedProcessService`, and wait for a result:

```

final QueryBasedCreationData processData = QueryBasedCreationData.builder() //
    .withQuery("SELECT {PK} FROM {Title}") //
    .withHandlerId("titlesProcessing")
    .build();

final DistributedProcessModel process = distributedProcessService.create();
distributedProcessService.start(process.getCode());
distributedProcessService.wait(process.getCode(), 100);
```

We have now created an instance of the `QueryBasedCreationData` class. We instructed it to use `FlexibleSearch` to query for all existing `title` items in the system. We instructed it also to use the previously registered `SimpleDistributedProcessHandler` under Bean ID `titlesProcessing`. That Bean was set up to use our custom implementation of `SimpleBatchProcessor`. By default, the batch size is set to 100, and the number of retries is 3.

If you think your case is so complicated that `QueryBasedCreationData` is not enough for your needs, you can extend the `SimpleAbstractDistributedProcessCreationData` class and implement your batching strategy yourself. All classes and methods provided by Simple Template for Distributed Process have public or protected access so you can have more control over the whole process. If you think that the problem is even more complicated than just a turn-based process with retries, go for the pure Distributed Process framework and implement it yourself.

### Builder Methods

You can use the following `QueryBasedCreationData`'s Builder methods to have more control over your distributed process:

- `withProcessId(String)` provides a custom process Id
- `withNodeGroup(String)` specifies on which cluster node group you want to execute your process
- `withBatchSize(int)` changes the default batch size
- `withNumOfRetries(int)` changes the number of retries for an individual batch
- `withQueryParams(Map<String, Object>)` passes query parameters for `FlexibleSearch`
- `withResultClasses(List<Class<?>>)` sets up custom result class list; you may find it handy when querying for a few item attributes
- `withProcessModelClass(Class<? extends SimpleDistributedProcessModel>)` sets up a custom process model
- `useDatabasePaging()` instructs `QueryBasedCreationData` to use database level paging
- `withBeforeQueryHook(QueryHook)` provides a custom before-query logic.

See the `de.hybris.platform.processing.distributed.simple.SimpleDistributedProcessIntegrationTest` test class for examples of using `QueryBasedCreationData`.

## Using Scripting

You can set up `QueryBasedCreationData` to use Scripting so that you can write implementation of `SimpleBatchProcessor` interface in Groovy, upload it in runtime, and execute the whole distributed process.

To use Scripting for our example, we first create a script in Groovy:

```
import de.hybris.platform.core.PK
import de.hybris.platform.processing.model.SimpleBatchModel
import de.hybris.platform.processing.distributed.simple.SimpleBatchProcessor

class TestBatchProcessor implements SimpleBatchProcessor {

    def modelService

    public void process(final SimpleBatchModel inputBatch) {
        inputBatch.context.each {
            // Do some very complicated logic here with a TitleModel
            modelService.remove(title)
        }
    }
}

new TestBatchProcessor(modelService: modelService)
```

Next, we save the script in SAP Commerce Administration Console or Backoffice under some code, let's say `removeTitlesProcessor`. Now we set up our distributed process, and run it via Scripting (you can use Administration Console for it too):

```
import de.hybris.platform.processing.distributed.simple.data.*

def data = QueryBasedCreationData.builder().withQuery("SELECT {PK} FROM {Title} ORDER BY {code}").wi
def process = distributedProcessService.create(data)
distributedProcessService.start(process.getCode())
distributedProcessService.wait(process.getCode(), 100)
```

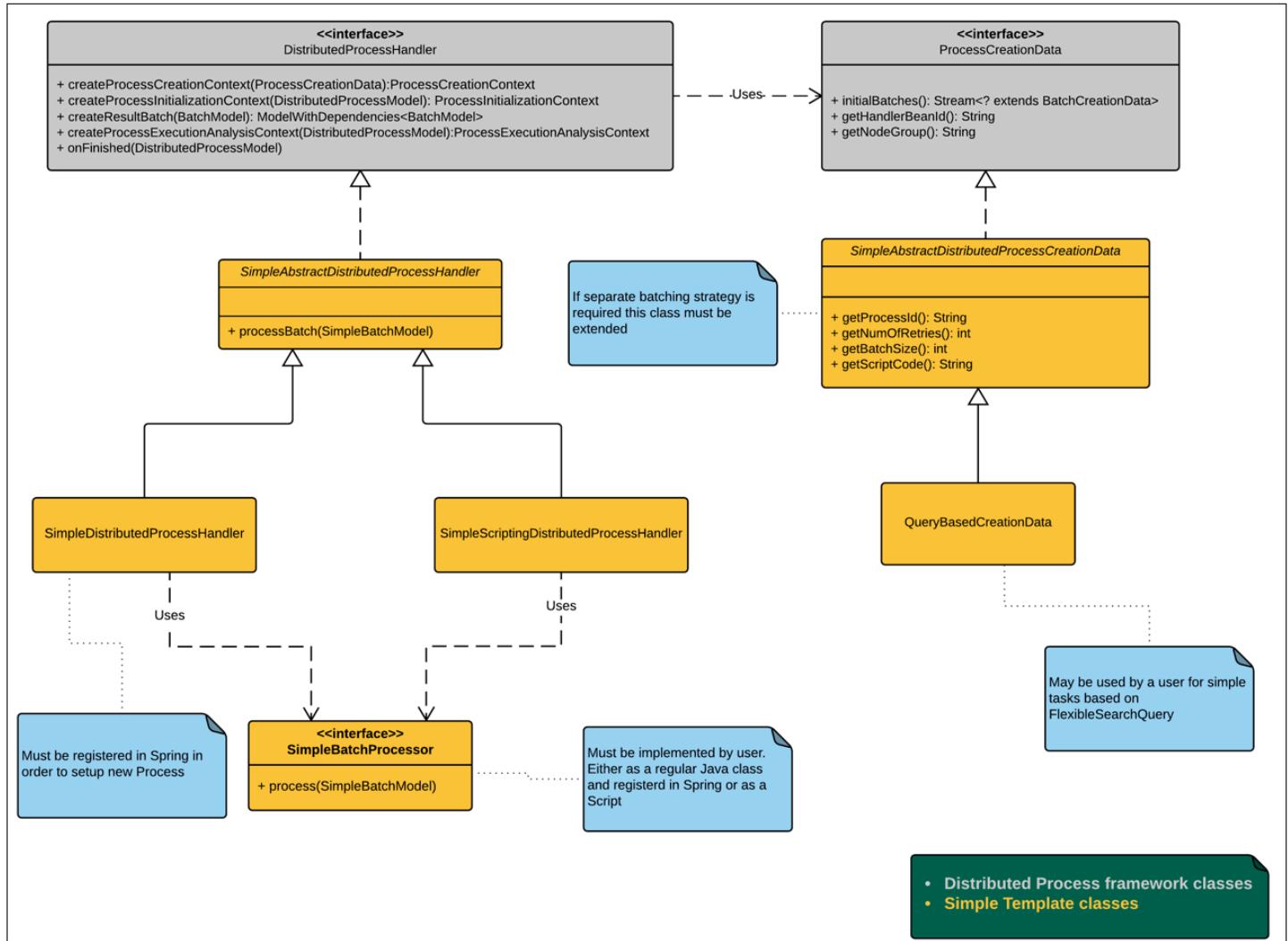
### i Note

Notice that we haven't provided information about which `handlerId` we use when running Scripting based process. `QueryBasedCreationData` knows which handler to choose. Whenever you pass a Script code to it, it automatically uses `SimpleScriptingDistributedProcessHandler`.

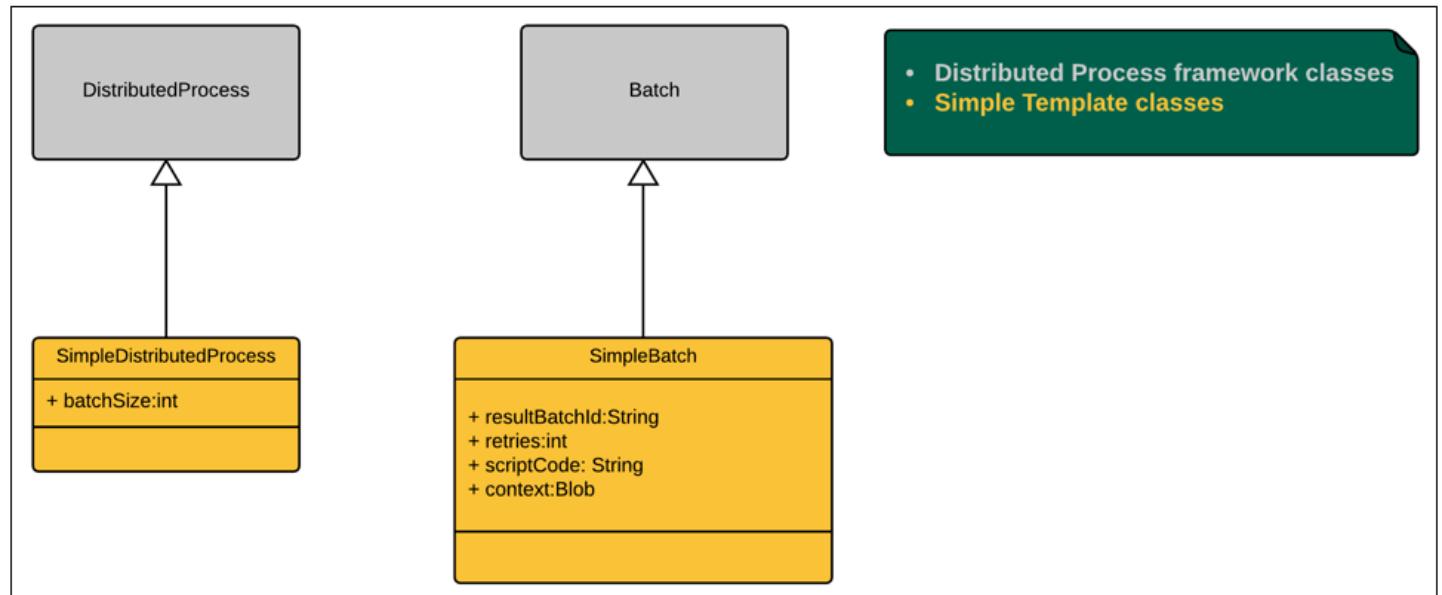
## Technical Overview

See on UML diagrams what Simple Template looks like.

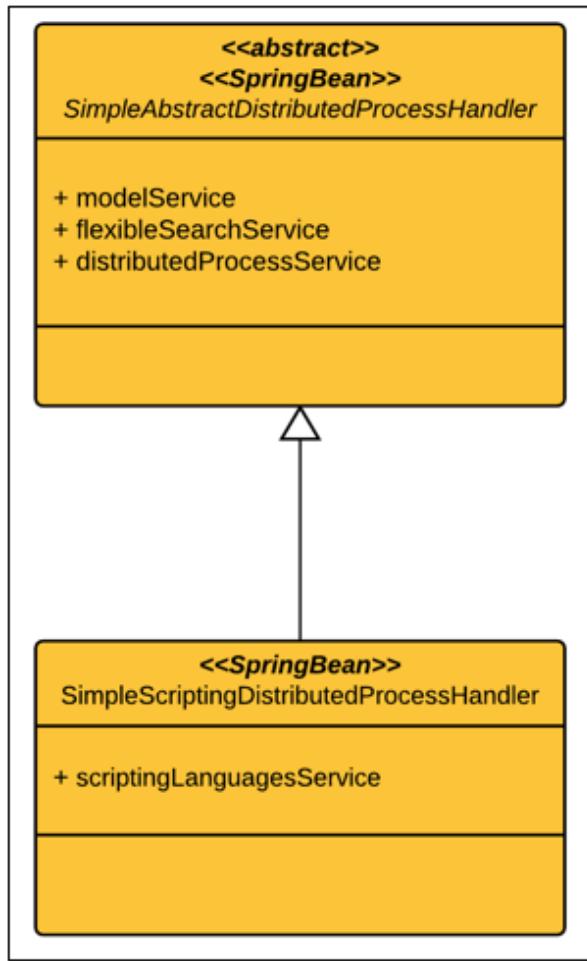
The following diagram shows Distributed Process interfaces (gray) and Simple Template classes that implement them (yellow). It also includes comments on what you need or may implement to set up a distributed process the way we did in our example:



This diagram shows items provided with Distributed Process, and those introduced with Simple Template:



This diagram shows the existing Spring Beans:



## Strategies for Fetching Tasks from the Database

You can create custom strategies for the task engine to fetch tasks from the database. `AuxiliaryTablesBasedTaskProvider` is an example of such a strategy. We created it having a clustered environment in mind.

One of the most crucial mechanisms of the task engine is to obtain tasks to process. While the default strategy meets the needs of most scenarios, there may exist cases for which you would like to change it.

### Creating a Strategy

The logic responsible for fetching tasks from the database is located in a separate Spring bean, and is executed through the `de.hybris.platform.task.impl.TasksProvider` interface.

To create a new strategy, implement the `TasksProvider` interface:

#### MyTaskProvider.java

```

class MyTasksProvider implements TasksProvider
{
    @Override
    public List<VersionPK> getTasksToSchedule(final RuntimeConfigHolder runtimeConfigHolder,
                                              final TaskEngineParameters taskEngineParameters, final int maxItemsToSchedule)
    {
        final List<VersionPK> tasksToProcess = new ArrayList<>();
        // fetch tasks from db
        return tasksToProcess;
    }
}
  
```

```

    }
}
```

Declare it as a Spring bean, and redefine the `tasksProvider` alias:

#### `my-extension-spring.xml`

```

( ... )

<bean id="myTasksProvider" class="MyTasksProvider"/>
<alias name="myTasksProvider" alias="tasksProvider"/>
( ... )
```

## AuxiliaryTablesBasedTaskProvider Strategy for Fetching Tasks

The `AuxiliaryTablesBasedTaskProvider` task fetching strategy is suited for a clustered environment as it solves many obstacles you may find when using the default task engine strategy.

The default task engine strategy for fetching tasks may have some problems when you run SAP Commerce in a cluster with many nodes and there are many tasks waiting for execution. It is so because each node tries to perform the same computation-heavy query to fetch tasks to be processed. With a high number of nodes, you can observe the snowball effect: each node adds its own query, which increases the load on the database and slows the execution of the same query for other nodes. As a result, the query takes longer, and it is more possible to "clutch" with queries for other nodes.

### Cluster-Oriented Design

Having in mind a clustered environment, we created a strategy that is more suited for such scenarios than the default strategy. The main difference is that with `AuxiliaryTablesBasedTaskProvider` each node doesn't store its own queue of tasks to be processed. Instead, the queue is stored in the database in a form of an auxiliary table, and each node fetches tasks from this queue (the query to fetch tasks from the queue is much simpler computation-wise). The queue is filled with an original query, which is executed only by one node at a time, and only if the queue has a low number of tasks waiting to be executed.

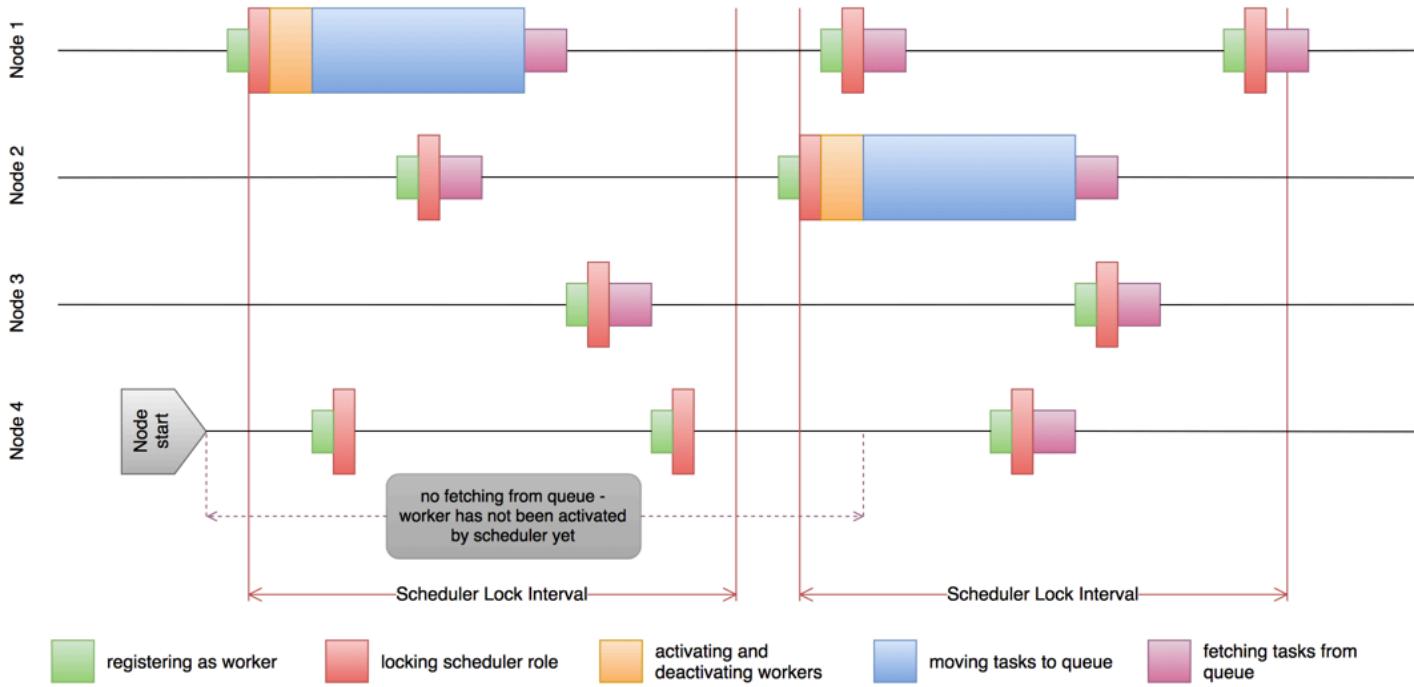
## Auxiliary Tables

In the `AuxiliaryTablesBasedTaskProvider` strategy, the queue of tasks is stored in the database in a form of an auxiliary table.

The design is based on three auxiliary tables, each having different purpose:

- The `tasks_aux_queue` table stores the queue of tasks to be executed. Only tasks that are valid for execution should be in this table.
- The `tasks_aux_workers` table stores the nodes that want to access the queue. This table is used to track whether nodes are active, and what capabilities they have (in regard of `nodeId` and `nodeGroups` set for each worker).
- The `tasks_aux_scheduler` table contains only one row. It is used to lock the scheduler role for only one node at a time. It is also used to determine the version of the database scheme for auxiliary tables. This information is used to recreate the tables when needed.

The diagram shows an example interaction when using the new strategy. We refer to this diagram to explain the details of the design in sections below.



## Upgrading Considerations

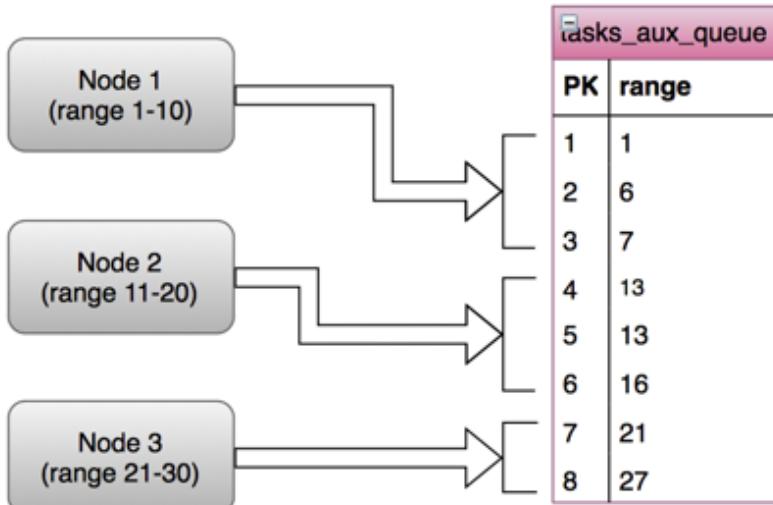
Once you use the `AuxiliaryTablesBasedTaskProvider` strategy, the auxiliary tables are created in the database schema. If you reinitialize such a database with ant, those tables are dropped but aren't re-created.

The `AuxiliaryTablesBasedTaskProvider` auxiliary tables should not be migrated. They are created and completely managed by the scheduling strategy. They are re-created and filled with proper data from the `Tasks` table that is created during initialization.

## Ranges

The `AuxiliaryTablesBasedTaskProvider` design uses the concept of ranges inside a queue of tasks. Ranges help minimize the number of tasks that are scheduled for processing but at the same time have already been processed by other node.

We call such a task a miss. Each task when inserted into a queue, gets a random number assigned to it within a defined range. Then each active worker has a range of numbers assigned. The given worker can only access tasks with the numbers within the worker's range. This way, the tasks in the queue are logically distributed among active workers. As a result, the workers fetch different tasks, and the possibility to get a miss while processing the task is minimized.



# Scheduler Role

The role of a scheduler is crucial for the `AuxiliaryTablesBasedTaskProvider` design to work properly. A scheduler serves a few purposes.

The role of a scheduler is to:

- deactivate workers, for example nodes that have stopped working for various reasons
- activate workers, for example nodes that has just been started
- assign ranges for workers
- move tasks from the task table to the queue

## Fighting To Be a Scheduler

The `AuxiliaryTablesBasedTaskProvider` strategy is based on the concept that only one node can be a scheduler at a given time. Moreover, a scheduler should only perform as often as the `task.auxiliaryTables.scheduler.interval.seconds` parameter allows it to. To achieve this, the time a scheduler last started is stored in the database in the `tasks_aux_master` table. It is fetched by each node that seeks to be a scheduler. Each node checks whether the interval is fulfilled, and if so, it tries to update the row with an optimistic lock pattern. With that, only one node receives the information that the update succeeded - this node assumes it can perform scheduler tasks.

On the diagram above, you can see that only Node 1 and Node 2 succeeded in obtaining the scheduler role. Moreover, Node 2 became the scheduler only when Node 1 finished its role as a scheduler and the interval was long enough.

## Activating / Deactivating Workers, and Assigning Ranges

After obtaining the scheduler role, the node accesses `tasks_aux_table`. It fetches all the workers, and for each it checks whether its heartbeat is still valid. If not, the worker is deactivated. After that, each worker gets a random range assigned that is used to query the tasks in queue. This also activated the workers that were added since the last scheduler's activation.

If a worker is inactive for longer than defined in the `task.auxiliaryTables.worker.removal.interval.seconds` property, it gets removed. The scheduler deletes the entry from the `tasks_aux_workers` table and unlocks all tasks that have been locked for processing in the tasks table (the mechanism of the task engine). For example, in the event of node dying or losing connection to the database, the scheduler automatically recovers tasks that have been locked by the inaccessible node.

On the diagram, Node 4 is not fetching for tasks from the queue because it hasn't been activated by the scheduler. Fetching is enabled only after Node 4 has been activated by Node 2.

## Moving Tasks to Queue

After updating the `tasks_aux_worker` records, the scheduler checks the current state of the queue. It counts all the tasks in the queue grouped by `nodeId` and `nodeGroup`, and calculates the number of tasks in the queue for a given node. It takes into account the exclusive mode of the node and `nodeId` and `nodeGroups` set for each node. If the number of tasks in the queue for any worker is below the threshold then the scheduler moves all valid tasks to the queue. The threshold is calculated by the number of tasks to be fetched by this node multiplied by the `task.auxiliaryTables.worker.lowTasksCountThreshold.multiplier` property value.

# Worker Role

Registering a node as a worker is possible through adding the record in the `tasks_aux_workers` table, or updating its data.

## Registering as Worker

The registration is done with the interval defined by the `task.auxiliaryTables.worker.registration.interval.seconds` parameter. During registration, a node updates its heartbeat and sets its capabilities such as `nodeId`, configured `nodeGroups` to be processed, and whether the node is on the exclusive mode. All this data is later used by the scheduler to determine whether the node has enough tasks scheduled in the queue.

## Accessing Tasks in Queue

Each node accesses tasks within the range defined by the scheduler. This is true for workers that aren't in the exclusive mode. If the worker is configured to be in the exclusive mode, it accesses all tasks (regardless of the defined range) that have the proper `nodeId` or `nodeGroup` set. This can lead to collisions while processing given task by two or more nodes. Because of that, for fetching tasks from the queue, there is an additional optimistic lock mechanism based on the `datetime` column called `PROCESSING_TIME`. A node is only allowed to fetch tasks with the `PROCESSING_TIME` value lower than the current time. Then, for each fetched task its `PROCESSING_TIME` value is increased by the number of seconds determined in `task.auxiliaryTables.tasks.lockDuration.seconds`. This locks the tasks from being fetched by another node for the given time. If the task doesn't get processed within this time, any other node can fetch it and increase the `PROCESSING_TIME` once more.

For a better fetching performance, all columns in the task queue are defined as `NOT NULL` (some databases can't handle indexing rows with `NULL` values). If a task has no `nodeId` defined, the value of the `NODE_ID` column will be `-1`. If there is no `nodeGroup` defined for a task, the value of `NODE_GROUP` will be `---`.

## Properties

See the properties that you can use for a custom task fetching strategy.

Property	Default Value	Description
<code>task.auxiliaryTables.worker.activation.interval.seconds</code>	10	Interval since the current time in which a node heartbeat is treated as valid.
<code>task.auxiliaryTables.worker.registration.interval.seconds</code>	5	Interval (in seconds) at which each update (or insert if not existing) its heartbeat in the auxiliary table. The heartbeat is used to determine whether a node is still online and working.
<code>task.auxiliaryTables.worker.deactivation.interval.seconds</code>	30	Interval after which the node is deactivated. If the node doesn't update the heartbeat during this interval, it doesn't get taken into account when the scheduling takes place.
<code>task.auxiliaryTables.worker.removal.interval.seconds</code>	150	Interval after which the node is removed from the system. If the node doesn't update its heartbeat during this interval, all tasks assigned to the node get unassigned.
<code>task.auxiliaryTables.worker.lowTasksCountThreshold.multiplier</code>	20	Multiplier of the number of tasks that are available in the queue for each node. The base value is equal to <code>task.worker.maxTasks</code> * 2 multiplied by the value of this property. The number of tasks available in the queue for any node is less than the calculated amount, then the scheduler tries to move valid tasks to the queue.

Property	Default Value	Description
task.auxiliaryTables.worker.deleteTasks.maxBatchSize	100	Max number of tasks that can be batched and removed from the queue.
task.auxiliaryTables.scheduler.interval.seconds	10	Interval (in seconds) of how often the scheduler can try to fill the task queue.
task.auxiliaryTables.scheduler.cleanQueue.oldTasksThreshold.seconds	900	Threshold (in seconds) of task wait time in the queue. The tasks that haven't been processed within this time are removed from the queue (if the task is still present in the task table, it gets rescheduled next).
task.auxiliaryTables.tasks.range.start	0	Lowest value of range (inclusive) assigned to the tasks in the queue.
task.auxiliaryTables.tasks.range.end	1000	Highest value of the range (inclusive) assigned to the tasks in the queue.
task.auxiliaryTables.tasks.lockDuration.seconds	20	Number of seconds that a task in the queue can be locked for processing by other workers.
task.auxiliaryTables.worker.tasks.count.multiplier	2	Multiplier of the number of maximum tasks that can be fetched from database in one worker thread. The base value is equal to runningState.getMaxThreads multiplied by the value of this property.
task.auxiliaryTables.delete.schedulerRow.during.update	true	The property decides whether the AuxiliaryTablesBasedTaskProcessor strategy should remove a scheduled task during a system update to prevent task engine locks during an upgrade or newer patch.
task.auxiliaryTables.tasks.get.retriesIfDeadlock	1	In setups where nodeGroups or nodePools are used to pin tasks to a specific node, nodes in large clusters can encounter deadlocks while polling tasks from auxiliary tables. The property defines the additional number of times nodes try to poll for tasks in such situations.  Set this property to 0 or to a negative number if you don't want nodes to retry polling.

## The Cronjob Service

The **cronjob** functionality is used for executing tasks, called cron jobs, regularly at a certain point of time. Typically cron jobs can be used for creating data for backups, updating catalog contents, or recalculating prices.

The key idea of applying cron jobs is to start a long or periodic process in the background, with the possibility to log each run and to easily check its result. The concept of cron jobs in SAP Commerce is explained in detail here.

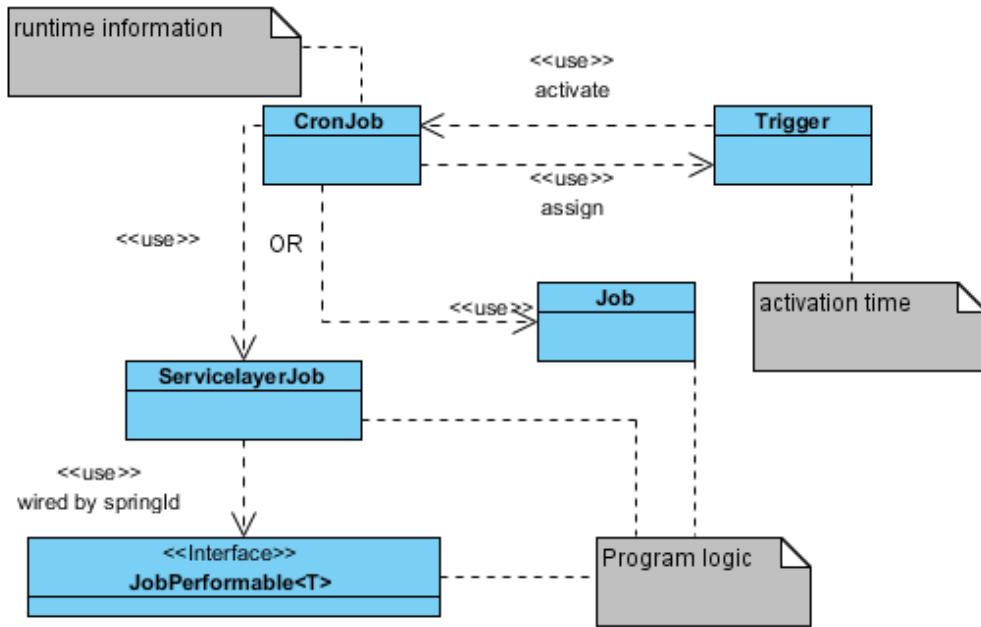
## General Concept

The concept consists of three types that interact with one another: **CronJob**, **Job**, and **Trigger**, as well as the **JobPerformable** class that is tightly related to the **Job** type.

- **Job** type describes the logic to be executed, defined by an associated **JobPerformable**
- **CronJob** type holds the configuration for a single run of a **Job**, as well as the protocol information, like logs
- **Trigger** type is used for scheduling when to run a **Job**.

Summarizing, a **Job** defines what has to be done, a **Trigger** says when, and a **CronJob** specifies the setup used by **Job**.

The division into **CronJob**, **Job/JobPerformable**, and **Trigger** types allows to reuse the code. For example, you may create a database backup as a **Job** several times using **Trigger** at different locations as **CronJob** without coding the actual backup logic twice.



Relations between **CronJob**, **Job**, and **Trigger** types

## CronJob

**CronJob** describes a single run of a **Job**. It contains a mandatory reference to a **Job** instance related to the **CronJob**. **CronJob** allows the definition of several configuration parameters, like session settings, for a specific execution or logging configuration. Furthermore, it stores details of the execution such as start and end time, its result such success or failure, and provides access to the logs.

For more information, see [Clustered Environment](#).

To provide a custom configuration that is specific for an execution, you need to create a subtype of **CronJob**. As a job can be used for several executions, even in parallel, the **Job** should not contain any specific configuration for an execution. It should be specified by the execution-related **CronJob** to keep the **Job** instance stateless. An execution can be started synchronously or asynchronously, and can be aborted on demand if the abort functionality is activated in the implementation of **JobPerformable**.

## Job and JobPerformable

**Job** instance represents one kind of execution logic in the system and each **CronJob** should reference one instance. Although the **Job** instance does not contain any logic, it provides the **springId** attribute, in case of **ServicelayerJob** that references a Spring bean definition of the execution logic. The Spring bean has to implement the **JobPerformable** interface. This is the place where the real logic is performed in a stateless manner. The configuration is passed in by the execution-related instance of **CronJob**. It enables to manage different kinds of logic on item level, while referring to an instance of **JobPerformable** providing the logic.

For more information, see [Writing a Hello World CronJob](#): How to implement a **JobPerformable**

The advantage of splitting the logic into a separate Spring bean is that it is then possible to use the power of the Spring framework. You can inject all loosely coupled dependencies you need into your logic and have the full power of the ServiceLayer available. Nonetheless, it is much simpler to use the **AbstractJobPerformable** that provides a basic functionality for a common **JobPerformable** implementation like request abort handling. Furthermore, the usage of Generics is recommended when implementing **JobPerformable**, to specify the **CronJob** type you expect when calling the implemented **perform** method. The definition of Generics is checked at run time, and it is already assured that the related **Job** can be assigned to the **CronJob** that fulfills the requirement. Think of **JobPerformable** that needs a specific **CronJob** subtype, then simply specify this requirement in the definition of Generic.

To simplify the process of creating a different logic, a **Job** instance is created at the system setup (essential data) for each Spring bean definition that fulfills the **JobPerformable** interface. The created **Job** instance has the same bean id as code.

## Trigger

There is often a need to trigger an execution of a specific logic again, either on a regular basis or only once. To enable it, **Trigger** type references a **Job** that needs to be executed and a cron expression to define the schedule. If the current time fits to the defined cron expression, the referenced **CronJob** or **Job** gets executed automatically. There is always a **Task** item created for every **Trigger**. It is then handled completely by the **TaskEngine** that is polling the Tasks, for example every 10 seconds. It takes care of trigger activation and performing cronjob this way.

For more information, see [Defining a Custom CronJobFactory](#):

When a job is fired by a trigger, a **CronJob** instance is automatically created, because each execution needs a **CronJob** instance to provide the configuration. Be aware that your system might get polluted by this. The **Job** referenced by the **Trigger** is assigned to the **CronJob** and the **CronJob** is executed. In case you wish to use a specific subtype of **CronJob**, simply use the Generics in **JobPerformable**. If you wish to perform some additional logic during the creation of **CronJob**, you can also provide a custom factory. **Trigger** can be also assigned directly to **CronJob** instead of **Job**. However then **CronJob** is reused for each execution and, as a result, the protocol information is overridden.

## CronJob-Related Functionality

### Starting a CronJob

To initiate a **CronJob** execution, pass a **CronJob** instance to the **perform** method of a **CronJobService**:

```
cronJobService.performCronJob(myCronJobModel, true);
```

During the call you can specify if the execution should be forked from the current process or not. If not, the execution logic gets called with the same thread, so be aware that the call can be very time consuming. If the call is done asynchronously, a new thread is created and the calling thread returns immediately.

### Aborting a CronJob

If a **Job** is implemented as abortable, you can abort a running cron job by calling the following method:

```
cronJobService.requestAbortCronJob(myCronJobModel);
```

For more information, see [Writing an Abortable Job](#): How to implement a **Job** as abortable.

### Adjusting the Session for CronJob Execution

A cron job is an automatized task. If it were a human user, it would sit in front of the computer and work through a checklist of steps. Because it works in SAP Commerce, it must run within a certain user context. **CronJob** type has an attribute named **sessionUser** that holds a single user. This user is used for the session, in which the cron job is going to run. By default, the user account set for the

**sessionUser** attribute is the one used for the creation of the cron job. That is, if the **admin** user creates a cron job, it then uses the **admin** user for the session by default. If a cron job is created using the **anonymous** account, the **sessionUser** attribute is set to **anonymous**. A cron job has the same access rights as the user whose account is used for the session. This is both for security and safety reasons. For example, if you create a user with only read access, that user context can back up the database, as the context does not allow the user to manipulate the database. For security reasons, it is recommended to run cron jobs always in the context of a user whose rights are only enough for the task. A consequence of the cron job running in a user context is that it is affected by access rights set for the user account.

For more information, see:

- [Access Rights](#)
- [Defining a Custom CronJobFactory](#)

Other session-related attributes of **CronJob** are **sessionLanguage** and **sessionCurrency**. They use the settings of the **sessionUser** by default.

You can easily modify the attributes before performing a cron job:

```
CronJobModel myCronJob=modelService.create(CronJobModel.class);
// assign Job to CronJob
myCronJob.setSessionUser(mySessionUserModel);
myCronJob.setSessionLanguage(mySessionLanguage);
myCronJob.setSessionCurrency(mySessionCurrency);

modelService.save(myCronjob);
cronJobService.performCronJob(myCronJob);
```

You need to use the logic for an automatically created **CronJob**, which has been triggered inside a custom factory.

All session settings active at the time a **CronJob** gets created are stored at the **CronJob** itself, using **sessionContextValues** attribute. The stored settings are applied at the startup of the automatic job execution. With this, all restrictions and special settings are applied to the execution active at the creation of the **CronJob** creation.

## Checking the Status and Result of a CronJob

A **CronJob** has **status** and **result** attributes:

- **status** indicates the condition in which the cron job is, for example, if it is running or finished
- **result** reports about the last execution of the cron job, for instance, if it has ended with success or failure

To check the result, you can choose one from the following options:

- Access the attributes directly by using:

```
if(myCronJob.getStatus()==CronJobStatus.FINISHED){ ... }
if(myCronJob.getResult()==CronJobResult.SUCCESS){ ... }
```

- Use the getter methods from **CronJobService**, which are available for the most common states, for instance:

```
if(cronJobService.isFinished(myCronJob)){ ... }
if(cronJobService.isSuccessful(myCronJob)){ ... }
```

SAP Commerce contains a **CronJob** that automatically removes all cron jobs with a specified status and a specified result after a given time. Jobs and triggers are not automatically removed and they are kept in the database until removed manually.

## Generic CronJob Creation and Retry Functionality

As mentioned before, the usual way of executing jobs periodically is to define a trigger for the job. When the job gets triggered, a **CronJob** gets created automatically for the execution of the job. The creation is done by a **CronJobFactory**, which enables to customize the generation of **CronJob**, for example by setting custom attributes. Afterward, a predefined map of basic attributes of the triggered **Job** is copied to the **CronJob**, like **sessionUser** or **priority**, if values of those attributes are not filled yet.

For more information, see:

- [Defining a Custom CronJobFactory](#)

If a **Job** is currently not performable, as **JobPerformable.isPerformable** returns false, and the **retry** flag is activated for the **CronJob** or **Job**, then a **Trigger** gets created implicitly. Because the job is not performable at the moment, the trigger will fire the cron job at the next time slot again and again, until it gets executed.

## Running CronJobs in Sequence

The **CompositeCronJob** that is available in SAP Commerce defines any number of entries. Every single entry is represented as a **CompositeEntry**, which can hold either an individual **CronJob** or **Job**. When the **CompositeCronJob** is run, then the individual entries are executed sequentially and independently of one another, in the order specified by the **compositeEntries** attribute. If one entry has finished the execution, the next entry is executed, whether the finished entry has completed successfully or not.

The result of the **CompositeCronJob** depends only on whether exceptions have occurred during its entries' execution. The result of individual entries is not considered for the result of the **CompositeCronJob**:

- If no exceptions occur during the execution of the entries, the **CompositeCronJob** returns **SUCCESS**, even if all results of entries return **FAILURE**.
- If any exceptions occur during the execution of the entries, the **CompositeCronJob** returns **FAILURE**, even if all results of entries return **SUCCESS**.

## CronJobs Across a Clustered System

Cronjobs with triggers are handled completely by the TaskEngine and they don't have to be bound to an individual cluster node. If the **nodeID** is not set for a cronjob (default case) then any cluster node is able to execute it (of course only one at a given time). However, it is still possible to pin a cronjob to some node, which will allow the execution only on that node. You can set it as in the example below.

```
CronJobModel myCronJob= ... //get the CronJob instance
myCronJob.setNodeID(2);
modelService.save(myCronJob);
```

## Running CronJobs Through Ant

SAP Commerce allows running cron jobs using an Ant target: **ant runcronjob**. This command starts an instance of SAP Commerce in a stand-alone mode and executes the **CronJob**. For details, see the Ant target help: **ant -p**

# Job-Related Functionality

## Using Restrictions with Jobs

SAP Commerce allows to use [Restrictions](#) for jobs. The effect is that you can limit the scope of items that the job can access. For example, you can explicitly exclude certain catalog versions from the scope of the **SyncItemJob**, with the effect that the **SyncItemJob** ignores these catalog versions.

The **restriction** attribute of the **Job** type can hold any number of **JobSearchRestrictionModel** objects. Each object holds mainly the type, on which the restriction takes effect and the restricting query.

For more information, see:

- [Restrictions](#)
- [Synchronizing Catalogs](#)

To allow access to attributes of the **Job** and **CronJob**, the **SessionContext** used within the **Job** holds a reference to the **Job** and the **CronJob** via **session.currentJob** and **session.currentCronJob**, respectively. For example, to ensure that a **Job** only uses the set catalog version, you could create a restriction using the following query:

```
{code} IS NULL AND {catalogversion}=?session.currentJob.targetversion
```

## Trigger-Related Functionality

### The Interval for Executions

There is a timer built into the **cronjob** functionality. This timer regularly checks if any trigger should be fired and activates them, if needed. By factory default, the timer is set to 30 seconds. It means that you cannot fire a trigger less than half of one minute apart by default.

If you want to fire a trigger on a more fine scale, you should adjust this interval by overriding the value of the **cronjob.trigger.interval** property in the **local.properties** file:

```
cronjob.trigger.interval=30
```

For more information, see:

- [Configuring the Behavior of SAP Commerce](#): Details on the **local.properties** file
- [Cron Expressions](#)

A database query is performed at each interval to check which triggers should be fired. The value of the interval should not be too low, because you may experience slower system performance due to more frequent database accesses. 30 seconds is a good default value, and we recommend not to change it unless you really need a shorter interval.

### i Note

If you create a trigger, whose activation date is set to a point in the past, the trigger is fired immediately after it is created.

### Automatic Activation of Triggers at Startup

The regular check for triggers is executed for the first time when SAP Commerce is started. If the check detects that the activation date for a trigger is overdue, the trigger is activated immediately. As a consequence, the trigger will fire directly after or possibly during the start of the SAP Commerce. In some situations you may not want to run cron jobs right after the SAP Commerce is started, but instead once a certain time window has expired. For example, if a cron job tidies up the database by 3:30 AM, it may have a very long execution time and also may cause a heavy load on the database. If SAP Commerce has a downtime between 2:30 AM and 4:15 AM, then the trigger is overdue and the cron job is started automatically during the startup of SAP Commerce. It is possible to define a time window for automatic launches of overdue cron jobs. Using the **maxAcceptableDelay** attribute of a **Trigger**, you can set a value in seconds, which defines a threshold of time that is added to an overdue trigger. If the point of time SAP Commerce starts up matches the trigger's execution point of time plus the value of the **maxAcceptableDelay** attribute, then the trigger is activated.

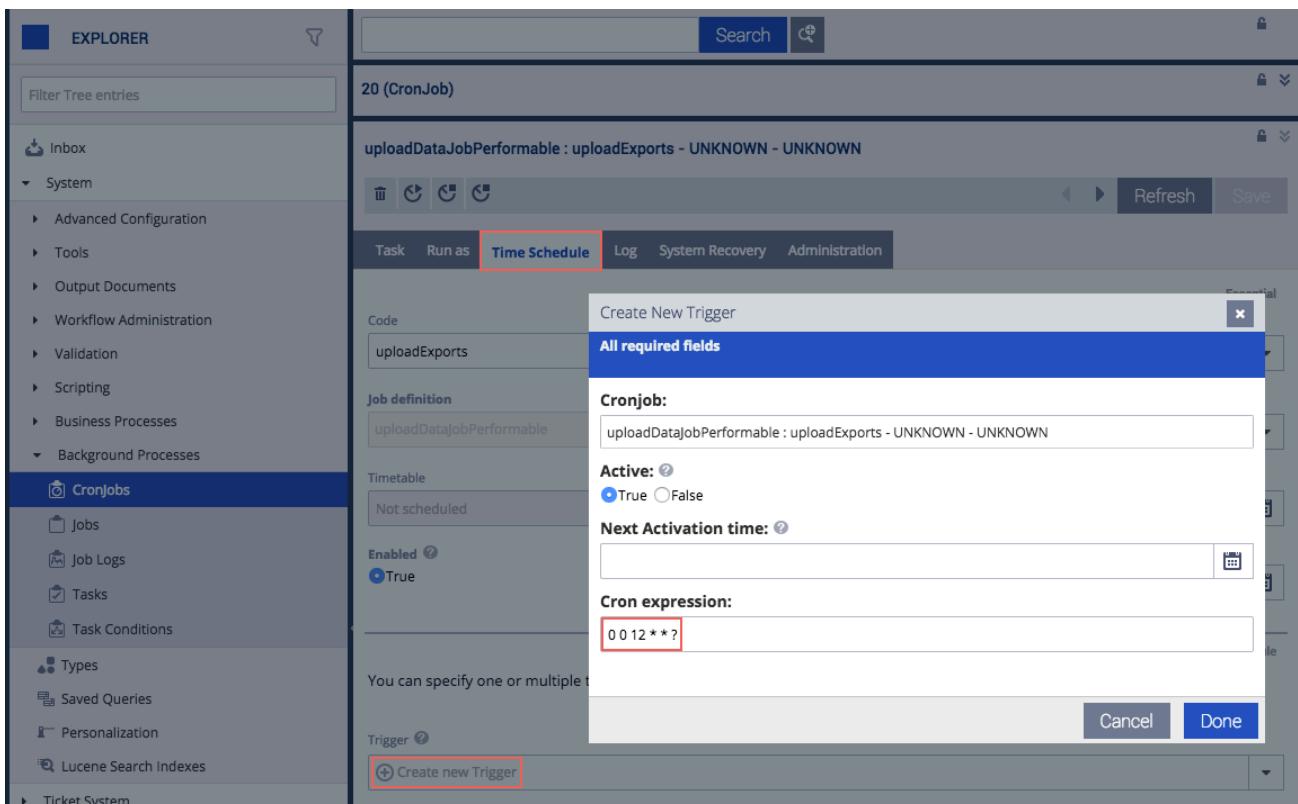
Regular Trigger Execution Time	maxAcceptableDelay Value	Point of Time of the Start-Up	Is the Trigger Activated by Start-Up?
16:00:00	3600 (one hour)	16:30:00	<span style="color: green;">+</span>
16:00:00	3600 (one hour)	17:00:00	<span style="color: green;">+</span>
16:00:00	3600 (one hour)	17:30:00	<span style="color: red;">-</span>
16:00:00	-1 (unlimited)	17:00:00	<span style="color: green;">+</span>
16:00:00	-1 (unlimited)	17:30:00	<span style="color: green;">+</span>

## Creating a Trigger through the Backoffice

This tutorial shows how to create a trigger for a cron job:

1. Log into the Backoffice.
2. In the **Explorer tree**, go to:
  - o **System** tab
  - **Background Processes** tab
  - **Cronjobs** tab
 You are in the cronjob collection browser.
3. Click a chosen cron job to switch to this cron job's editor.
4. In the editor, go to the **Time Schedule** tab.
5. In the **Time Schedule** tab (or other tabs depending on a type of a cron job), click **Create new Trigger** in the **Triggers** field to open the trigger wizard.
6. In the wizard, define a cron expression for your trigger.

We recommend that you schedule triggers by using cron expressions instead of the **Next Activation Time** feature.



7. Click **Done**.

You have completed creating your trigger.

## Cleaning Up CronJobs

It is possible to use the **CleanupCronJobStrategy** to clean up instances of the **CronJob**. This strategy is based on maintenance framework created for processing instances of any type. For more information, see [Creating Strategy to Process Instances](#). Default settings for the cron job cleanup mechanism is to remove instances of the **CronJob**, which fulfill the following:

- Has no trigger
- Are successfully finished
- Are older than 14 days

The sample of **jobs-spring.xml** file below shows the default bean declaration that can be easily customized. See also the Customizing section.

### i Note

The **CleanupCronJobStrategy** replaces the deprecated: **CleanUpJobPerformable** and **CleanUpCronJob**.

## jobs-spring.xml

```
<bean id="cleanupCronJobsPerformable" parent="abstractGenericMaintenanceJobPerformable" >
    <property name="maintenanceCleanupStrategy">
        <bean class="de.hybris.platform.jobs.maintenance.impl.CleanupCronJobStrategy" >
            <property name="modelService" ref="modelService"/>
            <property name="typeService" ref="typeService" />
            <property name="status">
                <set>
                    <value type="de.hybris.platform.cronjob.enums.CronJobStatus">
                </set>
            </property>
            <property name="result">
                <set>
                    <value type="de.hybris.platform.cronjob.enums.CronJobResult">
                </set>
            </property>
            <property name="excludedCronJobCodes" >
                <set/>
            </property>
        </bean>
    </property>
</bean>
```

## Customizing Strategy

The following properties of the **cleanupCronJobsPerformable** bean can be modified in your bean declaration:

- **status**: Here you can configure cleanup behaviour: cronjobs of what status should be handled by cleanup. This property must contain at least one value. You can put multiple values here, like for example:

```
<property name="status">
    <set>
        <value type="de.hybris.platform.cronjob.enums.CronJobStatus">FINISHED</value>
        <value type="de.hybris.platform.cronjob.enums.CronJobStatus">ABORTED</value>
    </set>
</property>
```

- **result**: Here you can configure cleanup behaviour: cronjobs of what result should be handled by cleanup. This property must contain at least one value. You can put multiple values here, like for example:

```
<property name="result">
    <set>
```

```

</value type="de.hybris.platform.cronjob.enums.CronJobResult">SUCCESS</value>
</value type="de.hybris.platform.cronjob.enums.CronJobResult">FAILURE</value>
</value type="de.hybris.platform.cronjob.enums.CronJobResult">ERROR</value>
</set>
</property>

```

- **excludedCronJobCodes:** Here you list cron jobs that should be excluded from the cleanup. Put code of the cron job here, like in the following example:

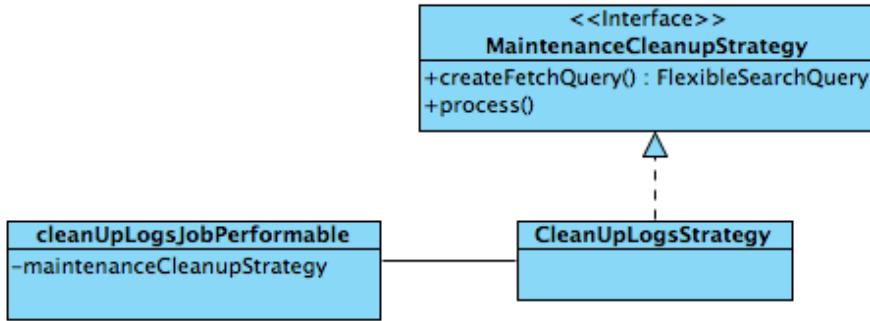
```

<property name="excludedCronJobCodes" >
<set>
<value>myOwnTestCronJob</value>
<value>manualExecutedIndexCronJob</value>
</set>
</property>

```

## CronJob Logs Clean-up

The following provides information on cleaning CronJob Logs.



To define maintenance CronJob for cleaning CronJob log entries (both JobLogs and LogFiles) please create generic CronJob and use `cleanUpLogsJobPerformable` as a job performable.

`cleanUpLogsJobPerformable` uses `CleanUpLogsStrategy` to search for CronJobs with logs eligible for deletion.

`CleanUpLogsStrategy` deletes unnecessary log entries depending on new fields in the CronJob's type definition:

- `logsDaysOld` - how old the JobLog should be to get deleted
- `logsCount` - max amount of JobLogs to keep
- `logsOperator` - if set to OR log entry is deleted if only one condition is met (max count exceeded OR entry is old enough), if set to AND both conditions must be met (max count exceeded AND log entry is old enough)
- `filesDaysOld` - how old the LogFile should be to get deleted
- `filesCount` - max amount of LogFiles to keep
- `filesOperator` - if set to OR log entry is deleted if only one condition is met (max count exceeded OR entry is old enough), if set to AND both conditions must be met (max count exceeded AND log entry is old enough)

Default values of new fields are listed below and can be adjusted using following properties:

Field	Property	Default value
<code>logsDaysOld</code>	<code>cronjob.logs.logsdaysold</code>	14
<code>logsCount</code>	<code>cronjob.logs.logscount</code>	5
<code>logsOperator</code>	<code>cronjob.logs.logsoperator</code>	AND

Field	Property	Default value
filesDaysOld	cronjob.logs.filesdaysold	14
filesCount	cronjobs.logs.filescount	5
filesOperator	cronjob.logs.filesoperator	AND

## Cronjob Scripting

Traditionally, creating a new cronjob was time-consuming and entailed many manual steps, for example, you had to create a new java class, take care of spring bean definition, rebuild the platform, restart the server and so on and so forth.

Using dynamic scripting, creating cronjobs becomes much easier and, most importantly, it can be done dynamically at runtime.

To find out more about cronjob scripting, see [Cronjob Scripting](#).

## Ensuring Stuck Synchronization Cron Jobs Are Aborted

The following property ensures that synchronization cron jobs that are stuck in the **NEW** status when other cron jobs of the same type are already running are moved to the **ABORTED** state:

```
synchronization.cronjob.abortOnCollidingSync.always=
```

The property is disabled by default.

## Related Information

[items.xml](#)

[Initializing and Updating SAP Commerce](#)

[Models](#)

[The Task Service](#)

[Cronjob Scripting](#)

## Writing a Hello World CronJob

Here you will create a simple cron job named **HelloWorldCronJob**, which uses a job that prints out log information.

Cron jobs have some advantages over using services, because they may be called not only on demand, but also at some defined time and may be run asynchronously. You may store the logged output for every call of a cron job. Moreover, logic of a cron job can be aborted on demand.

## Creating a New Extension

To be able to create and use cron jobs, first install and initialize the Platform. The **cronjob** functionality is then automatically available as a part of the Platform.

To make sure that **HelloWorldCronJob** is kept apart from other parts of SAP Commerce, use a new extension to implement it. See also [Installation Based on Specified Extensions](#) for information on how to do it. Use the following settings for the new extension:

- extension.name=cronjobtutorial
- extension.package=de.hybris.cronjobtutorial

The new `cronjobtutorial` extension is generated in `<HYBRIS_BIN_DIR>/custom` directory. Do not forget to reference the new extension in the `localextension.xml` file. See also [Installation Based on Specified Extensions](#) for information on how to do it.

## Declaring HelloWorldCronJob

Declare a custom CronJob type called `HelloWorldCronJob`. Do this in the `cronjobtutorial-items.xml` file located in resources folder in your new extension. Add there a subtype of CronJob type like in the following example:

`cronjobtutorial-items.xml`

```
<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="items.xsd">

    <itemtypes>
        <itemtype generate="true"
                  code="HelloWorldCronJob"
                  jaloclass="de.hybris.cronjobtutorial.jalo.HelloWorldCronJob"
                  extends="CronJob"
                  autocreate="true">
        </itemtype>
    </itemtypes>
</items>
```

The subtype of general CronJob type can provide some additional attributes, which are used only by this subtype. For more details, read [items.xml](#). If no additional attributes are needed, you may skip the definition.

Rebuild SAP Commerce by calling `ant` in the `<HYBRIS_BIN_DIR>\platform` directory. `HelloWorldCronJobModel` is generated in the `bootstrap/gensrc` directory. For more information see [Models, Model Class Generation](#) section.

## Create JobPerformable

To define the logic for a job, in the `de.hybris.cronjobtutorial` package create a new class extending `AbstractJobPerformable` and override the `perform` method. In this tutorial, the job should create log information. You may also implement the `JobPerformable` interface directly. The `AbstractJobPerformable` class is used to define the job as performable and non-abortable by default and to provide access to some services. It implements the `JobPerformable` interface.

`MyJobPerformable.java`

```
public class MyJobPerformable extends AbstractJobPerformable<HelloWorldCronJobModel>
{
    private static final Logger LOG = Logger.getLogger(MyJobPerformable.class.getName());

    @Override
    public PerformResult perform(final HelloWorldCronJobModel cronJobModel)
    {
        LOG.info("*****");
        LOG.info("Greeting from MyJobPerformable!!!!");
        LOG.info("*****");

        return new PerformResult(CronJobResult.SUCCESS, CronJobStatus.FINISHED);
    }
}
```

Rebuild the SAP Commerce by calling the `ant` in the `<HYBRIS_BIN_DIR>\platform` directory. The new `MyJobPerformable` has to be defined as a Spring bean in the `cronjobtutorial-spring.xml` file located in the `resources` folder of the `cronjobtutorial` extension:

`cronjobtutorial-spring.xml`

```
<bean id="myJobPerformable" class="de.hybris.cronjobtutorial.MyJobPerformable"
parent="abstractJobPerformable"/>
```

After changing the Spring configuration, restart the hybris Platform and perform a system update. For details see [Initializing and Updating SAP Commerce](#). A system update is needed, because during the phase of essential data creation, for each Spring definition of a class implementing the JobPerformable interface, a ServicelayerJob instance gets created and the code attribute of the job is set to the name of the Spring bean. Hence, using an instance of ServicelayerJobModel with the code attribute set to the myJobPerformable job will call the new implemented myJobPerformable.

## Creating Trigger and Running HelloWorldCronJob

To run the new cron job, you need to perform the following steps:

1. Get the job instance of ServicelayerJobModel having springid attribute set to myJobPerformable.
2. Create and configure an instance of HelloWorldCronJob.
3. Assign the job to the cron job and set attributes for the cron job.

See an example of a test class that you can create in your cronjobtutorial extension:

`HelloWorldIntegrationTest.java`

```
public class HelloWorldIntegrationTest extends ServicelayerTest
{
    private static final Logger LOG = Logger.getLogger(HelloWorldIntegrationTest.class.getName());

    @Resource
    CronJobService cronJobService;

    @Resource
    ModelService modelService;

    @Resource
    FlexibleSearchService flexibleSearchService;

    List<ServicelayerJobModel> servicelayerJobModelList = Collections.EMPTY_LIST;
    ServicelayerJobModel servicelayerJobModel = null;
    HelloWorldCronJobModel helloWorldCJ = null;

    @Before
    public void setUp()
    {
        //The update of the JUnit tenant creates automatically an instance of the defined My
        //Search for it
        ServicelayerJobModel sjm = new ServicelayerJobModel();
        sjm.setSpringId("myJobPerformable");
        try
        {
            servicelayerJobModel = flexibleSearchService.getModelByExample(sjm); //searching t
        }
        catch(ModelNotFoundException e)
        {
            //The cronjob functionality in the processing extension creates for each JobPerfor
            //You just create a job here
            servicelayerJobModel = modelService.create(ServicelayerJobModel.class);
            servicelayerJobModel.setSpringId("myJobPerformable");
            servicelayerJobModel.setCode("myJobPerformable");
            modelService.save(servicelayerJobModel);
            //Keep in mind that creating models in the catch clause is bad style
        }

        // Create a CronJob and set the servicelayerJob
        helloWorldCJ = modelService.create(HelloWorldCronJobModel.class);
        helloWorldCJ.setActive(Boolean.TRUE);
        helloWorldCJ.setJob(servicelayerJobModel);
    }
}
```

```

        modelService.save(helloWorldCJ);

        //Below is just to show how to create a trigger but not really necessary for this JUnit
        createTrigger(helloWorldCJ);
    }

    @Test
    public void testIfThePerformableExist()
    {
        //Check if there is an instance of myJobPerformable
        assertNotNull("*****No performable with springID *myJobPerformable* found p
                      + "Update your JunitTenant to let create an instance!", serviceLayerJob
    }

    @Test
    public void testExecuteThePerformable()
    {
        //Check if setup works correctly
        assertNotNull("*****The in set upcreated CronJob is null?", helloWorldCJ);

        //Perform the CronJob once for the test
        cronJobService.performCronJob(helloWorldCJ);

        //Wait for the result to be written
        try
        {
            Thread.sleep(2000);
        }
        catch (final InterruptedException e)
        {
            e.printStackTrace();
        }

        LOG.info("***** lets wait 2 seconds for the result *****");

        //Test if the job was executed successfully, if it fails here then try to extend the
        assertEquals("***** The perfromable has not finished successfull or more wa
                     CronJobResult.SUCCESS, helloWorldCJ.getResult());
    }

    //Create a trigger just to show how to implement it
    public void createTrigger(final HelloWorldCronJobModel helloWorldCJ)
    {
        final TriggerModel triggerModel = modelService.create(TriggerModel.class);
        triggerModel.setActive(Boolean.TRUE);
        triggerModel.setMinute(new Integer(1));
        triggerModel.setCronJob(helloWorldCJ);
        modelService.save(triggerModel);
    }
}

```

The assigned JobPerformable type should match the CronJob type. It is important to use an appropriate `springid`, so that the `ServiceLayerJobModel` points to an existing JobPerformable bean definition.

Triggers can be created using cron expressions. When you create a trigger, you should then assign it to the cron job. The code sample above presents how to create a trigger that executes the cron job once a minute. Below is a sample with CronExpression:

```
//Choose the cron expression for starting the job every 10 seconds
triggerModel.setCronExpression("0/10 * * * ?");
```

Running the `HelloWorldCronJobModel` results in the following output on a console:

```
*****
Greeting from MyJobPerformable!!!
*****
```

# Writing an Abortable Job

Because job execution may take a lot of time, you may wish to abort a job. To do this, add a code snippet for checking if you wish to abort a job. Because you can not easily and safely abort a **JobPerformable** at any time unless you kill the thread, it is necessary to place hooks for the abort before stages that you expect to be time-consuming.

The tutorial uses the job created in the [HelloWorldCronJob Tutorial](#) that is adequately adjusted as described below.

## Adjusting MyJobPerformable

The execution of **MyJobPerformable** from [HelloWorldCronJob Tutorial](#) does not take a lot of time. Let us modify the job to extend its execution time by adding a **for** loop:

`MyJobPerformable.java`

```
public class MyJobPerformable extends AbstractJobPerformable<HelloWorldCronJobModel>
{
    private L10NService l10nService;

    @Required
    public void setL10nService(final L10NService l10nService)
    {
        this.l10nService = l10nService;
    }

    @Override
    public PerformResult perform(final HelloWorldCronJobModel cronJob)
    {

        for (int i = 0; i <= 1000; i++)
        {
            try
            {
                System.out.println("Greeting '" + l10nService.getLocalizedString(cronJob.getMessage())
                    + "' from MyJobPerformable for " + i + " times.");
                Thread.sleep(5000);
            }
            catch (final InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
        }
        //the following will be executed when the loop is finished
        return new PerformResult(CronJobResult.SUCCESS, CronJobStatus.FINISHED);
    }
}
```

The job displays the message in a console:

```
Greeting 'Hallo' from MyJobPerformable for 1 times.
Greeting 'Hallo' from MyJobPerformable for 2 times.
Greeting 'Hallo' from MyJobPerformable for 3 times.
Greeting 'Hallo' from MyJobPerformable for 4 times.
Greeting 'Hallo' from MyJobPerformable for 5 times.
...

```

After 5000 seconds the job succeeds and the following message is displayed in the console:

```
Greeting 'Hallo' from MyJobPerformable for 5000 times.
```

## Overriding isAbortable Method

By default each job is not abortable. To enable the abort feature, you should override the `isAbortable` method in `MyJobPerformable`:

#### MyJobPerformable.java

```
@Override
public boolean isAbortable()
{
    return true;
}
```

Alternatively, if you extend the `AbstractJobPerformable`, you may override a bean property value in Spring configuration:

#### cronjobtutorial-spring.xml

```
<bean id="myJobPerformable" class="de.hybris.cronjobtutorial.MyJobPerformable"
parent="abstractJobPerformable" >
<property name="l10nService" ref="l10nService" />
<property name="abortable" value="true"/>
</bean>
```

## Checking Abort Request

Currently the job is marked as abortable, which allows to activate the abort flag for the cron job, similarly to the `interrupt` flag for `Thread`. When implementing an abortable job it is important to regularly check if the user has requested to abort the job. If true, the abort flag is set, the job should be stopped, and the cron job should have a proper status and result. You should decide when to check the flag and what to do if the flag is set, for example, to clean up. Time of processing between verifications should be moderate.

The following code sample calls the `clearAbortRequestedIfNeeded` of the `AbstractJobPerformable`. It checks if the given `cronJob` is requested to be aborted and has `REQUESTABORT` flag set to `true`. If so, the flag is set to `false` again and the job execution is prematurely stopped.

#### MyJobPerformable.java

```
if(clearAbortRequestedIfNeeded(cronJob))
{
    //abort the job
    //do some clean-up

    return new PerformResult(CronJobResult.ERROR, CronJobStatus.ABORTED);
}
```

After including the `if` condition in `perform` method, the `MyJobPerformable` looks as follows:

#### MyJobPerformable.java

```
public class MyJobPerformable extends AbstractJobPerformable<HelloWorldCronJobModel>
{
    private L10NService l10nService;

    @Required
    public void setL10nService(final L10NService l10nService)
    {
        this.l10nService = l10nService;
    }

    @Override
    public PerformResult perform(final HelloWorldCronJobModel cronJob)
    {
```

```

        for (int i = 0; i <= 1000; i++)
    {
        try
        {
            System.out.println("Greeting '" + l10nService.getLocalizedString(cronJob.getKey() + " from MyJobPerformable for " + i + " times.");
            Thread.sleep(5000);

            if (clearAbortRequestedIfNeeded(cronJob))
            {
                System.out.println("The job is aborted.");
                return new PerformResult(CronJobResult.ERROR, CronJobStatus.ABORTED);
            }
        }
        catch (final InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }

    return new PerformResult(CronJobResult.SUCCESS, CronJobStatus.FINISHED);
}

@Override
public boolean isAbortable()
{
    return true;
}
}

```

## Aborting the Job

To abort the job, you should call the method:

```
cronJobService.requestAbortCronJob(cronJob);
```

The cron job results then with **ERROR**, its status is set to **ABORTED** and the following message is displayed in the console:

```
The job is aborted.
```

## Importing CronJob Instances Using ImpEx

Using ImpEx for creating instances of **CronJob** is useful if you want to define it declaratively, and not by creating code that must be compiled and deployed. This is worth applying especially for a system set up in combination with ImpEx Import for Essential and Project Data, as the data setup is often adjusted.

The tutorial presumes that you have already created **MyJobPerformable** and configured it as a Spring bean, as described in the [HelloWorldCronJob Tutorial](#).

Update the system to automatically create the instance of **ServicelayerJob** related to defined **myJobPerformable** bean from Spring context.

## Importing CronJobModel Instance

Create a cron job by importing an instance of **HelloWorldCronJob** type and reference it to the **JobPerformable** instance:

```
INSERT_UPDATE HelloWorldCronJob;code[unique=true];job(code);message;sessionLanguage(isocode);
;myHelloWorldServicelayerCronJob;myJobPerformable;greeting.key;en;
```

This example creates an instance of `HelloWorldCronJobModel` with an assigned `ServicelayerJob` instance, and some custom localized message key.

## Creating Trigger

If you wish to start your cron job automatically, you need to define a trigger:

```
INSERT_UPDATE Trigger; cronJob(code)[unique=true]; cronExpression
; myHelloWorldServicelayerCronJob; 0 30 10-11 ? * WED,FRI
```

This ImpEx script creates an instance of `Trigger` assigned to `myHelloWorldServicelayerCronJob` and a cron expression that causes the trigger to start the cron job at 10:30, 11:30, every Wednesday and Friday.

## Saving SessionContext Attributes During Import

When importing cron jobs through ImpEx, the system adds additional attributes needed for import purposes to `SessionContext`. All new cron jobs that are created receive all `SessionContext` attributes as `sessionContextValues`. To prevent `sessionContextValues` from holding redundant attributes that are required only during import, use the following property with a list of attributes that you don't want to save in new cron jobs' `sessionContextAttributes`:

```
cronjob.ctx.filtered.attributes.in.impex.import.mode=disableRestrictions,disableRestrictionGroupInher
```

Use comma without whitespace as a delimiter. Attributes are case-sensitive. By default, the following attributes are included as the value of this property:

- `disableRestrictions`,
- `disableRestrictionGroupInheritance`,
- `use.fast.algorithms`,
- `import.mode`,
- `disable.attribute.check`,
- `disable.interceptor.beans`,
- `disable.interceptor.types`,
- `disable.UniqueAttributesValidator.for.types`,
- `currentCronJob`,
- `currentJob`,
- `core.types.creation.initial`,
- `save.from.service.layer`,
- `ctx.enable.fs.on.read-replica`,
- `impex.creation`.

## Migrating CronJobs

If you have created cron jobs using the deprecated Jalo layer, you should migrate your cron jobs to the ServiceLayer to be able to use them with the current version of SAP Commerce. Here you will find an example of migrating a job created using Jalo with the custom `cronjobtutorial` extension.

When creating cron jobs using Jalo you had to create a subtype of `Job` and a subtype of `CronJob` in `items.xml` file of the respective extension, in this case in `cronjobtutorial-items.xml`. Then the corresponding Jalo classes for the job and cron job were

automatically generated. To migrate to the ServiceLayer, you need to create a new job that implements the **JobPerformable** and register it as Spring bean.

## Jalo-Based Implementation

When creating cron jobs using Jalo you had to create a subtype of **Job** and a subtype of **CronJob** in the **items.xml** file of the corresponding extension. The **cronjobtutorial-items.xml** is located in the **resources** directory of the **cronjobtutorial** extension. After rebuilding the Platform, the corresponding classes are generated. Your custom Jalo classes for the job and the cron job should extend the adequate generated classes.

## Defining Types

The first step to define cron jobs in the Jalo layer was to create a subtype of **Job** and a subtype of **CronJob** in **cronjobtutorial-items.xml** file.

- **FindUserJob** is a subtype of **Job**:

**cronjobtutorial-items.xml**

```
<itemtype code="FindUserJob"
          jaloclass="de.hybris.platform.jalo.FindUserJob"
          extends="Job"
          autocreate="true"
          generate="true">
</itemtype>
```

- **FindUserCronJob** is a subtype of **CronJob**. It contains a String attribute **userId**:

**cronjobtutorial-items.xml**

```
<itemtype code="FindUserCronJob"
          autocreate="true"
          generate="true"
          extends="CronJob"
          jaloclass="de.hybris.platform.jalo.FindUserCronJob">
<attributes>
    <attribute type="java.lang.String" qualifier="userId">
        <modifiers read="true" write="true" initial="true" optional="true"/>
        <persistence type="property"/>
    </attribute>
</attributes>
</itemtype>
```

During the build process, proper getter and setter are created.

## Defining Jalo Classes

Jalo classes with a logic for job and cron job should extend proper generated classes. The job logic should override the **performCronJob** method to allow the job to be executed. This method should contain tasks that should be performed during the execution of the cron job. The example below presents how to find a user login based on user id:

- **FindUserJob.java** with job logic:

**FindUserJob.java**

```
public class FindUserJob extends GeneratedFindUserJob
{
    private final static Logger LOG = Logger.getLogger(FindUserJob.class);
```

```

@Override
protected CronJobResult performCronJob(final CronJob cronJob) throws AbortCronJobException {
    if (!(cronJob instanceof FindUserCronJob))
    {
        throw new AbortCronJobException("Given cronjob is not instance of LDIFI");
    }
    final FindUserCronJob findUserCronJob = (FindUserCronJob) cronJob;
    final User user = UserManager.getInstance().getUserByLogin(findUserCronJob.getLogin());
    LOG.info("JaloJob: " + user.getName());
    return cronJob.getFinishedResult(true);
}

```

- **FindUserCronJob.java** file without any specific logic:

#### FindUserCronJob.java

```

public class FindUserCronJob extends GeneratedFindUserCronJob
{
}

```

## Migrating to the ServiceLayer

### Creating JobPerformable

To migrate your job to the ServiceLayer you should:

1. Create a new class for **JobPerformable**

It should extend **AbstractJobPerformable**, which marks the job as performable and non-abortable. The new class performs the same logic as the job created in Jalo, but instead of using **UserManager**, it has a dependency to **UserService**. Marking the setter by **@Required** annotation means that there has to be a proper configuration for dependency injected value for **UserService**.

#### FindUserJobPerformable.java

```

public class FindUserJobPerformable extends AbstractJobPerformable<FindUserCronJobModel>
{
    private final static Logger LOG = Logger.getLogger(FindUserJobPerformable.class);

    private UserService userService;

    @Override
    public PerformResult perform(final FindUserCronJobModel cronJob)
    {
        final UserModel user = userService.getUserForUID(cronJob.getUserUid());
        LOG.info("PerformableJob: " + user.getName());
        return new PerformResult(CronJobResult.SUCCESS, CronJobStatus.FINISHED);
    }

    @Required
    public void setUserService(final UserService userService)
    {
        this.userService = userService;
    }
}

```

2. After introduced changes rebuild the Platform.

3. Register **UserService** as a Spring bean in the **cronjobtutorial-spring.xml** file to inject it to **FindUserJobPerformable**.

As a parent attribute of the bean, provide the class that extends your **JobPerformable**, that is **FindUserJobPerformable**. Specify which dependencies should be injected. In our case, it should be only **userService**:

#### cronjobtutorial-spring.xml

```
<bean id="findUserJobPerformable" class="de.hybris.cronjobtutorial.FindUserJobPerformable"
      parent="abstractJobPerformable">
    <property name="userService" ref="userService"/>
</bean>
```

- After changing the Spring configuration, restart the platform and perform a system update.

During the update process essential data is created. Here the system searches for all Spring beans implementing the **JobPerformable** interface and creates for each one an instance of **ServicelayerJob**. You do not need to create a new instance of job, because it is already created during the update.

## Migrating the CronJob

A model class for the cron job is generated during the build process, based on the created subtype in the **cronjobtutorial-items.xml** file. There is no need to perform any other actions.

## Removing the Unnecessary Code

You can remove the old subtype of **Job** from the **cronjobtutorial-items.xml** file, but do not remove the subtype of **CronJob**, because it is used for creating the Model class for the cron job. The Jalo class for the old job can be also removed, but remember to verify existing dependencies to that class. After removing the Jalo classes, you may notice some warnings with regard to the missing Jalo class. Perform [Cleanup Type System](#) in the Administration Console to get rid of them.

## Related Information

[ServiceLayer](#)

[Models](#)

## Defining a Custom CronJobFactory

Using a custom **CronJobFactory**, you can execute a **JobPerformable** without creating cron jobs using a trigger.

If you assign a trigger to a job, the related cron job for executing the job should be created on demand in a generic way. Sometimes there is a need to customize the generic creation to set a value for a specific attribute of the cron job, for example, in case of a mandatory attribute.

This tutorial presumes that you have already created **MyJobPerformable** and configured it as a Spring bean, as described in the [HelloWorldCronJob Tutorial](#).

## Creating Trigger for a Job

Let us create a trigger and assign it to the job to regularly fire the job as a part of creating essential data during the initialization or update process. The method annotated with **@SystemSetup** should look like this:

```
@SystemSetup(type = Type.ESSENTIAL, process = Process.ALL)
public void createJobTrigger()
{
    // Create Trigger
```

```

        final TriggerModel triggerModel = modelService.create(TriggerModel.class);
        triggerModel.setActive(Boolean.TRUE);
        triggerModel.setCronExpression("0 0/25 14-16 ? * *");

        // get the job instance having set "myJobPerformable" as springId
        final FlexibleSearchQuery query = new FlexibleSearchQuery("SELECT {" + ServicelayerJobModel
                + ServicelayerJobModel._TYPECODE + "} WHERE {" + ServicelayerJobModel
        query.addQueryParameter("springid", "myJobPerformable");
        final ServicelayerJobModel job = flexibleSearchService.<ServicelayerJobModel> searchUnique

        // Assign a Job to a Trigger
        triggerModel.setJob(job);
        modelService.save(triggerModel);
    }
}

```

For more details on `@SystemSetup`, see [Hooks for Initialization and Update Process](#).

When the job gets triggered, you get a `NullPointerException`, because the `message` attribute provides no default value, and is null after the generic creation of the cron job. The goal is to modify the generic cron job creation such that the attribute has a fixed value set after the instantiation. You can do this by using a default value defined in `items.xml` file, but if the value should be defined in a programmable or non-static way, use the `CronJobFactory` mechanism.

## Defining CronJobFactory

To create a custom `CronJobFactory`, perform the following steps:

- Provide a factory class setting the `message` attribute to a fixed value by default:

```

public class HelloWorldCronJobFactory implements CronJobFactory<HelloWorldCronJobModel, JobMode
{
    private ModelService modelService;

    @Required
    public void setModelService(final ModelService modelService)
    {
        this.modelService = modelService;
    }

    @Override
    public HelloWorldCronJobModel createCronJob(final JobModel jobModel)
    {
        final HelloWorldCronJobModel result = modelService.create(HelloWorldCronJobMode
        result.setMessage("attribute from factory");
        result.setJob(jobModel);

        return result;
    }
}

```

- Define the factory as Spring bean:

`cronjobtutorial-spring.xml`

```

<bean id="helloWorldCronJobFactory" class="de.hybris.cronjobtutorial.HelloWorldCronJobFactory">
    <property name="modelService" ref="modelService" />
</bean>

```

- Compile and restart the system.

As a result, the following message indicating the correct usage of the factory is displayed:

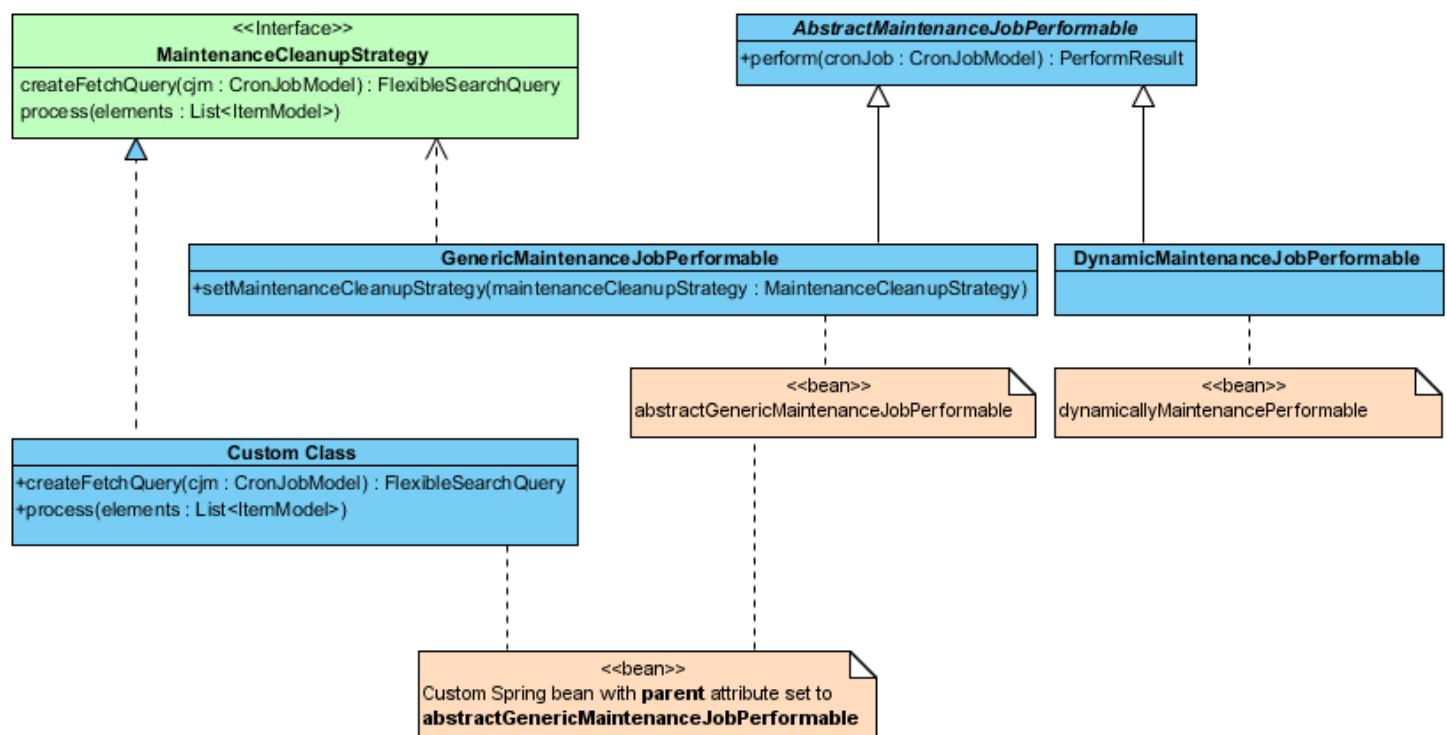
Greeting 'attribute from factory' from HelloWorldJobPerformable for 0 times.

# Creating Strategy to Process Instances

You can implement your own strategy for processing instances of a specific type using the maintenance framework.

Maintenance framework shown on below UML diagram consists of two main parts:

- The **MaintenanceCleanupStrategy** is an interface for your custom strategy implementation with a definition of:
    - Search for instances of the specific type
    - How the processing should proceed
  - The **AbstractMaintenanceJobPerformable** class is a **JobPerformable** that:
    - Executes the given strategy by paging through the search results
    - Executes processing of instances for each page
- It is extended in two different classes:
- The **GenericMaintenanceJobPerformable**. You need to extend its **abstractGenericMaintenanceJobPerformable** bean in the bean declaration of your custom strategy.
  - The **DynamicMaintenanceJobPerformable**. It is provided with ready to use **Job**, which you can use on the running system.



To use maintenance framework you need to do the following:

1. Implement a task that searches for specific instances and process them. It has to implement the **MaintenanceCleanupStrategy** interface.
2. Register the new bean in the Spring context. It must extend the **abstractGenericMaintenanceJobPerformable** bean.

## → Tip

### Configuration of AbstractMaintenanceJobPerformable Behavior

You can configure behavior of the **AbstractMaintenanceJobPerformable** through the **abstractGenericMaintenanceJobPerformable** bean. For more details see also the [Additional Configuration](#) section.

## Configuring Static Strategy

Static strategy means that you need to implement a custom class with its bean declaration. It requires a system rebuild and update. All the examples below are based on a custom strategy **CleanupDeactivatedUsersStrategy** that you can copy and use in your system. If you do not need a static strategy but a solution that you need to use only once on a running system, read the section [Using Dynamic Strategy](#).

### Create Your Own Task for Searching and Processing

Your custom task for searching and processing instances must implement the **MaintenanceCleanupStrategy** interface. The interface is shown in the code snippet below:

#### **MaintenanceCleanupStrategy.java**

```
public interface MaintenanceCleanupStrategy<T extends ItemModel, C extends CronJobModel>
{
    FlexibleSearchQuery createFetchQuery(C cjm);
    void process(List<T> elements);
}
```

The **createFetchQuery** returns a **FlexibleSearchQuery** object, which, when executed by the **FlexibleSearch**, returns all the **ItemModels** that should be processed. In the **process** method, an instance of the **ItemModel** can additionally be checked for some parameters before processing. Below is an example of custom class that uses the **MaintenanceCleanupStrategy** interface:

#### **CleanupDeactivatedUsersStrategy.java**

```
public class CleanupDeactivatedUsersStrategy implements MaintenanceCleanupStrategy<UserModel, CronJobModel>
{
    private final static Logger LOG = Logger.getLogger(CleanupDeactivatedUsersStrategy.class);
    private ModelService modelService;

    //set a default value, we do not want to accidentally remove all non-disabled users
    private boolean loginDisabled = true;

    @Override
    public FlexibleSearchQuery createFetchQuery(final CronJobModel cjm)
    {
        FlexibleSearchQuery fsq = new FlexibleSearchQuery("SELECT {PK} FROM {USER}"); //return
        fsq.setResultClassList(Arrays.asList(UserModel.class)); //always provide the expected
        return fsq;
    }

    @Override
    public void process(final List<UserModel> elements)
    {
        LOG.info("Removing " + elements.size() + " " + (this.loginDisabled ? "deactivated" :
        modelService.removeAll(elements));
    }

    public void setModelService(final ModelService modelService)
    {
        this.modelService = modelService;
    }

    public void setLoginDisabled(final boolean loginDisabled)
    {
        this.loginDisabled = loginDisabled;
    }
}
```

When implementing the custom strategy, you have access to the bean properties, which can only be changed in the Spring configuration file. Additionally, when implementing the **createFetchQuery** you can use the **CronJobModel** in the strategy. By default

the `cronJobModel.getJob()` contains a **threshold** and a **searchType** attributes which can be used for some dynamic search process attributes. In the example below those attributes are used to specify the search on a certain type of certain age:

#### CleanupDeactivatedUsersStrategy.java

```
//...
@Override
public FlexibleSearchQuery createFetchQuery(final CronJobModel cjm)
{
    if (cjm.getJob() instanceof MaintenanceCleanupJobModel)
    {
        final MaintenanceCleanupJobModel job = (MaintenanceCleanupJobModel) cjm.getJob();

        //first, create the FlexibleSearch query as String
        //we could setup the value directly, but for example purpose it is a parameter
        final StringBuilder builder = new StringBuilder();
        builder.append("SELECT {PK} FROM {" + job.getSearchType() + "}" );
        //the search type is defined in the cronjob
        builder.append("WHERE {loginDisabled} = ?loginDisabledValue AND {lastLogin} < ?lastLogin");

        //the parameter map
        final Map<String, Object> params = new HashMap<String, Object>();
        params.put("loginDisabledValue", Boolean.valueOf(loginDisabled));
        params.put("threshold", new Date(System.currentTimeMillis() - 1000L * 3600L * 24L));
        //the threshold value of the job is used as: "every user which last login was more than 24 hours ago"

        //now, create the FlexibleSearchQuery object (query + parameters), the value
        final FlexibleSearchQuery query = new FlexibleSearchQuery(builder.toString(), params);

        //set the class for the result, internal the performable will work with PKs
        query.setResultClassList(Arrays.asList(UserModel.class));

        return query;
    }
    throw new IllegalStateException("The job was not a MaintenanceCleanupJob");
}
//...
```

#### Declare the Bean

After implementing the custom task, you must declare a bean in the Spring configuration file. The bean should use the **abstractGenericMaintenanceJobPerformable** as the parent bean to inherit its properties as default configuration, like in the example below :

#### extension-spring.xml

```
<bean id="cleanupDeactivatedUsersPerformable" parent="abstractGenericMaintenanceJobPerformable" >
    <property name="maintenanceCleanupStrategy">
        <bean class="de.hybris.example.impl.CleanupDeactivatedUsersStrategy" >
            <property name="loginDisabled" value="true"/>
            <property name="modelService" ref="modelService"/>
        </bean>
    </property>
</bean>
```

After rebuilding and updating your SAP Commerce you will have the new **cleanupDeactivatedUsersPerformable** job available and ready to assign to your cron job.

#### Additional Configuration

In the declaration of your bean you can also set a new value for the **pageSize** and the **abortOnError** properties; these are declared in the **abstractGenericMaintenanceJobPerformable** bean:

- The **pageSize**: This property controls the number of the elements available on the list for the process. A value of **100** means that the list contains at least **100 ItemModels**.

- The **abortOnError**: During the processing part an exception may occur. Depending on the value of this property the following happens:
  - **true**: The job is aborted.
  - **false** The job still proceeds and logs the exception.

#### **extension-spring.xml**

```
<bean id="cleanupDeactivatedUsersPerformable" parent="abstractGenericMaintenanceJobPerformable" >
<property name="maintenanceCleanupStrategy">
<bean class="de.hybris.example.impl.CleanupDeactivatedUsersStrategy " >
<property name="loginDisabled" value="true"/>
<property name="modelService" ref="modelService"/>
<property name="pageSize" value="20"/>
<property name="abortOnError" value="true"/>
</bean>
</property>
</bean>
```

## Using Dynamic Strategy

The purpose of a dynamic strategy is to give you a possibility to process instances of a specific type whenever you want, without rebuilding and updating your SAP Commerce. It is possible that you do not need to create a static strategy that can be used periodically. For example, you need to perform a cleanup of a specific type and you need to do it once only. The maintenance framework provides a ready-to-use the **dynamicallyMaintenancePerformable** job with two additional attributes: the **Process code** and the **Search code**. For each attribute you can provide a BeanShell script. The first attribute should contain a process script, the second one a search script. In other words, first you define what instances of what type should be processed and then what should be done to these instances. You can use this job in Backoffice.

The screenshot shows the SAP Fiori interface for managing cron jobs. On the left, there's a sidebar with a tree view of system configurations and a section for saved queries. The main area is focused on a specific cron job named "dynamicallyMaintenancePerformable". The job is currently unbound. It has fields for process code and search code. Under comments, there are options for request abort (set to False) and request abort step (set to False). The search composed type is set to "dynamicallyMaintenancePerformable". The spring ID is also set to "dynamicallyMaintenancePerformable". The threshold is marked as N/A.

## Related Information

[SavedValues - Keeping Track of Attribute Value Modification](#)

## Assigning a Cron Job to a Group of Nodes

You can assign a cron job to a group of nodes. It can then be executed by any of the nodes belonging to that group. If a node that is currently executing a cron job stops operating, one of the other nodes from that group takes over and executes it.

## Defining Groups of Nodes

You have to define a group of nodes to be able to assign a cron job to it. Defining a group of nodes means grouping them under the same group name.

To define one or more groups of nodes, in the `local.properties` file, set one or more group names as attributes for the `cluster.node.groups` parameter, per node:

```
cluster.node.groups=group name, another group name
```

Example of defining two groups for a given node:

```
cluster.node.groups=backoffice, frontoffice
```

Nodes on which given groups are defined belong to these groups.

## Assigning a Cron Job To a Group of Nodes

To assign a cron job to a specific group of nodes, set a chosen, previously defined group name as an attribute for the cron job's `nodeGroup` property:

```
.setNodeGroup("group name")
```

Example of assigning a cronjob to the backoffice group

```
.setNodeGroup ("backoffice")
```

You can also do it via ImpEx:

```
INSERT_UPDATE CronJob; code[unique=true];job(code);nodeGroup  
;myCronJob;myJob;backoffice
```

### i Note

If you set attributes for both `nodeGroup` and `nodeID` for a cron job, `nodeID` overrules the `nodeGroup` setting.

### → Tip

When changing the `setNodeGroup`'s attribute for a cron job at run time, you do not need to perform any operations on a trigger corresponding to this cron job - the next scheduled execution of this cron job will be performed automatically by any available node from the group the cron job has just been assigned to.

## Cron Job Events

Platform can publish events on CronJob start and CronJob end. This allows developers to implement logic based on the processing of cron jobs.

When a cron job is about to start, the `BeforeCronJobStartEvent` event is published. Right after a cron job has finished (regardless of the result or state - even when it crashed), the `AfterCronJobFinishedEvent` event is fired.

## Event Attributes

Both events offer these attributes:

- `cronJob`, `cronJobPK`, `cronJobType`: represent respectively the code, the PK, and the type code of a cron job being executed
- `job`, `jobType`: represent respectively the code, and the type code of the job the cron job belongs to
- `scheduled`, `scheduledByTriggerPK`: represent respectively whether cron job execution was scheduled, and, if so, the PK of the trigger scheduling it

- **synchronous**: in case of a cron job being explicitly performed (not scheduled) this attribute tells whether it was run synchronously within the caller's thread or not

In addition, `AfterCronJobFinishedEvent` has the following attributes:

- **result**: the result (enum) a cron job ended with
- **status**: the status (enum) a cron job ended with

## Cluster Considerations

Note that both events are published to the local Platform instance only and aren't sent to other cluster members.

However, you can register a listener that captures these events and wraps them into your own cluster aware events so that the information arrives where you need it. This is also a good opportunity to select only those cron job events that are interesting for the specific use case.

## Capturing Events

Like for all other SAP Commerce events, you must implement and register an `ApplicationListener` spring bean. For more information, see [Registering Event Listeners](#).

## Tracking CronJob Progress

You can add a cronjob progress tracker to an existing cronjob by using the `CronJobProgressTracker` class.

It holds information in memory and saves data to the `CronJobHistory` item. To add the tracker, create a new instance of the `CronJobProgressTracker` class, and use its `setProgress` method to update the value displayed in the UI. Both the average and the remaining times are calculated from historical runs of a given cronjob. If no historical data is available, N/A is displayed.

Code	Job code	Progress
sampleProgressingCronJob1	sampleProgressingJobPerformable	(21%) [=====] elapsed: 15s, average: 58s, remaining: 43s

Showing 1 to 1 of 1 entries

**Abort cron jobs**

To change the intervals of a cron job, use the property `cronjob.progress.interval.seconds`. The default value is set to 5 seconds.

## Example

In the example, we create a new extension using the ant `extgen` command. We named it `extendedcronjob`, used the `yempty` template, and set `de.hybris.extendedcronjob` as package.

Next, we create a simple `SampleProgressingJobPerformable` class that just waits some time, and increments the cronjob progress.

We create a new `CronJobProgressTracker` instance:

```
final CronJobProgressTracker tracker = new CronJobProgressTracker(modelService.getSource(cronJob));
```

And we set its progress:

```
tracker.setProgress(Double.valueOf(i));
```

To flush the data from the last 5 seconds to the database, use the `close()` method:

```
tracker.close();
```

See the complete implementation:

`SampleProgressingJobPerformable.java`

```
package de.hybris.extendedcronjob;

import de.hybris.platform.cronjob.enums.CronJobResult;
import de.hybris.platform.cronjob.enums.CronJobStatus;
import de.hybris.platform.cronjob.jalo.CronJobProgressTracker;
import de.hybris.platform.cronjob.model.CronJobModel;
import de.hybris.platform.servicelayer.cronjob.AbstractJobPerformable;
import de.hybris.platform.servicelayer.cronjob.PerformResult;

public class SampleProgressingJobPerformable extends AbstractJobPerformable<CronJobModel>
{
    @Override
    public PerformResult perform(final CronJobModel cronJob)
    {
        final CronJobProgressTracker tracker = new CronJobProgressTracker(modelService.getSource(cronJob));
        for (int i = 1; i < 100; i++)
        {
            try
            {
                tracker.setProgress(Double.valueOf(i)); // <- set progress
                Thread.sleep(Double.valueOf(100 + (1000 * Math.random())).intValue());
            }
            catch (final InterruptedException e)
            {
                return new PerformResult(CronJobResult.FAILURE, CronJobStatus.ABORTED);
            }
        }
        tracker.close(); // <- save last progress to the database
        return new PerformResult(CronJobResult.SUCCESS, CronJobStatus.FINISHED);
    }
}
```

`extendedcronjob-spring.xml`

```
<bean id="sampleProgressingJobPerformable" class="de.hybris.extendedcronjob.SampleProgressingJobPerf...
```

## Deletion of CronJobHistory Entries

SAP Commerce comes with `cronJobHistoryRetentionCronJob` that limits the number of `CronJobHistory` entries per cronjob.

`cronJobHistoryRetentionCronJob` is enabled by default in new SAP Commerce installations. For that reason, you should make sure that your data is safe before you update to a new version of SAP Commerce.

The cronjob uses a retention rule of the data retention framework to delete `CronJobHistory` entries.

The property for creating a clean-up job for deleting old cronjob history entries is defined and configured as `cronjobhistory.cleanupjob.create=true`. The job is created during system initialization or system update.

The job leaves only one entry and removes all older entries from the history.

The job runs according to the given cron expression. The default is every hour:

```
cronjobhistory.cleanupjob.cronexpression=0 0 * ? * *
```

## Unlocking Cron Jobs

Use cron job unlocking to update cron jobs that have been terminated abnormally and are stuck in the running state.

Infrastructure problems such as a node failure or a database outage can terminate cron jobs. Such cron jobs are stuck in the **RUNNING** state, which prevents the system from processing them in any way. Use cron job unlocking to mark abnormally terminated cron jobs as **ABORTED**.

Cron job unlocking ensures that the system identifies cron jobs with the **RUNNING** status that are assigned to dead nodes, and subsequently terminates them at established intervals.

### i Note

It's necessary to enable clustering and cluster ID auto discovery before using cron job unlocking.

Cron job unlocking is managed by the `StaleCronJobUnlocker` class. The class is activated during the startup of the task engine, which occurs every time a tenant is started or when the first session has been created.

Activating cron job unlocking creates an active thread that tries to unlock stale cron jobs that are running on shutdown cluster nodes.

Use the following properties in your `local.properties` file to configure cron job unlocking:

Property	Default Value	Description
<code>cronjob.unlocker.active</code>	<code>#{cluster.nodes.autodiscovery}</code>	Activating cron job unlocking
<code>cronjob.unlocker.interval.ms</code>	300,000 milliseconds	Intervals between subsequent cron job unlocking
<code>cronjob.unlocker.stale.node.interval.ms</code>	1,800,000 milliseconds	How long nodes need to be inactive to be considered stale
<code>cronjob.unlocker.cronJobs.unlockLimit</code>	200	Maximal number of cron jobs that can be unlocked during 1 pass. Set it to 0

Property	Default Value	Description
		or fewer to check all items.
cronjob.unlocker.stale.node.cutoff.interval.seconds	1800 seconds	Nodes that are inactive longer than the time determined by this property are ignored. Set it to 0 or fewer to include all nodes.

## Running Cron Job Unlocking Manually

To run cron job unlocking manually, create a new `StaleCronJobUnlocker` thread and use its start method:

```
final ClusterNodeManagementService clusterManagement = DefaultClusterNodeManagementService.getInstance();
if (clusterManagement.isAutoDiscoveryEnabled())
{
    new StaleCronJobUnlocker(Registry.getCurrentTenant()).start();
}
```

The thread continuously attempts to unlock stale cron jobs unless stopped explicitly by the following method:

```
staleCronJobUnlocker.stopUpdatingAndFinish(TIMEOUT_DURATION);
```

## Cron Job History Update

Cron job history status needs to be reset when there are crashed cron jobs. This process is implemented by cron job unlocking that assumes that there's only one `CronJobHistory` item with the **RUNNING** status for each crashed `CronJob` item.

All `CronJobHistory` items that are incorrectly marked with the **RUNNING** status are updated to the **ABORTED** state by `ResetCronJobHistory`:

```
@SystemSetup(extension = ProcessingConstants.EXTENSIONNAME)
public class ResetCronJobHistory
{
    @SystemSetup(type = SystemSetup.Type.PROJECT, process = SystemSetup.Process.ALL)
    public void resetHistoryStatus()
    {
        CronJobManager.getInstance().findAndFixAllCronJobHistoryEntries();
    }
}
```

`ResetCronJobHistory` provides a system update logic for the processing extension. Cron job history is updated when the extension's project data is created.

In order to prevent aborting history items of new cron job items that are running during the update, only `CronJobHistory` items that are created before a set time threshold are aborted. Use the following property to adjust the threshold:

```
cronjob.history.reset.threshold.hours
```

The default value is 24 hours.

# Ensuring Cron Jobs Are Run After Lost Database Connection

Ensure that aborted cron jobs are run after a lost database connection.

## Prerequisites

You have enabled cron job unlocking in your configuration. For more information cron job unlocking, see [Unlocking Cron Jobs](#).

## Context

Use the `cronjob.enableRepeat` property to ensure that aborted cron jobs are restarted after the system regains lost connection with the database. After applying this property, it's necessary that you determine how many times the system makes attempts to restart affected cron jobs by setting the `number of retries` attribute separately, for example, in Backoffice, for each cron job item that you want to be restarted.

## Procedure

1. Add the following property to your configuration.

```
cronjob.enableRepeat=true
```

2. Set the `number of retries` attribute for your selected cron jobs items with the number of the attempts that the system should make to restart the cron jobs as its value.

## Results

The value of the `current_retry` attribute shows the number of attempts that have been made to restart aborted cron jobs.

## TenantAwareThreadFactory

The `ThreadFactory` allows you to create threads with some additional implementation.

The `ThreadFactory` interface defines a `newThread(Runnable r)` method that returns a `Thread`. When some class is using a new thread to do some business logic, the thread isn't aware of the tenant context in which the object of that class was initialized. To have a thread that has access to the tenant session, use `TenantAwareThreadFactory`. Threads created by that factory have the `run()` method overridden in such a way that whenever a thread is run, it sets a tenant and activate a `JaloSession` before runnable and deactivate a session afterwards.

## Spring Scheduler

If you would like to use a Spring scheduler (`ThreadPoolTaskScheduler`), be aware that it uses a `ThreadFactory` that is not tenant aware. You can pass `TenantAwareThreadFactory` to `ThreadPoolTaskScheduler` by creating a bean yourself. Instead of:

```
<task:scheduler id="defaultThreadPoolTaskScheduler" pool-size="${scheduler.thread.pool.size}" />
```

use:

```
<bean id="threadFactory" class="de.hybris.platform.core.TenantAwareThreadFactory">
    <constructor-arg name="tenant" ref="tenantFactory"/>
</bean>
```

```
<bean id="defaultThreadPoolTaskScheduler" class="org.springframework.scheduling.concurrent.ThreadPool
<property name="poolSize" value="${scheduler.thread.pool.size}" />
<property name="threadFactory" ref="threadFactory" />
</bean>
```

After that you can still use the scheduler using the Spring `task` element:

```
<task:scheduler-tasks scheduler="taskScheduler">
<task:scheduled ref="someBean" method="someMethod" cron="someCronExpression" />
</task:scheduler-tasks>
```

## Cache

Cache intercepts calls whenever you access an API (for example ServiceLayer). If you want to get data, it's either returned from the cache, or retrieved from the database and written to the cache.

### When Data Is Cached

The SAP Commerce Cache works transparently. Every time the API is accessed, the cache intercepts calls and handles caching implicitly. The following examples present how caching works:

- Caching item attributes:
  - When calling `product.getCode()`, the underlying data is returned from the cache or, if not yet cached, retrieved, and written to the cache.
  - When calling `product.setCode(X)` and then `save()` (for one value) or `saveAll()` (for multiple values) the cached value is removed (invalidated) from the cache, because it's no longer valid.
- Caching FlexibleSearch results:
  - When executing a FlexibleSearch query such as `SELECT code FROM Product`, the result list is cached in the main cache.
  - When a product is removed, then its item data and the cached FlexibleSearch result for such query is removed from the cache.

### How Data Is Cached

The SAP Commerce Region Cache provides more fine-grained control over your cache configuration. You can define multiple cache regions by specifying their size and eviction strategy, as well as the type instances for which they're responsible. To find an optimal configuration for your needs, see [Region Cache](#).

### When Data Is Removed from Cache

A cache entry is removed from the cache in the following cases:

- The displacement strategy removes the entry from the cache because the cache has reached its maximum size and cannot hold any additional entries. This is done whenever the cache is full and a newer entry is put into the cache.
- The invalidation strategy detects that the cache entry is no longer valid and doesn't represent the correct database content. Note the following about invalidation:
  - Inside a single VM, the cache invalidation is done by a method call.

- When using multiple cluster nodes, the other nodes are notified via TCP or UDP invalidation packets.
- Invalidation isn't performed whenever you change something. There are circumstances where the invalidation is delayed, for example when working with transactions. By default, entries modified inside a transaction are invalidated after the transaction.

## Managing the SAP Commerce Cache

Using the SAP Commerce Administration Console, you can completely clear the cache and monitor cache statistics. Refer to [Monitoring Tab](#) for details.

## Region Cache

The SAP Commerce Region Cache introduces the possibility to split its content into so-called regions. You can specify objects to be cached for each region, which allows you to tune a running system and to make sure that certain objects are cached for a longer time, while other objects are removed more quickly due to a limited cache size.

The SAP Commerce Region Cache is a flexible, modular, and configurable solution. It offers:

- Easy replacement of each new cache module.
- Full programmatic control of cache partitions, for instance, turning off caching for one of the partitions at runtime).
- Easy monitoring.
- Possibility to plug in other third-party cache frameworks, such as Coherence, Gigaspaces, Hazelcast, Memcached.
- The possibility to use a distributed cache. Distributed cache and implementation, which relies on serialization and is possible only for query result region. It isn't possible on entity and type system regions because of not serializable object nature.

## Region Cache Overview

The SAP Commerce Region Cache can be split into multiple partitions called cache regions. Each cache region can be configured separately and holds its own set of types. For instance, it's possible to set up a region only for products. Out of the box, you can configure the size of each region and its eviction strategy. Supported eviction strategies include:

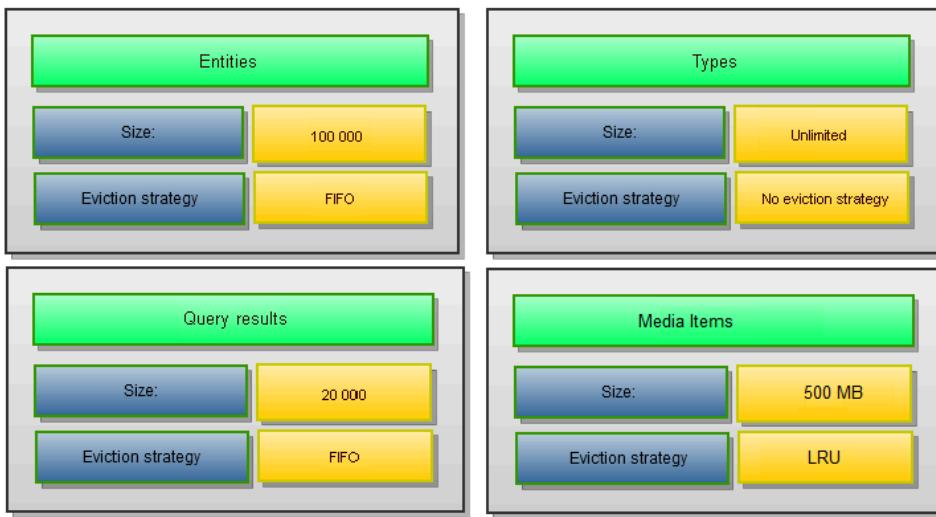
- Least Recently Used (LRU): The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a GET call.
- Least Frequently Used (LFU): For each GET call on the element, the number of hits is updated. When a PUT call is made for a new element, and assuming that the maximum limit is reached for the memory store, the element with the least number of hits, the Less Frequently Used element, is evicted.
- First In, First Out (FIFO): Elements are evicted in the same order as they come in. When a PUT call is made for a new element, and assuming that the maximum limit is reached for the memory store, the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

## Standard Configuration

The default setup of the SAP Commerce Region Cache contains the following regions:

- Type system region: for storing entities of type system items
- Entity region: for storing entities of all types except type system ones
- Query results region: for storing all query results

- Type system query results region: for storing results of queries containing references to the type system items
- Media items region: for storing all media items
- Session region: for storing HTTP sessions



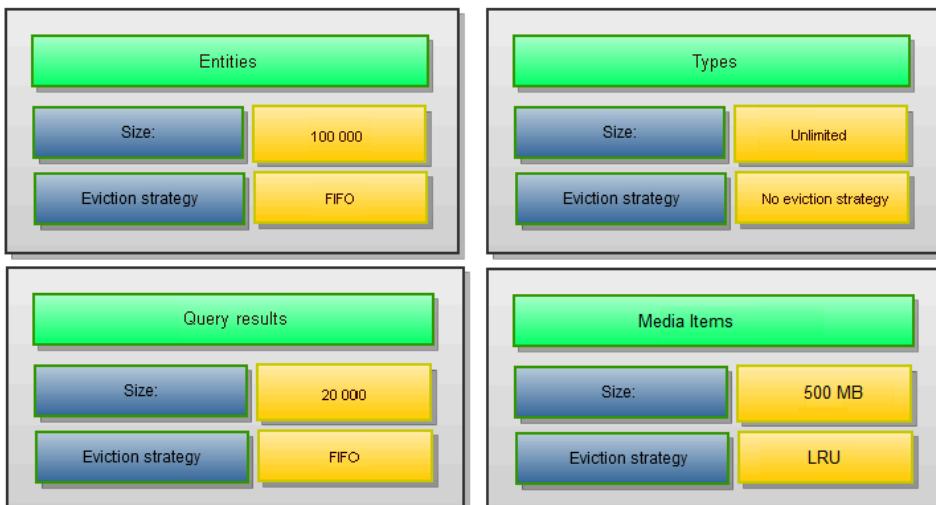
## Example Scenario

The SAP Commerce Region Cache is already preconfigured, but may be easily customized. This section presents default cache region settings and an example scenario of modifying those settings.

## Standard Configuration

The default setup of the SAP Commerce Region Cache contains the following regions:

- Type system region: For storing entities of type system items
- Entity region: For storing entities of all types except type system ones
- Query results region: For storing all query results
- Media items region: For storing all media items
- Session region: For storing HTTP sessions



## Extended Configuration

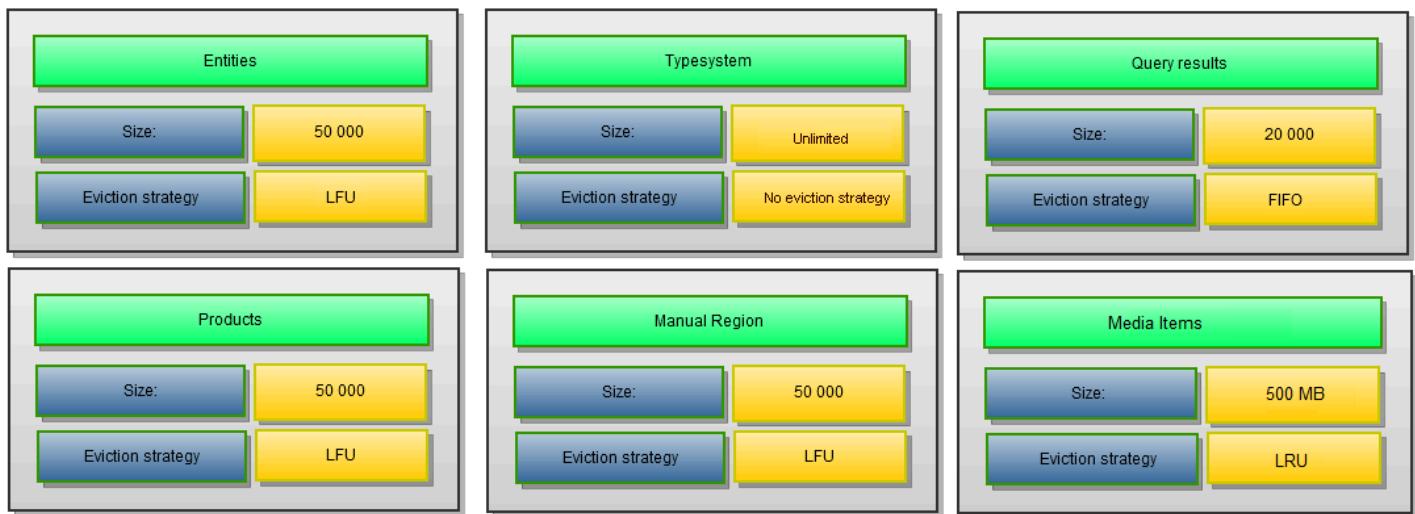
It's possible to easily extend the default configuration of cache regions. In this example, the standard configuration is modified in the following way:

- A cache region is added for storing product entities.
- A manual cache region is added.
- The size of the cache region for all entities is decreased to 50000.

### i Note

Because of the numerous number of type system items, the default size of the type system region shouldn't be decreased.

Make sure that the size of the type system query results region is always larger than the number of type system items multiplied by three. Such a configuration ensures that all needed queries that are stored in this cache region aren't evicted prematurely.



Manual cache regions aren't controlled by the cache framework, but by user custom logic. You can decide when and what has to be stored in it. Additionally, you can control when manual region entities are invalidated.

## Configuring Region Cache

The SAP Commerce Region Cache is configured in the `advanced.properties` and `core-cache.xml` files.

### Modifying Default Region Settings

Default settings can be modified:

1. By changing values in `local.properties`.
2. By overriding the cache bean in your `spring.xml` file using an alias.

To change preconfigured cache region settings, provide new values in the `local.properties` for the preconfigured parameters, for example :

`local.properties`

```
# Size of a region that stores all other, non-typesystem and non-query objects. Default value is 1000
regioncache.entityregion.size=50000

# Change eviction policy used by entity region. Possible values are F
regioncache.entityregion.evictionpolicy=FIFO
```

```
# LRU and LFU can be used for EhCacheRegion only. For other region type
regioncache.entityregion.evictionpolicy=LRU

# specifies root cache folder for all cached files
media.default.local.cache.rootCacheFolder=cache
# specifies max size of media cache in MB
media.default.local.cache.maxSize=500
```

Media Items Region extends the LRU Cache Region so, by default, it uses the LRU eviction strategy.

### i Note

Because of the numerous number of type system items, the default size of the type system region shouldn't be decreased.

Make sure that the size of the type system query results region is always larger than the number of type system items multiplied by three. Such a configuration ensures that all needed queries that are stored in this cache region aren't evicted prematurely.

## Adding New Cache Region

You can implement your own cache region with a cache implementation different than the default one. The following example presents a Spring configuration for your region cache that uses Ehcache implementation:

`example-spring.xml`

```
<bean name="productCacheRegion" class="de.hybris.platform.regioncache.region.impl.EHCacheRegion">
<constructor-arg name="name" value="productCacheRegion" />
<constructor-arg name="maxEntries" value="50000" />
<constructor-arg name="evictionPolicy" value="LFU" />
<property name="handledTypes">
<array>
<value>1</value>
</array>
</property>
</bean>

<bean name="manualCacheRegion" class="de.hybris.platform.regioncache.region.impl.EHCacheRegion">
<constructor-arg name="name" value="manualCacheRegion" />
<constructor-arg name="maxEntries" value="50000" />
<constructor-arg name="evictionPolicy" value="LFU" />
<property name="handledTypes">
<array>
</array>
</property>
</bean>
```

### → Tip

By setting the `handledTypes` parameter to 1 you actually map it to the `Product` item type, which has got the internal type code 1.

A cache region has the following properties:

- `name`: unique region name
- `maxEntries`: cache region size
- `evictionPolicy`: LRU, LFU, and FIFO
- `handledTypes`: list of types automatically stored in this region. To set up types use deployment code. Manual regions don't have any type configured

Additionally, new configured regions have to be connected with the Region Cache Spring configuration.

Cache regions are Spring beans holding objects of a specified type. Each region has its own set of types. It's also possible to have a region without a defined type of objects. However, such a region isn't taken into account by a controller and you need to implement a region resolver to be able to use it. To define a region type, you have the following options:

- Provide a deployment code list from `items.xml` file if you want that a region stores entities for the provided deployment code.
- Choose a dedicated type called `ALL_TYPES`. It denotes that a region stores all unconfigured types.

To configure a cache region to handle query results, use one of the following types:

- `QUERY_CACHE`: It denotes that a region stores query results.
- `NO_QUERY`: Used together with `ALL_TYPES` to denote that a region stores all unconfigured types except query results.

Query results and entities can be kept in the same region, but separation is recommended.

## SAP Commerce Cache API

See the API of the SAP Commerce Cache.

The SAP Commerce Cache API is backwards compatible, so for instance, single item invalidation is done by:

```
PK primaryKey = ....;
de.hybris.platform.util.Utilities.invalidateCache(primaryKey);
```

## Accessing Cache Region

You can access statistics for every cache region:

```
@Autowired
CacheController controller;
(...)
Collection<CacheRegion> regions = controller.getRegions();
for (CacheRegion region : regions)
{
    CacheStatistics stats = region.getCacheRegionStatistics();
    ...
}
```

## Clearing Cache

- Clearing all cache regions:

```
@Autowired
CacheController controller;
(...)
controller.clearCache();
```

- Clearing a single cache region:

```
CacheRegion region = ....;
region.clearCache();
```

## Customizing Region Cache

SAP Commerce allows you to customize region cache.

Main implementation elements that you may wish to customize are:

- **CacheController:** Main cache interface that contains methods coordinating cache operations on multiple regions.
  - **CacheConfiguration:** Provides configuration to the controller and it holds:
    - **CacheRegion:** Defines the interface of the cache region. Currently, the following implementations are provided:
      - **DefaultCacheRegion:** Simple FIFO cache region, which is the default for entity and query results regions
      - **EHCCacheRegion:** Cache based on Ehcache
      - **UnlimitedCacheRegion:** Unlimited cache region for type system
    - **CacheRegionResolver:** It is responsible for providing regions to the cache controller. There can be more than one region resolver in use.
- When reading from a database or adding to the database, the cache controller sequentially asks region resolvers for a given key. The first region that holds cached data is used. In case of invalidation, the cache controller asks all region resolvers and the invalidated value for the given key is removed from all returned regions.
- **InvalidationFilter:** The system can have more than one filter registered. Registered invalidation filters are processed one by one to check if all of them enable invalidation. If at least one filter disallows invalidation, data stays in cache until it is evicted. Invalidation filters are checked every time the invalidation is fired. It is possible to build time-based invalidation filters.
  - **CacheRegionProvider:** Keeps regions lists and provides regions list access methods.
  - **CacheStatistics:** Keeps statistics for a single cache region.

## License Restrictions on Cache Size

Depending on your license, you have a certain limit of total cache size for all regions. After extending the cache sizes without additional license, the Platform will not start and the following error message may be displayed on the console:

```
ERROR [WrapperSimpleAppMain] [RegionCacheAdapter] *****
ERROR [WrapperSimpleAppMain] [RegionCacheAdapter] Configuration is no
ERROR [WrapperSimpleAppMain] [RegionCacheAdapter] Total size for all
ERROR [WrapperSimpleAppMain] [RegionCacheAdapter] *****
shutting down hybris registry..
```

## Disabling the SAP Commerce Region Cache

The SAP Commerce Region Cache is the default cache. However, you can still use the previous cache solution by adding the following parameter to the `local.properties` file:

`local.properties`

```
cache.legacymode=true
```

It is recommended to use the SAP Commerce Region Cache. That is why after switching to the previous cache solution you can expect some warning messages displayed in the console logs and in the SAP Commerce Administration Console on Cache page.

## Maintaining queryCacheRegion

SAP Commerce allows you to clear entries in the `queryCacheRegion`.

As the data in the `queryCacheRegion` is removed only by displacement, the region reaches its size limit quickly. In order to clear its entries, you can implement a strategy that removes obsolete entries from the Flexible Search query cache at fixed intervals by using the dedicated interface: `CleanUpQueryCacheRegionStrategy` (with `DefaultCleanUpQueryCacheRegionStrategy` as the default implementation). You can define the value of the interval in the `queryRegionCache.clean.interval` property (it can be modified at run time). You can enable or disable this strategy by changing the value of the property `queryRegionCache.clean.enabled`. The value is enabled by default.

All query results, including results of queries with references to the type system items, are cleared by `CleanUpQueryCacheRegionStrategy`

### **i Note**

When the type system query results region cache reaches 90 or more percent of its size even after being cleared, a warning with a suggestion to increase the size of the region is thrown.

## Specifying a Minimum TTL for Cached FlexibleSearch Results

Platform offers a mechanism to enforce a minimum time to live for cached `FlexibleSearch` query results.

This mechanism keeps the results of `FlexibleSearch` queries in the SAP Commerce main cache at least for the specified time to live even if the search results are invalidated. It means that query results are still taken from the cache while their referenced item types have already received changes (for example `SELECT ... FROM {PRODUCT} ...` being still cached despite `ProductModels` having been created, updated, or deleted).

This can result in noticeable performance advantage for read-mostly scenarios such as storefronts rendering product data, which would otherwise suffer from query cache invalidation in case of frequent or bulk updates in SAP Commerce.

Depending on the nature of changes being done while TTL queries stay alive, the application may need to cope with stale data. For example when products are being deleted, those cached queries may still contain stale primary keys.

To define the minimum time to live, set the `cache_ttl` attribute for `SessionService` to a value specified in seconds, such as:

```
// the CACHE_TTL parameter is set to 10 seconds;
// search result of the following query will stay inside the cache for at least 10 sec

String qry = ...

private FlexibleSearchService flexibleSearchService;
private SessionService sessionService;
private String qry;

public void searchWithTTL()
{
    sessionService.executeInLocalViewWithParams(ImmutableMap.of(FlexibleSearch.CACHE_TTL,
        @Override
        public Object execute() {
            return flexibleSearchService.search(qry).getResults();
        }
    });
}
```

# Charon

In addition to standard HTTP requests, Charon can use OAuth to retrieve a secure token from an external authentication service. This token is used to authenticate subsequent requests on the remote service.

## Get Charon

To get Charon, use the following dependency configuration:

```
<dependency>
    <groupId>com.hybris</groupId>
    <artifactId>charon</artifactId>
    <version>1.2.0-M0</version>
</dependency>
```

## Remote Service Definition

The External Service pubsub service:

```
@Http(value = "pubsub", url = "https://${environment}/hybris/pubsub/v1")
@OAuth
@Control(timeout = "${timeout:4000}")
public interface PubSubClient
{

    @POST
    @Path("/topics/${client}/order-created/read")
    @Produces("application/json")
    @OAuth(scope = "hybris.pubsub.topic=hybris.order.order-created")
    Observable<RawResponse<ReadResponse>> readOrderCreated(ReadRequestParameters options);

    @POST
    @Path("/topics/${client}/order-created/commit")
    @Produces("application/json")
    @Control(retries = "3", retriesInterval = "2000")
    @OAuth(scope = "hybris.pubsub.topic=hybris.order.order-created")
    Observable<Void> commitOrderCreated(CommitRequestParameters options);
}
```

The External Product Service:

```
@Http(url = "https://api.external.io/hybris/product/b1")
@OAuth
public interface ProductClient
{
    @POST
    @Produces("application/json")
    @Path("/${tenant}/products")
    Observable<Void> createByMap(Map<String, Object> productMap);

    @POST
    @Produces("application/json")
    @Path("/${tenant}/products")
    Observable<Void> create(Product product);

    @GET
    @Produces("application/json")
    @Path("/${tenant}/products/?q=sku:{sku}")
    Observable<List<Product>> findBySku(@PathParam("sku") String sku);
}
```

- `@Http` is used in conjunction with `@EnableHttpClients` for automatic client discovery.
- "pubsub" in `@Http` is the clientId and it is used as configuration key's prefix.

- **@OAuth** means all the requests made within that client are subject of oauth authentication.
- **\${tenant}** is a config placeholder. Configuration provider is inquired for that specific key (tenant).

Charon is responsible to proxy those interfaces and making HTTP calls accordingly to the JAX-RS annotations and the dynamic configuration provided.

To properly set up Charon you need to:

- construct your interface(s) for define the REST client interface
- instantiate the client within the Charon builder or spring context
- inject the proper configuration management

## Creating a Client

1. Get Charon api:

```
ProductClient client = Charon.from(ProductClient.class).build()
```

2. Configure Sprig Java:

```
@Configuration
@EnableHttpClients(classes = {ProductClient.class})
```

**i Note**

At startup time, using `@EnableHttpClients`, Spring scans the classpath on configured packages looking for all interfaces annotated with `@Http` then wrap them to the proxy.

3. Configure Spring XML:

```
<bean id="productClient" class="com.hybris.charon.HttpClientFactoryBean">
    <property name="type" value="com.hybris.client.ProductClient"/>
</bean>
```

## Using OAuth

When interfaces are annotated with `@OAuth` an authorization request is first performed against a remote authentication server. Upon successful authentication the returned TOKEN is then cached for its entire time to live, and used in each subsequent transaction against the remote service, therefore a client credentials authorization grant is performed.

`@OAuth` parameters may also be provided as hard-coded values or placeholders, the same way as `@Http` annotations. `@OAuth` configuration follows the same hierarchy of defaults. For more information, see the **Configuring Charon** section of this document.

Below is an example of configuration properties provided in the annotation. The properties could be hardcoded or make use of placeholders.

**i Note**

The token is available for **every** request made with Charon as long as the request contains the same `clientId` and `scope`, and the request is made within the token time to live.

- In the case the token is used against a service, and rejected (by the oauth2 resource server) within `NotAuthorizedException` (401 Not Authorized), Charon will perform the oauth2/token request again.
- All client-based exceptions (not the >500 error codes) are remapped in `OAuthException`. The original exception is mapped in `OAuthException.getCause()`.

```
@OAuth(clientId = "...", clientSecret = "...", scope = "...", url = "...") //all annotation properties
```

OAuth is also able to make use of the `serviceId` provided in the `@Http` annotation. In other words, if no properties are provided for `@OAuth`, then the `serviceId` provided in the `@Http` client interface annotation are used.

```
//client interface
@Http("myClient")
@OAuth

//if properties are not provided in the annotation, they will be taken from the given configuration.
```

The above example would therefore look for properties provided as follows:

```
myClient.oauth.url=https://accounts.google.com/o/oauth2/auth
myClient.oauth.clientId=...
myClient.oauth.clientSecret=...
myClient.oauth.scope=...
```

OAuth annotation can be specified also at method level, and it overrides the values present at class level:

```
@Http("order")
@OAuth // all oauth inputs are specified in the given property resolver
@Control(timeout = "${timeout:4000}")
public interface OrderClient
{
    @GET
    @Path("/{tenant}/salesorders/{orderId}")
    Observable<Order> getOrder(@PathParam("orderId") String id);

    @POST
    @Path("/{tenant}/salesorders/{orderId}/transitions")
    @OAuth(scope = "hybris.order_update")
    Observable<Void> changeStatus(@PathParam("orderId") String id, OrderStatus newStatus);
}
```

## Connection Pooling

Charon reuses TCP connections for lowering latency and improving speed. The pool size for a given `host:port` is unlimited and new connection is created on concurrent requests, if not available in the pool. For more control on concurrent requests, see [Secure HTTP Transactions](#).

## Usage of JAX-RS Annotations

### HTTP Method

Charon supports the `@GET`, `@POST`, `@DELETE`, and `@PUT` methods. The default method is `@GET`.

The following sections show how to provide query strings for `GET` requests, and form parameters for `POST`. In addition, you can also provide custom headers and path parameters, and define the request content type.

The `@Produces` annotation defines the request's content type. In the case of `application/json` only the first argument without annotation (`@FormParam`, `@QueryParam`, `@HeaderParam`) is to be serialized. If more than one argument without annotation is

present, an exception is thrown.

```
@Produces("application/json")
void create(Wishlist wishlist);
```

To include a **GET** query string in the URI, provide the name-value pairs in the **getAll()** method as follows. This adds the query parameters as a form-url-encoded series of **<query-param-name>=<argument-value>**.

```
@GET
@Path("/order")
Observable<Order> getAll(@QueryParam("pageNumber") int pageNumber);
```

To submit form values for the **POST** method, provide the field name-value pairs in the **postForm()** method as follows:

```
@POST
void postForm(@FormParam("fp1") String param1, @FormParam("fp2") String param2);
```

When **@FormParam** arguments are present, the **application/x-www-form-urlencoded** content type is sent to the server within the url-encoded parameters.

Path parameters may be injected into the URI using the **getOrder()** method as follows:

```
@Path("/order/{id}")
Observable<Order> getOrder(@PathParam("id") String id);
```

You can also provide headers, using the following method:

```
@Header( name="X-appId", val="static-value-${customKeyValue}")
Observable<Product> retrieveProduct()
```

Custom headers can also be provided using the **getWithHeaders()** method:

```
String getWithHeaders(@HeaderParam("header1") String header1, @HeaderParam("header2") String header2)
```

For performing a basic authentication per request:

```
Observable<Result> getResult(@HeaderParam("Authorization") String basic);
//basic="Basic QWxhZGRpbjpCGVuU2VzYW1l"
```

For performing a basic authentication by configuration:

```
@Header( name="Authorization", val="Basic ${basicAuth}")
Observable<Result> getResult();
//basic="QWxhZGRpbjpCGVuU2VzYW1l"
```

## Request body encoding

The encoding of the request body is specified by the **@Produces** annotation. Only two encoders are currently provided:

- plain/text

Simply invoke `.toString()` on the request object

- application/json

Serialize the given object with Jackson

- multipart/form-data uploads files and mixed parts

@POST

```
@Produces("application/json") //the POJO `post` argument is serialized in json
Observable<Void> createPost(Post post);
```

The decoding of the response is driven by the **Content-Type** header sent by the server. However, some arguments are self-explanatory regarding their encoding.

Type Based Encoding utilizes the following:

- `java.net.URL`: the remote content is fetched and redirected to the target endpoint, as long as it is `<content-type>` and `<content-length>`
- `java.nio.file.Path`: the content-type is deducted by `Files.probeContentType` and it is forward along with the binary data
- `java.io.File`: as `<Path>`
- `java.io.InputStream`: the stream is redirected to the target endpoint, though the `<content-type>` is by default `application/octet-stream` and the request will be chunked
- `java.lang.String`: the string is sent as is with `<content-type: text/plain>`
- `byte[]`: the byte array is sent as is with `content-type: application/octet-stream`

```
//content-type is determined by the file itself
@POST
Observable<Response> sendFile(Path path)

//send a text file
sendFile(Path.get("/tmp/test.txt"))

//will send
content-type: text/plain
content-length: ...

//content-type could be overrided by @Produces annotation
@POST
@Produces("application/json")
Observable<Response> sendFile(Path path)

//will send
content-type: application/json
...
```

For adding or replacing the default encoders, you may use the builder:

```
Typicode service = Charon.from(Typicode.class)
.putEncoder(MediaType.APPLICATION_XML, (EncoderRequest req)-> ....) // XML encoder
.putDecoder(MediaType.APPLICATION_XML, (Observable<byte[]> bytes, Type type)->...) // XML de
.build();
```

where `EncoderRequest` contains arguments and parameters, and `EncodeResult` contains the headers and the content.

The encoder is a `Function<EncodeRequest, Observable<EncodeResult>>`.

The decoder is a `BiFunction<Observable<byte[]>, Type, Observable<Object>>`.

```
BiFunction<Object[], Parameter[], byte[]> firstArgumentToStringEncoder = (args, parameters)-> args[0]
```

## Annotation-Based Configuration

```

@Encoder(contentType = "application/xml", encoder = XMLEncoder.class)
@Decoder(contentType = "application/xml", decoder = XMLDecoder.class)
public interface Test{
    @POST
    @Path("/")
    @Produces("application/xml")
    PojoResponse post(PojoRequest val);
}

```

The decoder type is a **BiFunction<ByteBuf, Type, Object>**.

```
BiFunction<ByteBuf, Type, Object> toStringDecoder = (buffer, type)-> buffer.toString(defaultCharset())
```

The decoder function takes the **ByteBuf** and the **Type** of the returned object as input, then returns the actual **Type** instance.

## Upload Files and Multipart Requests

### MediaClient

```

@Http(url = "https://api.external.io/hybris/media/b2")
@OAuth(scope = "hybris.media_manage")
interface MediaClient {
    @POST
    @Path("/{tenant}/{client}/media")
    @Produces("multipart/form-data")
    Observable<MediaCreateResponse> upload(@Disposition URL from);

    @POST
    @Path("/{tenant}/{client}/media")
    @Produces("multipart/form-data")
    Observable<MediaCreateResponse> upload(@Disposition URL from, @Disposition(contentType = "application/json", name = "metadata", fileName = "report.json"));
}

```

File upload is performed through multipart/form-data requests and with Charon it is possible to specify the details for every part defined as **Content-Disposition** in HTTP.

```
@Disposition(contentType = "application/json", name = "metadata", fileName = "report.json")
```

The **@Disposition** annotation defines the following properties:

- <*ContentType*>: the HTTP header included in the request and by which the parameter is serialized. This value must be specified, there is no default value.
- <*Name*>: the **name** field for that **Content-Disposition**. Default: <*file*>.
- <*FileName*>: the **filename** field for that **Content-Disposition**. Default: <*fileName*>.

In the case of URL type parameter:

- <*ContentType*> is given by remote endpoint **Content-Type**, and it overrides the **@Disposition <contentType>** value.
- <*FileName*> is given by the <*file*> url part, if it is present.

In case of <*Path/File*> type parameter:

- <*ContentType*> is given by the type of file
- <*FileName*>: the file/s name

In case of <*String*> type parameter:

- <Content-Type>: text/plain

In case of <byte[]> type parameter:

- <Content-Type>: application/octet-stream

## Upload without MultiPart

```
@POST
@Path("s3-url")
Observable<S3Response> upload(URL fromEndpoint);
```

Similarly to the multi-part request, Charon uploads the content from `fromEndpoint` to the specified target as a single-part request. The `<content-type>` and `<content-length>` will be the one returned by the typed parameter (see the Type-Based Encoding paragraph)

## Error Handling

The exceptions thrown by Charon are the standard exceptions defined in `com.hybris.charon.exp` package. These are listed in the following table.

Http Code	Exception
304	NotModifiedException
300–399	RedirectException
400	BadRequestException
401	NotAuthorizedException
403	ForbiddenException
404	NotFoundException
405	NotAllowedException
406	NotAcceptableException
409	ConflictException
500	InternalServerErrorException
503	ServiceUnavailableException
505	NotSupportedException
50*	ServerErrorException

Http Code	Exception
client errors in oauth2/token	OAuthException
*	java.lang.RuntimeException

## Logging

The Platform makes use of the Loopback logger for logging purposes. For dumping all HTTP activities during development phases, the logger category has to be set to debug level.

```
<logger name="io.netty.handler.logging.LoggingHandler" level="debug" />
```

This outputs hex content similar to the following:

```
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 50 4f 53 54 20 68 74 74 70 3a 2f 2f 6c 6f 63 61 |POST http://loca|
|00000010| 6c 68 6f 73 74 3a 38 30 38 30 2f 6f 61 75 74 68 |lhost:8080/oauth|
|00000020| 2f 74 6f 6b 65 6e 20 48 54 54 50 2f 31 2e 31 0d |/token HTTP/1.1.|
|00000030| 0a 43 6f 6e 74 65 6e 74 2d 54 79 70 65 3a 20 61 |.Content-Type: a|
|00000040| 70 70 6c 69 63 61 74 69 6f 6e 2f 78 2d 77 77 77 |pplication/x-www|
|00000050| 2d 66 6f 72 6d 2d 75 72 6c 65 6e 63 6f 64 65 64 |-form-urlencoded|
|00000060| 0d 0a 43 6f 6e 74 65 6e 74 2d 4c 65 6e 67 74 68 |..Content-Length|
|00000070| 3a 20 38 37 0d 0a 41 63 63 65 70 74 3a 20 61 70 |: 87..Accept: ap|
|00000080| 70 6c 69 63 61 74 69 6f 6e 2f 6a 73 6f 6e 2c 74 |plication/json,t|
|00000090| 65 78 74 2f 70 6c 61 69 6e 0d 0a 48 6f 73 74 3a |ext/plain..Host:|
|000000a0| 20 6c 6f 63 61 6c 68 6f 73 74 3a 38 30 38 30 0d | localhost:8080.|
|000000b0| 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 52 78 4e |.User-Agent: RxN|
|000000c0| 65 74 74 79 20 43 6c 69 65 6e 74 0d 0a 0d 0a |etty Client.... |
+-----+
```

## SAP Commerce Data Hub Integration

For integrating Charon with the SAP Commerce Data Hub in a servlet engine, follow **Spring XML Integration** section of the [Charon Spring Integration](#) page.

You should keep all the configuration in the **local.properties** and let Charon to grab the configuration from there using the **HybrisCommonsResolver**. An Example could be:

- Pubsub client

```
@OAuth
public interface PubSubClient
{

    @POST
    @Path("/topics/${client}/{topic}/read")
    @Produces("application/json")
    @Control(timeout = "${timeout:2000}")
    Observable<ReadResponse> read(@PathParam("topic") String topic, ReadRequestParameters option

}
```

- **local.properties**

```
external.pubsub.url=https://api.external.io/hybris/pubsub/b2
external.pubsub.client=hybris.order
external.pubsub.oauth.clientId=....
external.pubsub.oauth.clientSecret=....
external.pubsub.oauth.scope=....
```

- ext-datahub-spring.xml

```
<bean id="pubsubClient" class="com.hybris.charon.HttpClientFactoryBean">
<property name="type" value="com.hybris.client.PubSubClient"/>
<property name="clientId" value="pubsub"/>
<property name="propertyResolver">
<bean class="com.hybris.charon.conf.HybrisCommonsResolver">
<construct-arg value="external"/>
</bean>
</property>
</bean>
```

## Proxy Support

It is possible to perform request through an HTTP proxy following the system properties provided by Oracle:

- The following proxy settings are used by the HTTP protocol handler:

- http.proxyHost** (default: <none>)

The hostname, or address, of the proxy server.

- http.proxyPort** (default: 80)

The port number of the proxy server.

- http.nonProxyHosts** (default: localhost|127.\*|[:1])

Indicates the hosts that have to be accessed without going through the proxy. Typically this defines internal hosts. The value of this property is a list of hosts, separated by the | character. In addition, the wildcard character \* can be used for pattern matching. For example -Dhttp.nonProxyHosts="\*.foo.com|localhost indicates that every host in the foo.com domain and the localhost should be accessed directly even if a proxy server is specified. The default value excludes all common variations of the loopback address.

- The following proxy settings are used by the HTTPS protocol handler:

- https.proxyHost**(default: <none>)

The hostname, or address, of the proxy serve.

- https.proxyPort** (default: 443)

The port number of the proxy server.

The HTTPS protocol handler uses the same **nonProxyHosts** property as the HTTP protocol.

## Request Throttling

Requests performed by Charon are natively asynchronous, that means if Charon is invoked in a hypothetical infinite loop, the client will finally run into a **too many open files** exception or the target server will reject the large amount of concurrent requests. For that case, it is possible to limit concurrent requests with the @Control annotation:

```
@Control(maxConcurrentRequests = "50") //50 is intended for in a single method
interface ProductServiceClient{

    @Control(maxConcurrentRequests = "20") //overrides the interface-level attribute
    Observable<ProductCreationResponse> createProduct(Product product)
}
```

By default it throws an `com.hybris.charon.exp.ExhaustedPoolException` that could be caught in the `<.onError() Observable>` method. For specifying a different error handling there are the following options:

```
//exceptionInline throws the exception in the method
@Control(maxConcurrentRequests = "50", exhaustedPoolStrategy = ExhaustedPoolStrategy.exceptionInline)
Observable<ProductCreationResponse> createProduct(Product product) throws ExhaustedPoolException;

//disabled will disable request throttling for that specific method
@Control(exhaustedPoolStrategy = ExhaustedPoolStrategy.disabled)
Observable<Void> unlimited(...);

//wait for available connections when pool is fully in action
@Control(exhaustedPoolStrategy = ExhaustedPoolStrategy.blocking, blockWait="2000")
Observable<ProductCreationResponse> createProduct(Product product) throws ExhaustedPoolException;
```

When the `<strategy>` is set to `<blocking>`, the invoking thread will be suspended until an available connection will be returned to the pool or the `blockWait` will elapse after a configurable number of milliseconds (1000ms by default). When the `blockWait` expires, an `ExhaustedPoolException` is thrown.

## Getting Headers from Response

Declare `com.hybris.charon.RawResponse` as return type:

```
@POST
@Path("/topics/${client}/{topic}/read")
@Produces("application/json")
Observable<RawResponse<ReadResponse>> read(@PathParam("topic") String topic, ReadRequestParameters or
```

In `RawResponse` the content is accessible through: `Observable<T> content();`

All major headers are accessible to their specific methods:

```
Optional<Integer> contentLength();
```

```
Optional<String> contentType();
```

```
List<Link> links();
```

```
Optional<Long> age();
```

```
List<HttpMethod> allows();
```

```
List<CacheControl> cacheControls();
```

```
List<String> contentEncodings();
```

```
List<Locale> contentLanguages();

Optional<Date> date();

Optional<String> eTag();

Optional<Date> expires();

Optional<Date> lastModified();

Optional<URL> location();

Optional<String> server();

List<NewCookie> getSetCookies();

Response.Status status();
```

While help methods can enable developer to access custom headers:

```
Optional<Long> headerLong(String header);

Optional<Integer> headerInt(String header);

Optional<Boolean> headerBoolean(String header);

Optional<Date> headerDate(String header);

Multimap<String, String> headers();
```

## Testing

It is possible to run integration tests which perform real HTTP request to a test web server:

```
com.hybris.charon.TestTools.runTest(TestUtils.ServerHandler requestHandler, TestUtils.Task task) throws
```

For example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(loader = AnnotationConfigContextLoader.class, classes = {FeatureIT.Config.class})
public class FeatureIT
{
    @Inject
    TestLocal client1;

    @Test
    public void simpleGet() throws Throwable
    {
        TestUtils.runTest((req, resp, content) -> {
            //server side
            //incoming request from the client
            resp.setStatus(HttpStatus.OK);
        });
    }
}
```

```

        TestUtils.writeString(resp, "ciao");
        return resp.close();
    }, () -> {
        //client side
        //execute client calls
        assertThat(client1.getSimpleResponse()).isEqualTo("ciao");
    });
}

@Http
interface TestLocal
{
    @GET
    String getSimpleResponse();
}

@Configuration
@EnableHttpClients(classes = {TestLocal.class})
@PropertySource("classpath:/test.properties")
static class Config
{
    @Bean
    SpringEnvironmentResolver propertyResolver()
    {
        return new SpringEnvironmentResolver("external");
    }
}
}
}

```

To use it you must include the following artifacts:

```

<dependency>
    <groupId>com.hybris</groupId>
    <artifactId>charon</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <scope>test</scope>
    <type>test-jar</type>
</dependency>
<dependency>
    <groupId>io.reactivex</groupId>
    <artifactId>rxnetty</artifactId>
    <version>0.4.15</version>
    <scope>test</scope>
</dependency>

```

## Configuring Charon

Charon can be used out of the box with no environment, or with Spring. Configuration is primarily annotation-based, however with Spring you may also use XML.

### Annotation-Based Configuration

The URL and **clientId** values can be declared in the interface using the `@Http` annotation.

```
@Http(value = "pubsub", url = "https://api.external.io/hybris/pubsub/b2")
interface PubSubClient {
...
}
```

The `@Http` annotation may be static as per the above example, or you can define a unique client ID for the entire remote service. This is a key the system uses for finding all associated properties. For example:

```
@Http('pubsub')
```

Given this `clientId`, the system looks for the following properties:

```
pubsub.url = ...
pubsub.oauth.url = ...
```

You may use standard Spring-style placeholders of the form  `${VALUE}` for some or all parts of the property.

```
//placeholders are similar to Spring. Use default value after ':'
@Http(value= "pubsub", url = "https://${environment}/hybris/pubsub/b2")
```

And your property file looks like:

```
#pubsub.environment=stage.external.io
environment=api.external.io
```

## i Note

You can also define a default value for each placeholder, providing this after the colon (:)

## Dynamic Configuration with PropertyResolver

- Using a Given Map

You may provide the property values as a map. In this example the Guava's `ImmutableMap.of` method provides a simple way to build a map with a single key-value pair:

```
...
PubSubClient client = Charon.from(PubSubClient.class).config(ImmutableMap.of("environment", "ap
```

- Using a Properties Object

The properties object is a representation of a properties file, usually this is a filepath, for example `load(String FILEPATH)`.

```
Properties props =...
Typicode client = Charon.from(Typicode.class).config((Map)props)).build();
```

```
<bean id="client" class="com.hybris.charon.HttpClientFactoryBean"
  p:clientId="pubsub"
  p:config-ref="map"/>
```

```
<util:map id="map">
  <entry key="environment" value="api.external.io"/>
</util:map>
```

- Using a Spring Property Resolver

In Spring, you can use the **SpringEnvironmentResolver** to define different properties per object or path.

```

@.Inject
SpringEnvironmentResolver resolver;

Typicode client = Charon.from(Typicode.class).config(resolver).build();

@Configuration
@PropertySource("classpath:*config.properties")
@EnableHttpClients(classes = {PubSubClient.class})
public class SpringConfig {
    @Bean
    public SpringEnvironmentResolver resolver(){
        return new SpringEnvironmentResolver("prefix");
    }
}

```

- Using the SAP Commerce ECP Configuration Service

A platform extension is available for handling through the ECP configuration service. The property resolver implementation is named **platformPropertyResolver** and it is configured as below.

```

<bean id="platformPropertyResolver" class="de.hybris.platform.yaas.config.PlatformPropertyResol
    <constructor-arg value="external" /> <!-- example configuration prefix -->
    <constructor-arg ref="configurationService" />
</bean>

```

- Using a Custom Property Resolver

Alternatively, you may create your own custom property resolver.

```

PropertyResolver resolver = new PropertyResolver(){
    boolean contains(String key){...}
    String lookup(String key){...};
}
PubSubClient client = Charon.from(PubSubClient.class).config(resolver).build();

```

- Using SAP Commerce Data Hub configuration

On datahub extension, you may use the **local.properties** defined in configuration, in order to do this, you may simply add the **HybrisCommonsResolver**:

```
<bean class="com.hybris.charon.conf.HybrisCommonsResolver"/>
```

## Default Values and the Configuration Property Hierarchy

Configuration properties are hierarchical and are first searched by the annotations. For example:

```
@Http( url = "http://www.example.com" )
```

The configuration hierarchy follows this order:

1. annotation - hard-coded or dynamic with placeholders
2. provided configuration

### 3. system properties

The properties file may contain generic property values as well as those identified by **clientId**. For example:

```
myClient.url = ...
```

In this case if no matching **clientId** is provided, the generic property values for url and port are used. If generic property values are not provided then the system default is used. Examples of these three types of property definition are provided below.

```
@Http("myClient")
...
#properties example
myClient.url=http://jsonplaceholder.typicode.com

#if myClient.url is not present it will lookup for url property
url=http://jsonplaceholder.typicode.com
```

## Related Information

[Charon Spring Integration](#)

[Control Over Reliability](#)

## Charon Spring Integration

Charon can be deployed in your application using Spring Integration. This can be configured using either Java annotations or Spring XML.

## Java Configuration

Below is a sample configuration using Java. The annotation `@EnableHttpClients` scans the base package for the list of classes and registers them with the `HttpClientFactoryBean`. The proxied interface is then available in the spring context.

`@EnableHttpClients` has the following attributes:

- `basePackages (String[])`
- `basePackageClasses (String[])`
- `classes (Class[])`

With the use of the `SpringEnvironmentResolver`, Spring injects its own configuration system. All properties are then managed automatically and transparently. This is not possible with the XML configuration.

```
// spring java config
@Configuration
@EnableHttpClients // charon specific annotation
@PropertySource(.....)
//ex:
//my.client.url=...
//my.client.oauth.url=...
//my.client.oauth.clientId=...
//my.client.oauth.clientSecret=...
//my.client.oauth.scope=...
public class Config {
    @Bean
    SpringEnvironmentResolver propertyResolver(){
        return new SpringEnvironmentResolver("my"); //indicating the prefix for properties
    }
}
```

```
// example client interface
@Http("myClient")
@OAuth
interface Typicode{...}

// business service
@Service
public class MyService{
    @Inject
    Typicode client; //auto injected by spring
}
```

## Spring XML Configuration

Below are two possible sample Spring XML configurations. The first loads a properties file as a properties object, while the second uses a custom map. Both examples are based on the following properties file:

### xml.properties

```
my.client.url=...
my.client.oauth.url=...
my.client.oauth.clientId=...
my.client.oauth.clientSecret=...
my.client.oauth.scope=...
```

### Loading Properties Files

With this method, you can explicitly instruct Spring to load a properties file and treat it as a properties object:

```
<util:properties id="props" location="classpath*:xml.properties" />

<bean class="com.hybris.charon.HttpClientFactoryBean">
    <property name="type" value="" />
    <property name="propertyResolver" >
        <bean class="com.hybris.charon.conf.PropertiesResolver">
            <constructor-arg value="my" />
            <property name="properties" ref="props" />
        </bean>
    </property>
    <property name="serviceId" value="client" />
</bean>
```

### Using PropertyPlaceholderConfigurer for Creating a Custom Map

With this method, you create a properties map. By using placeholders, Spring will look up the properties in the Spring context based on the provided key.

Spring cannot build the map automatically using XML configuration as it does with the Java example. Define the map explicitly, as below:

```
<bean class="com.hybris.charon.HttpClientFactoryBean">
    <property name="type" value="" />
    <property name="propertyResolver" >
        <bean class="com.hybris.charon.conf.MapPropertyResolver">
            <constructor-arg name="map">
                <util:map key-type="java.lang.String" value-type="java.lang.String">
                    <entry key="url" value="${my.client.url}" />
                    <entry key="oauth.url" value="${my.client.oauth.url}" />
                    <entry key="oauth.clientId" value="${my.client.oauth.clientId}" />
                    <entry key="oauth.clientSecret" value="${my.client.oauth.clientSecret}" />
                    <entry key="oauth.scope" value="${my.client.oauth.scope}" />
                </util:map>
            </constructor-arg>
        </bean>
    </propertyResolver>
</bean>
```

```

        </bean>
    </property>
</bean>
```

The `<util:map>` tag is declared by adding :

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util" xsi:schemaLocation=
           "http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
           http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-
</beans>
```

## Control Over Reliability

The `@Control` annotation allows you to pass parameters related to improving the reliability of your service. This includes defining timeouts and retries. As with the other annotations, parameters may be either hard-coded or placeholders.

### Timeouts

You may define a timeout value, expressed in milliseconds. It is recommended that you throw a `TimeoutException` on timeout, otherwise a `RuntimeException` is thrown.

```

@GET
@Path("/posts")
@Control(timeout = "2000") //expressed in milliseconds. ${} placeholders allowed
List<Post> getPosts() throws TimeoutException; //methods with timeout should throw TimeoutException
```

### Retries

In case of a request failure such as network problems or HTTP error response status, you can specify that the system retry the request after a given interval and also specify how many retry attempts have to be made. The retry interval is expressed in milliseconds.

If no value is provided for retries, the default is zero. That is, no retries are attempted.

```

@GET
@Path("/posts")
@Control(retries = "3", intervalRetries = "500") //default for intervalRetries is 200. ${} placeholders allowed
List<Post> getPosts();
```

## Control Over Void Methods

In the following example, the `createPost` method is executed asynchronously, so the method returns immediately.

```

@POST
@Consumes("application/json")
void createPost(Post post);
```

To prevent this and to allow for concurrent, parallel transactions, you can define your method as observable, as follows:

```

@POST
@Consumes("application/json")
```

```
Observable<Void> createPost(Post post);
```

Then it is up to the Observable owner to start the call, which begins after the subscription:

```
service.createPost(post).subscribe(next - {}, error -> log.error(e.getMessage(), e), () -> {
    //post completed
});
```

See the [ReactiveX Documentation](#) on Observable at reactivex.io for more details.

## Clustered Environment

The SAP Commerce Cluster is a number of individual SAP Commerce installations using a common set of data on one database. A cluster configuration lets you balance the load between the nodes to scale SAP Commerce in a way that is transparent to the user.

The cluster functionality offers a wealth of configurable options, such as:

- Choice of the communication protocol (UDP: both multicast or unicast, or JGroups)
- Node-specific runtime configurations
- Cron jobs for specific nodes
- Session failover
- Advanced logging and debug modes
- Encryption capability

An SAP Commerce cluster also enables you to run SAP Commerce inside a cloud environment.

For a detailed description of setting up SAP Commerce in a cluster environment, see [Configuring a SAP Commerce Cluster](#).

In a cluster, the individual SAP Commerce is referred to as a **node**. Each node uses an individual SAP Commerce instance running in an individual JVM.

### **i** Note

It is worth noting that running SAP Commerce in cluster mode is not the same as running it in a multi-tenant mode. The SAP Commerce Cluster is a number of individual, separate SAP Commerce instances sharing one single set of data, whereas the multi-tenant SAP Commerce is one single SAP Commerce using separate sets of data. For a discussion of running SAP Commerce in multi-tenant mode, refer to [Multitenancy](#).

For a general discussion of caching in the clustered SAP Commerce, see [Caching](#).

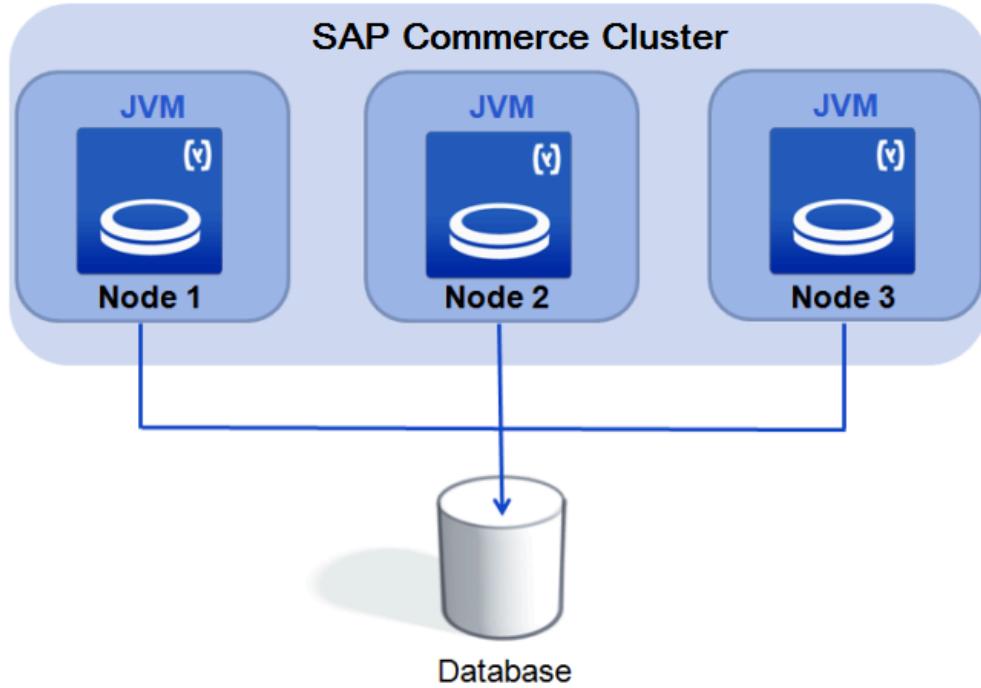


Figure: All nodes within a cluster use the same database.

An SAP Commerce cluster offers an array of features to ensure high availability of your implementation, while preserving transparency to the end user.

## Cache Invalidation Concept

The core of a SAP Commerce Cluster is a network communication system that makes sure the cache of each individual cluster member only holds valid data. The nodes in the SAP Commerce Cluster communicate using TCP (JGroups) or UDP by sending signals to other nodes that mark some cache entries as invalid because a database item has been changed. The following is the overview of the cache invalidation process:

1. A description of a product has changed. Therefore, all cache entries referring to the product are invalid.
2. A cluster node where the modification has been done sends a notification to all cluster nodes that all cache entries holding the product are invalid.
3. Nodes that hold the product in their cache discard the cached data of the product and re-retrieve the product from the database the next time the product is used.

For more information, see:

- [Caching](#)
- [HTTP Session Failover](#)

## Choice of UDP Multicast, UDP Unicast, or JGroups as Communication Protocols

Consult your network administrator to choose the right protocol for your deployment.

There are pros and cons to each solution. Generally, JGroups provides the fastest communication and can be used both in the cloud for testing purposes and in a LAN or WAN network. UDP Multicast has some limitations in that some routers do not allow multicast network traffic by default. Amazon Elastic Compute Cloud (Amazon EC2) does not work using UDP, either.

## Load Balancing

To balance the load between two cluster nodes means that user requests are balanced between all of the nodes, instead of being sent to only one server.

If you need redundancy, use a hardware load balancer or a software load balancer. Read our [Third-Party Compatibility](#) document for possible options on what kind of load balancers can be used with SAP Commerce. How to use and set up a load balancer is beyond the scope of this document but for more information about load balancing, see [Transparent HTTP Session Failover](#).

## → Tip

In some configurations, you might want to use two nodes but without load balancing. For example, let's say you have one machine that is a cluster node for your front-end system, and another machine for Backoffice. They both are part of a cluster, which means they do not display invalid data, and you can access them directly by the IP address or hostname.

## HTTP Session Failover

Sessions used by clients that are bound to an individual cluster node, stick with the cluster node. Therefore, the SAP Commerce Cluster uses the so-called sticky sessions. In addition, SAP Commerce offers a session failover mechanism. For more details, see [HTTP Session Failover](#).

## Node-Specific Runtime Configurations

Nodes in a cluster may have different memory capacity or varying CPU performance. Using node-specific cluster settings enables the specifying of a custom configuration for each individual node, such as having varying cache size, or having different folder paths for media read or replication folders. For more details, see [Node-specific Configurations](#).

## Performance Optimizations

It is possible to tweak Windows settings to improve the performance of the SAP Commerce Cluster by modifying:

- Datagram size
- Thread scheduling

For more details see the *SAP Commerce Cluster Performance Tips* section in [Tips on Performance Tuning](#).

## Encryption Configuration

To guarantee data integrity, you have to be sure that every node is using the same encryption key. A general default key is part of the release and is stored at  **`${HYBRIS_BIN_DIR} /platform/ext/core/resources/security/default-128-bit-aes-key.hybris`**. However, we strongly recommend that you replace this weak and unsecure keyfile with your own key, which has to be placed in the  **`${HYBRIS_CONFIG_DIR} /security`** directory. For more details, see [Transparent Attribute Encryption \(TAE\)](#).

## Related Information

[Caching](#)

[Transactions](#)

<http://www.onjava.com/pub/a/onjava/2001/09/26/load.html> ↗

## Cluster FAQ

Setting up SAP Commerce in a cluster is not difficult, but there are many variables, and every installation is different. Answers to questions posed by others are available here to help you.

## What About 3rd Party Application Server Clustering Functionality? Do I Have to Enable It?

SAP Commerce clustering is completely independent of application server clustering. You do not need it to ensure the data integrity of your cache. We recommend disabling all application server clustering systems.

HTTP session failover is also independent of the SAP Commerce cache. However, if you need session failover, then it might be necessary to use the application server session clustering.

## Is the Cluster Transactional?

Yes. Clustering is not a matter of transactions. What we assure is the correct invalidation and isolation of our cache when using transactions.

For more details, see [Transactions](#), and the appropriate sections in [Caching](#).

## Can I Use a Cluster with Nodes on Different Hardware, Operating Systems, or Application Servers?

Short answer: Yes, however, SAP Commerce does not recommend that you do that.

Long answer: The SAP Commerce Clustering feature is not bound to specific application servers, operating systems, or physical hardware infrastructure.

If your deployment meets the requirements (can send and receive UDP multicast packets), you can set up a cluster with completely mixed components. However, SAP Commerce does not recommend such a setup for production environments. It is very hard to balance load between different servers, troubleshooting is more complicated, and the overall setup requires broad knowledge.

## Clustering

With SAP Commerce clustering you can run SAP Commerce inside a cloud environment such as Amazon EC2.

Clustering can be based on the JGroups protocol or UDP. Extended capabilities such as balancing the load between different cluster nodes or HTTP Session replication can be added by using [Transparent HTTP Session Failover](#).

## Configuring a SAP Commerce Cluster

Follow the steps to configure a SAP Commerce cluster.

### Context

You can choose between three main protocols:

- JGroups
- UDP Multicast
- UDP Unicast

To set up a cluster, configure each node individually by editing the `local.properties` file located in the `config` folder.

### Procedure

1. Modify `local.properties` to activate the cluster mode:

```
clustermode=true #the same for all nodes
```

2. Modify `local.properties` to define a unique `id` for the node:

```
cluster.id=0 #This needs to be unique for each node
```

3. Modify `local.properties` to specify the communication protocol. You can choose between JGroups (UDP or TCP), UDP multicast and UDP Unicast:

- `cluster.broadcast.methods=jgroups` #the same for all nodes; Within Jgroups you can select TCP or UDP
- `cluster.broadcast.methods=udp` #the same for all nodes; udp Multicast

4. Add protocol-specific settings depending on your choice of communication protocol. For more information, see [JGroups Settings](#) and [UDP Cluster Settings](#).

5. Test the connection between cluster nodes:

- a. Go to SAP Commerce Administration Console to make sure that the nodes can see each other. Select the **Monitoring** tab, **Cluster** to see a list of available nodes.
- b. Make a change in Backoffice on node#1 and make sure you can see the change in Backoffice on node#2

## Related Information

[Configuring the Behavior of SAP Commerce](#)

[API Documentation and YAML Files](#)

## JGroups Settings

See the information about JGroups features and settings.

JGroups:

- Is a Java toolkit for reliable multicast communication
- Can be used to create groups of processes whose members can send messages to each other
- Saves development time
- Has a flexible protocol stack so an application can be deployed in different environments without having to change code

Main features of JGroups:

- Membership
- Transport protocols: UDP (IP Multicast), TCP, JMS
- Fragmentation of large messages
- Reliable unicast and multicast message transmission. Lost messages are retransmitted
- Failure detection: crashed members are excluded from the membership
- Ordering protocols: Atomic (all-or-none message delivery), Fifo, Causal, Total Order (sequencer or token based)
- Encryption
- Much faster message transmission than with UDP or TCP only

JGroups used in the Platform comes with two preconfigured setups - the first one using TCP and the second one using UDP as the transport protocol.

JGroups is much faster than using 'pure' UDP or TCP: in our local testing JGroups UDP cluster, sending 10 mln messages took 2 minutes as compared with 3 minutes with UDP Multicast. Cloud testing yielded similar speed improvement.

## JGroups-based Cluster Implementation

Add the following properties to your `local.properties` file:

### i Note

TCP properties included in this example are only valid for TCP configuration. They're ignored for UDP configuration.

#### `local.properties`

```
cluster.broadcast.method=jgroups=de.hybris.platform.cluster.jgroups.JGroupsBroadcastMethod
cluster.broadcast.method.jgroups.tcp.bind_addr=12.34.56.78
cluster.broadcast.method.jgroups.tcp.bind_port=7800
cluster.broadcast.method.jgroups.channel.name=hybris-broadcast
cluster.broadcast.method.jgroups.configuration=jgroups-udp.xml
```

`cluster.broadcast.method.jgroups.configuration` property specifies the path to an xml file with the configuration settings. The default configuration for JGroups is UDP.

If you want to use the **TCP** configuration instead, change the `jgroups.configuration` property to:

#### `jgroups-tcp.xml`

If you want to use a custom configuration, specify your `custom.xml` file for UDP, and `custom-jgroups-tcp.xml` under your `extension/resources/jgroups/`, add in `local.properties`, and restart the server:

#### `local.properties`

```
cluster.broadcast.method.jgroups.configuration=custom.xml or custom-jgroups-tcp.xml
```

## JGroups UDP

JGroups UDP clustering uses IP multicast for sending messages to all members of a group and UDP datagrams for unicast messages (sent to a single member). When started, it opens a unicast and multicast socket: the unicast socket is used to send/receive unicast messages, whereas the multicast socket sends/receives multicast messages. The channel's address becomes the address and port number of the unicast socket.

A protocol stack with UDP as transport protocol is typically used with groups whose members run on the same host or are distributed across a LAN. Before running such a stack, a programmer has to ensure that IP multicast is enabled across subnets. It's often the case that IP multicast isn't enabled across subnets. In such cases, the stack has to either use UDP without IP multicasting or other transports such as TCP.

## JGroups Configuration File

### JGroups UDP Configuration File

The following is an example of configuration in the `jgroups_udp.xml` file:

```
<!--
  Default stack using IP multicasting. It is similar to the "udp"
  stack in stacks.xml, but doesn't use streaming state transfer and flushing
-->

<config xmlns="urn:org:jgroups"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="urn:org:jgroups http://www.jgroups.org/schema/JGroups-4.0.xsd">
  <UDP>
```

```

mcast_port="${hybris.jgroups.mcast_port}"
tos="8"
ucast_recv_buf_size="20M"
ucast_send_buf_size="640K"
mcast_recv_buf_size="25M"
mcast_send_buf_size="640K"
max_bundle_size="64K"
ip_ttl="${jgroups.udp.ip_ttl:8}"
enable_diagnostics="true"
thread_naming_pattern="cl"
thread_pool.enabled="true"
thread_pool.min_threads="2"
thread_pool.max_threads="8"
thread_pool.keep_alive_time="5000"/>
[...]
</config>
```

For full information on JGroup protocols, see [Protocols Supported by JGroups](#).

### JGroups TCP Configuration File

Specifying TCP in your protocol stack tells JGroups to use TCP to send messages between group members. Instead of using a multicast bus, the group members create a mesh of TCP connections.

The following is an example of configuration in the `jgroups_tcp.xml` file:

```

<!-- TCP based stack, with flow control and message bundling. This is usually used when IP multicast
     e.g. because it is disabled (routers discard multicast). -->
<config xmlns="urn:org:jgroups"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="urn:org:jgroups http://www.jgroups.org/schema/JGroups-4.0.xsd">
    <TCP
        recv_buf_size="${tcp.recv_buf_size:20M}"
        send_buf_size="${tcp.send_buf_size:640K}"
        max_bundle_size="64K"
        sock_conn_timeout="300"
        thread_pool.enabled="true"
        thread_pool.min_threads="1"
        thread_pool.max_threads="10"
        thread_pool.keep_alive_time="5000"
        bind_addr="${hybris.jgroups.bind_addr}"
        bind_port="${hybris.jgroups.bind_port}" />
        <JDBC_PING connection_driver="${hybris.database.driver}"
        connection_password="${hybris.database.password}"
        connection_username="${hybris.database.user}"
        connection_url="${hybris.database.url}"
        initialize_sql="${hybris.jgroups.schema}"
        datasource_jndi_name="${hybris.datasource.jndi.name}"
        remove_all_data_on_view_change="${hybris.jgroups.remove_all_data_on_view_change}"
        write_data_on_find="${hybris.jgroups.write_data_on_find}" />
        [...]
    </config>
```

## Related Information

[JGroups Website](#) ↗

## JGroups Kube\_Ping Configuration

KUBE\_PING is a discovery protocol for JGroups cluster nodes managed by Kubernetes. See how to configure it.

SAP Commerce can discover cluster nodes using KUBE\_PING. To enable such JGroups configuration set this property:

```
cluster.broadcast.method.jgroups.configuration=jgroups-tcp-kubeping.xml
```

Now, SAP Commerce can discover its cluster nodes by querying the Kubernetes API directly.

The JGroups configuration file is a template located in `/bin/platform/ext/core/resources/jgroups/jgroups-tcp-kubeping.xml`. It contains placeholder variables, such as `<hybris.jgroups.portRange>`, that you can set through SAP Commerce properties mechanism. Properties that match the `cluster.conf.jgroups-tcp-kubeping.[property]` pattern are used as a value for the `<hybris.jgroups.[property]>` template placeholders.

For example, to set a `<hybris.jgroups.portRange>` value in the `jgroups-tcp-kubeping.xml` template, set this SAP Commerce property:

```
cluster.conf.jgroups-tcp-kubeping.portRange=0
```

Default properties are configured in `/bin/platform/project.properties`, under the **KUBE\_PING JGroups options** section.

## Related Information

<https://github.com/jgroups-extras/jgroups-kubernetes>

## UDP Cluster Settings

See the information about both UDP multicast and UDP unicast implementations.

### UDP Multicast Implementation

The UDP-based cluster implementation uses cache invalidation signals sent to all other nodes of the SAP Commerce Cluster using multicast. If a SAP Commerce item is modified on one of the nodes, the node sends a UDP multicast datagram to all other nodes informing them that the cached information on the item is invalid.

Figure: Connection Schema of a UDP-based SAP Commerce Cluster

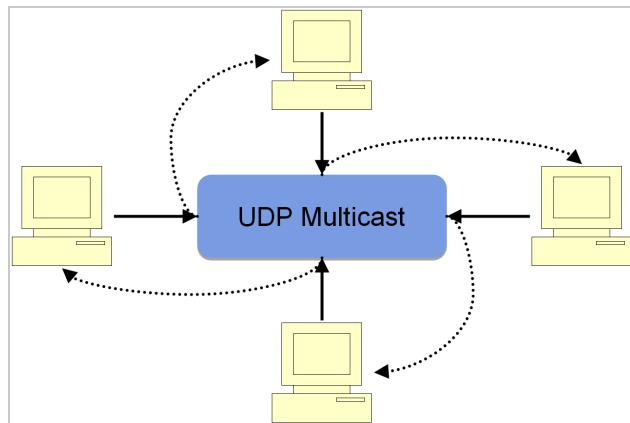
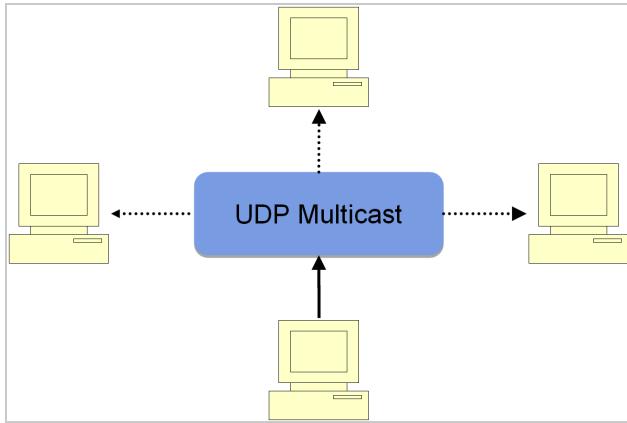


Figure: Communication Schema of a UDP-based SAP Commerce Cluster



## Configuring a UDP Multicast Based Cluster

When UDP is chosen as the communication protocol, the following settings have to be configured:

- Multicast address, on which the node listens for cluster messages:

### **local.properties**

```
cluster.broadcast.method.udp.multicastaddress=230.0.0.1 #this ip should be the same for all n
```

- Port on which the node listens for cluster messages:

### **local.properties**

```
cluster.broadcast.method.udp.port=9997 #the same for all nodes
```

- If you have many enabled network interfaces, specify the address of the interface used for communication:

### **local.properties**

```
cluster.broadcast.method.udp.networkinterface=<logical_name_of_your_network_interface> #use th
```

### i Note

If you're getting error **bad argument for IP\_MULTICAST\_IF2**, replace the address for the property **cluster.broadcast.method.udp.networkinterface** with a logical interface name, for example **eth4**. To get names for your interfaces, use this code:

```
import java.io.*;
import java.net.*;
import java.util.*;
import static java.lang.System.out;

public class ListNIFs
{
    public static void main(String args[]) throws SocketException {
        Enumeration<NetworkInterface> nets = NetworkInterface.getNetworkInterfaces();

        for (NetworkInterface netIf : Collections.list(nets)) {
            out.printf("Display name: %s\n", netIf.getDisplayName());
            out.printf("Name: %s\n", netIf.getName());
            displaySubInterfaces(netIf);
            out.printf("\n");
        }
    }

    static void displaySubInterfaces(NetworkInterface netIf) throws SocketException {
        Enumeration<NetworkInterface> subIfs = netIf.getSubInterfaces();

        for (NetworkInterface subIf : Collections.list(subIfs)) {
            out.printf("\tSub Interface Display name: %s\n", subIf.getDisplayName());
        }
    }
}
```

11/8/24, 3:01 PM

```
        out.printf("\tSub Interface Name: %s\n", subIf.getName());  
    }  
}
```

## i Note

Sometimes multicast traffic chooses to use IPV6 network interfaces even if we specifically bind by an IPV4 address. To make sure the JVM uses ipv4 add the following settings to your tomcat general options in `local.properties`:

tomcat.generaloptions=-Djava.net.preferIPv4Stack=true -Djava.net.preferIPv4Addresses=true

## Testing the Cluster

Testing is done with UDP Multicast broadcast packets. To test the configurations, use the

**ant udpsniff**

defined in the `build.xml` file in the  `${HYBRIS_BIN_DIR} /resources/ant` directory . Remember to use this specific category for running

**ant udpsniff**

**ant udpsniff**

listens to the configured port for invalidations and prints them to the console.

Run the following command on one of the other cluster nodes:

```
c:\data\hybris\bin\platform\resources\ant> ant udpsniff
...
[java] INFO  [DefaultBroadcastService] updating cluster island ID -1->15488775967864160
[java] INFO  [UDPSniffer] UDP Multicast Receiver (udpsniff) configured with the following parameters
[java] INFO  [UDPSniffer] Interface: /0.0.0.0, NetworkInterface: null
[java] INFO  [UDPSniffer] Receiving packets from group /230.0.0.1
[java] INFO  [UDPSniffer] /192.168.146.107:[?://?|ver:4010100|15488775967864160-7665975202688-58
[java] INFO  [UDPSniffer] /192.168.146.107:[?://?|ver:4010100|15488775967864160-7665975202688-59
[java] INFO  [UDPSniffer] /192.168.146.107:[?://?|ver:4010100|15488775967864160-7665975202688-60
[java] INFO  [UDPSniffer] /192.168.146.107:[?://?|ver:4010100|15488775967864160-7665975202688-61
[java] INFO  [UDPSniffer] /192.168.146.107:[?://?|ver:4010100|15488775967864160-7665975202688-62
[java] INFO  [UDPSniffer] /192.168.146.107:[?://?|ver:4010100|15488775967864160-7665975202688-63
[java] INFO  [UDPSniffer] /192.168.146.107:[?://?|ver:4010100|15488775967864160-25258161494096-4
[java] INFO  [UDPSniffer] /192.168.146.107:[?://?|ver:4010100|15488775967864160-25258161494096-4
```

If you receive these messages, the UDP multicast connection between these nodes is correctly configured. The UDP Sniffer uses the master tenant data, therefore only UDP messages from own cluster will display.

Running the SAP Commerce Server

If you have successfully tested the clustering, you can start the application server on each cluster node.

You will receive output that looks like this:

```
D:\data\platform>hybrisserver.bat  
[...]  
INFO [Http11Protocol] Initializing Coyote HTTP/1.1 on http-9001
```

11/8/24, 3:01 PM

```
INFO [Http11Protocol] Initializing Coyote HTTP/1.1 on http-9002
[...]
INFO [hybrisserver] ****
INFO [hybrisserver]
INFO [hybrisserver] Starting up hybris Server 4.0.1.0...
INFO [hybrisserver]
INFO [hybrisserver] Configuration:
INFO [hybrisserver]
INFO [hybrisserver] Cluster: 0 (master)
[...]
INFO [Catalina] Server startup in 12265 ms
```

You can see that the **cluster.id** is set to 0.

## Troubleshooting

- Double check that you have the same configuration for **cluster.port** and **cluster.multicastaddress** on both machines.
- If no packets are received, for testing purposes start the UDP sniffer on the same machine on which the pings are generated. Use another command prompt window to do this.
- If you do not get any output from using

### **ant udpsniff**

, try to run this command with root privileges (on linux machines), change the log level of the rootlogger to debug. Bear in mind that this will result in massive amounts of log output, so use it only in a testing environment and only for testing purposes.

- If you receive packets on the local machine but not on the remote one, it is possible that they are sent out using wrong network interfaces. You can change the network interface by configuring **cluster.interface**.
- Check the default time-to-live (TTL) value of the packets

The Platform does not specify a TTL and so uses the default value of the operating system. If this is set to 0, UDP packages are only processed on the local machine and are not sent to the network. TTL = 1 means that UDP packages are forwarded only in the LAN.

- Make sure that your network interface card fully supports multicasts, both sending and receiving
- Make sure that there is no IPv4/IPv6 issue. Try deactivating IPv6 or pinning the tomcat process to use a specific IP stack version

### **local.properties**

```
tomcat.javaoptions=-Djava.net.preferIPv4Stack=true -Djava.net.preferIPv6Addresses=false
```

- If you have set up the cluster correctly and started each instance but the cache invalidation still won't work, you can enable logging of the UDP packets. Add the following line to your **local.properties** file:

### **local.properties**

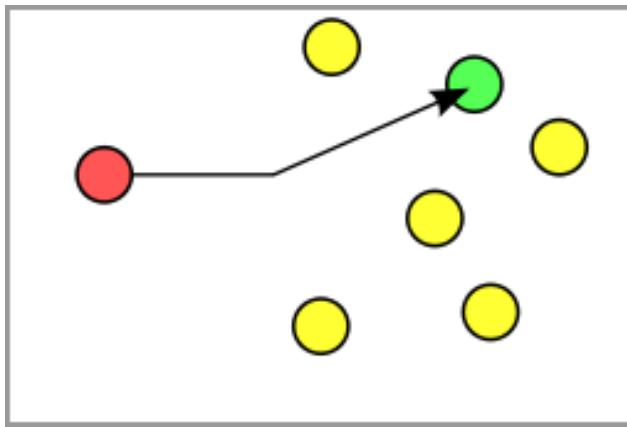
```
log4j.logger.de.hybris.platform.cache.udp=debug
```

and restart the SAP Commerce Server.

## UDP Unicast - Based Cluster Implementation

As an alternative to JGroups, it is also possible to use UDP Unicast only to send invalidation messages.

Figure: Unicast Traffic



## Configuring UDP Unicast

Add the following configuration to the **local.properties** file:

### local.properties

```
# define server host name or ip ( use a public ip here)
cluster.broadcast.method.unicast.serveraddress=myHostName

# define the server port
cluster.broadcast.method.unicast.port=myPort

# define all known nodes as hostnameOrIp:port
cluster.broadcast.method.unicast.clusternodes=node1HostName:node1Port ; node2HostName:node2Port ; nod
# if the interval value is higher than 0 , the platform synchronizes its list of known nodes with all
cluster.broadcast.method.unicast.sync.nodes.interval=-1
```

### i Note

The ping message handler now starts immediately on master tenant startup and it is no longer necessary for the user to go to SAP Commerce Administration Console for the service to start silently.

If more detailed logging is necessary, add the following configuration to the **local.properties** file.

### local.properties

```
log4j.logger.de.hybris.platform.cluster.PingBroadcastHandler=DEBUG
log4j.logger.de.hybris.platform.cluster.udp.UnicastBroadcastMethod=DEBUG
```

## Protocols Supported by JGroups

This document describes protocols of JGroup API.

## Transport Layer Protocols

See the configuration options for TCP and UDP protocols.

### TCP Configuration Options

Option	Description
bind_addr	The bind address which should be used by this transport.
bind_port	The port to which the transport binds. Default of 0 binds to any (ephemeral) port.
max_bundle_size	Maximum number of bytes for messages to be queued until they are sent.
recv_buf_size	Receiver buffer size in bytes.
send_buf_size	Send buffer size in bytes.
sock_conn_timeout	Max time allowed for a socket creation in ConnectionTable.
thread_pool.keep_alive_time	Timeout in milliseconds to remove idle thread from regular pool.
thread_pool.max_threads	Maximum thread pool size for the regular thread pool.
thread_pool.min_threads	Minimum thread pool size for the regular thread pool.
thread_pool_enabled	Switch for enabling thread pool for regular messages. Default true.

## UDP Configuration Options

Option	Description
mcast_port	The multicast port used for sending and receiving packets. Default is 7600.
tos	Traffic class for sending unicast and multicast datagrams. Default is 8.
ucast_recv_buf_size	Receive buffer size of the unicast datagram socket. Default is 64'000 bytes.
ucast_send_buf_size	Send buffer size of the unicast datagram socket. Default is 100'000 bytes.
mcast_recv_buf_size	Receive buffer size of the multicast datagram socket. Default is 500'000 bytes.
mcast_send_buf_size	Send buffer size of the multicast datagram socket. Default is 100'000 bytes.
max_bundle_size	Maximum number of bytes for messages to be queued until they are sent.
ip_ttl	The time-to-live (TTL) for multicast datagram packets. Default is 8.
enable_diagnostics	Switch to enable diagnostic probing. Default is true.
thread_naming_pattern	Thread naming pattern for threads in this channel. Default is cl.
thread_pool.keep_alive_time	Timeout in milliseconds to remove idle thread from regular pool.
thread_pool.max_threads	Maximum thread pool size for the regular thread pool.
thread_pool.min_threads	
thread_pool_enabled	Switch for enabling thread pool for regular messages. Default is true. Minimum thread pool size for the regular thread pool.

# Discovery Protocols

See the configuration options for discovery protocols.

## TCPPING

The TCPPING protocol layer retrieves the initial membership in answer to the GMS's FIND\_INITIAL\_MBRS event. The initial membership is retrieved by directly contacting other group members, sending Messages containing point-to-point membership requests. The responses should allow us to determine the coordinator whom we have to contact in case we want to join the group. When we are a server (after having received the BECOME\_SERVER event), we'll respond to TCPPING requests with a TCPPING response. The FIND\_INITIAL\_MBRS event will eventually be answered with a FIND\_INITIAL\_MBRS\_OK event up the stack.

Option	Description
initial_hosts	Comma delimited list of hosts to be contacted for initial membership.
port_range	Number of ports to be probed for initial membership. Default is 1.

## PING

Initial (dirty) discovery of members. Used to detect the coordinator (oldest member), either by multicasting PING requests to an IP MCAST address or connecting to a GossipRouter.

Each member responds with a packet {C, A}, where C=coordinator's address and A=own address. After milliseconds or replies, the joiner determines the coordinator from the responses, and sends a JOIN request to it (handled by GMS). If nobody responds, we assume we are the first member of a group.

Unlike TCPPING, PING employs dynamic discovery, meaning that the member does not have to know in advance where other group members are.

## JDBC\_PING

JDBC\_PING uses a database to store information about cluster nodes used for discovery. All cluster nodes are to be able to access the same database. When a node starts, it queries information about existing members from the database, determines the coordinator, and then asks the coordinator to join the cluster. It also inserts information about itself into the table, so that other nodes can subsequently find it.

## KUBE\_PING

For information about configuring the KUBE\_PING protocol, see [JGroups Kube\\_Ping Configuration](#).

# Merge Protocol Merge3

See the configuration options for the Merge Protocol Merge3 protocol.

If a group gets split for some reasons (e.g. network partition), this protocol merges the subgroups back into one group. It is only run by the coordinator (the oldest member in a cluster), and periodically multicasts its presence. If another coordinator (for the same group) receives this message, it will initiate a merge process. Note that this merges subgroups {A,B} and {C,D,E} back into {A,B,C,D,E}, but it does not merge state. The app has to handle the callback to merge state.

Option	Description
max_interval	Maximum time in ms between runs to discover other clusters.

Option	Description
min_interval	Minimum time in ms between runs to discover other clusters.

## Failure Detection Protocols

See the configuration options for FD\_SOCK, FD, FD\_ALL, and VERIFY\_SUSPECT protocols.

### FD\_SOCK Protocol

Failure detection protocol based on a ring of TCP sockets created between group members. Each member in a group connects to its neighbor (last member connects to first) thus forming a ring. Member B is suspected when its neighbor A detects abnormally closed TCP socket (presumably due to a node B crash). However, if a member B is about to leave gracefully, it lets its neighbor A know, so that it does not become suspected.

One FD\_SOCK disadvantage is that hung servers and/or crashed switches will not cause sockets to be closed. Therefore hung members will not be suspected and network partitions due to switch failures will not be detected. A solution to this problem is to use both FD and FD\_SOCK failure detection protocols. For more details refer to Failure Detection

FD\_SOCK uses JGroups defaults

### FD Protocol

Failure detection based on heartbeat messages. A member sends 'are-you-alive' messages with a periodicity of 'timeout' milliseconds. After the first missing heartbeat response, the initiating member send more 'max\_tries' heartbeat messages and the target member is declared suspect only after all heartbeat messages go unanswered.

In the worst case, when the target member dies immediately after answering a heartbeat, the failure takes  $\text{timeout} + \text{timeout} + \text{max\_tries} * \text{timeout} = (\text{max\_tries} + 2) * \text{timeout}$  milliseconds to detect.

Once a member is declared suspected it will be excluded by GMS. SUSPECT event handling is also subject to interaction with VERIFY\_SUSPECT. If we use FD\_SOCK instead, then we don't send heartbeats, but establish TCP sockets and declare a member dead only when a socket is closed.

Option	Description
max_tries	Number of times to send an are-you-alive message.
timeout	Timeout to suspect a node P if neither a heartbeat nor data were received from P. Default is 3000 msec.

### FD\_ALL Protocol

Failure detection based on simple heartbeat protocol. Every member periodically multicasts a heartbeat. Every member also maintains a table of all members (minus itself). When data or a heartbeat from P are received, we reset the timestamp for P to the current time. Periodically, we check for expired members, and suspect those.

FD\_ALL uses JGroups defaults

### VERIFY\_SUSPECT Protocol

Verifies that a suspected member is really dead by pinging that member once again. Drops suspect message if member does respond. Tries to minimize false suspicions.

The protocol works as follows: it catches SUSPECT events traveling up the stack. Verifies that the suspected member is really dead. If yes, passes SUSPECT event up the stack, otherwise discards it. Has to be placed somewhere above the FD layer and below the GMS layer (receiver of the SUSPECT event). Note that SUSPECT events may be reordered by this protocol.

Option	Description
timeout	Number of millisecs to wait for a response from a suspected member.

## Synchronization Protocol BARRIER

See the configuration options for the BARRIER protocol.

The BARRIER protocol can be used to suspend the delivery of messages up the protocol stack. It can be seen as a distributed counterpart to the barrier notion found in thread programming. See the configuration options for Protocol BARRIER. All messages up the stack have to go through a barrier. By default, the barrier is open.

When a CLOSE\_BARRIER event is received, we close the barrier. This succeeds when all previous messages have completed. Thus, when the barrier is closed, we know that there are no pending messages processed.

When an OPEN\_BARRIER event is received, we simply open the barrier again and let all messages pass in the up direction.

Option	Description
max_close_time	How long can a barrier stay closed (0 means forever).

## Reliable Message Transmission Protocols

See the configuration options of the pbcast.NAKACK, pbcast.STABLE, and UNICAST protocols.

### **pbcast.NAKACK**

Lossless and FIFO delivery of multicast messages, using negative acks. E.g. when receiving P:1, P:3, P:4, a receiver asks P for retransmission of message 2.

Option	Description
use_mcast_xmit	Retransmit messages using multicast rather than unicast.
discard_delivered_msgs	Should messages delivered to application be discarded.

### **pbcast.STABLE**

Garbage collects messages that have been seen by all members of a cluster. Each member has to store all messages because it may be asked to retransmit. Only when we are sure that all members have seen a message can it be removed from the retransmission buffers. STABLE periodically gossips its highest and lowest messages seen. The lowest value is used to compute the min (all lowest seqnos for all members), and messages with a seqno below that min can safely be discarded.

Note that STABLE can also be configured to run when N bytes have been received (size-based gossiping). This is recommended when sending messages at a high rate, because time based gossiping might accumulate messages faster than STABLE can garbage collect them.

Option	Description
desired_avg_gossip	Average time to send a STABLE message. Default is 20000 msec.
stability_delay	Delay before stability message is sent. Default is 6000 msec.
max_bytes	Maximum number of bytes received in all messages before sending a STABLE message is triggered. Default is 0 (disabled)

## UNICAST

Lossless and FIFO delivery of unicast messages.

UNICAST for TCP protocol uses JGroups defaults.

UNICAST for UDP protocol uses following options:

Option	Description
xmit_table_max_compaction_time	Number of milliseconds after which the matrix in the retransmission table is compacted (only for experts).
xmit_table_msgs_per_row	Number of elements of a row of the matrix in the retransmission table (only for experts). The capacity of the matrix is xmit_table_num_rows * xmit_table_msgs_per_row.
xmit_table_num_rows	Number of rows of the matrix in the retransmission table (only for experts).

## Group Membership Protocol pbcast.GMS

See the configuration options for the pbcast.GMS protocol.

Group Membership Service. Responsible for joining/leaving members. Also handles suspected members, and excludes them from the membership. Sends Views (topology configuration) to all members when a membership change has occurred.

Option	Description
print_local_addr	Print local address of this member after connect. Default is true.
join_timeout	Join timeout. Default is 5000 msec.
view_bundling	View bundling toggle.

## Fragmentation Protocol FRAG2

See the configuration options for the FRAG2 protocol.

Fragments messages larger than 'frag\_size' bytes. Unfragments at the receiver's side. Works for both unicast and multicast messages.

Compared to FRAG, this protocol does not need to serialize the message in order to break it into smaller fragments: it looks only at the message's buffer, which is a byte array anyway. We assume that the size addition for headers and src and dest address is minimal when the transport finally has to serialize the message, so we add a constant (by default 200 bytes). Because of the efficiency gained by not having to serialize the message just to determine its size, FRAG2 is generally recommended over FRAG.

Option	Description
frag_size	The max number of bytes in a message. Larger messages will be fragmented. Default is 8192 bytes.

## State Transfer Protocols

See the configuration options of the pbcast.STATE\_TRANSFER, and RSVP protocols.

### pbcast.STATE\_TRANSFER

Allows a joining member to retrieve a shared group state from the oldest member (coordinator). Other members do not have to stop sending messages, while state transfer is in progress.

pbcast.STATE\_TRANSFER uses JGroups defaults

### RSVP Protocol

Protocol which implements synchronous messages. A send of a message M with flag RSVP set will block until all non-faulty recipients (one for unicasts, N for multicasts) have acked M, or until a timeout kicks in.

## Related Information

[Cluster - Technical Guide](#)

## Node-specific Configurations

Using node-specific cluster settings allows specifying a custom configuration for each individual node, for example varying cache size.

Within an SAP Commerce cluster, it may be necessary to set up individual configuration parameters for each node. It may happen because nodes may have, for example, different memory capacity or varying CPU performance.

### i Note

If you want to use this feature, you still need a `local.properties` file on each cluster node with a proper configuration described in [Configuring a SAP Commerce Cluster](#).

A cluster node-specific property setting is prefixed by `cluster.cluster_number`, such as:

```
cluster.12.myproperty=myvalue
cluster.13.myproperty=myvalue2
```

A node-specific property applies to a cluster node if `cluster_number` matches the cluster node number. The code snippet above defines different values of the `myproperty` property for the nodes 12 and 13. By consequence, `myproperty` has a different value for both nodes, although it is defined under the same name.

During a SAP Commerce build, only the build framework-related properties are effective. Cluster-node specific property values do not apply. This means that you cannot specify node-specific build framework properties, such as a fictitious `cluster.1.build.compiler=classic`. The property would use during the SAP Commerce build the `classic` compiler setting for node 1 only.

The side effect is that you cannot configure the Apache Tomcat start-up settings via node-specific property values. The build framework of SAP Commerce writes the start-up values for the Apache Tomcat as part of the **deploy** build target. The build framework cannot write node-specific startup settings for the Apache Tomcat, because it does not evaluate node-specific property values: the node-specific property values would have to be available at compile time, but are not available until runtime. If you need to specify node-specific startup parameters for the Apache Tomcat, you should use the **-D** command line switch or edit the Apache Tomcat configuration files manually.

Cluster node 15 is a special case. It is used for SAP Commerce when started in stand-alone operation mode, for example when launched from Eclipse and during JUnit tests. When launched in stand-alone mode, SAP Commerce ignores the value of the **cluster.clusternode** property.

If no specific value for **cluster.id** is defined in **local.properties** file, the default value for the **cluster.id** is set to **15**. Cluster node 15 also represents an SAP Commerce installation in stand-alone mode of operation. In other words, all property values specific for the cluster node 15 also apply to:

- SAP Commerce launched in stand-alone mode
- Every instance of SAP Commerce with no value set for the **cluster.id** property

Region Cache is the default cache. You can configure the following properties in your **local.properties** file:

Property Name	Default Property Value	Description
<b>cluster.15.regioncache.entityregion.size</b>	<b>100000</b>	The size of a region that stores all other, non-typesystem and non-query objects.
<b>cluster.15.regioncache.querycacheregion.size</b>	<b>20000</b>	The size of a region that stores query results.

## Customizing Configuration of Read Directories

To configure read directories separately for each node, use the following code:

```
media.read.dir=C:\\images          #the default for all clusternodes
cluster.1.media.read.dir=C:\\data   #if running on node 1, use this
cluster.2.media.read.dir=D:\\images\\data #and this for node 2
```

### i Note

It is good practice to use identical physical machines with the same OS and directory structure to make system maintenance easier.

## Customizing Configuration of Media Replication Directories

Platform has a property **media.replication.dirs**, which accepts a number of directories to which media instances will be written. It can be set at **local.properties** or **project.properties** file.

The following code snippet shows a sample setting of media replication directories. Remember to use double backslashes when specifying the paths.

**local.properties**

```
...
media.replication.dirs=C:\\media;N:\\remotedir\\media
```

In this example, C:\media is a directory at a local hard disk and N:\remotedir\media represents a directory on a remote server. Whenever a media is created, it is written to any path given here.

In the same way as with the **media.read.dir**, it is possible to configure media replication directories specific to each node in a SAP Commerce cluster.

You can modify the properties for cluster nodes according to the following regular expression:

#### HybrisConfig.java

```
^(standalone\\.)?(cluster\\.(\\d+)\\.)?(.*$
```

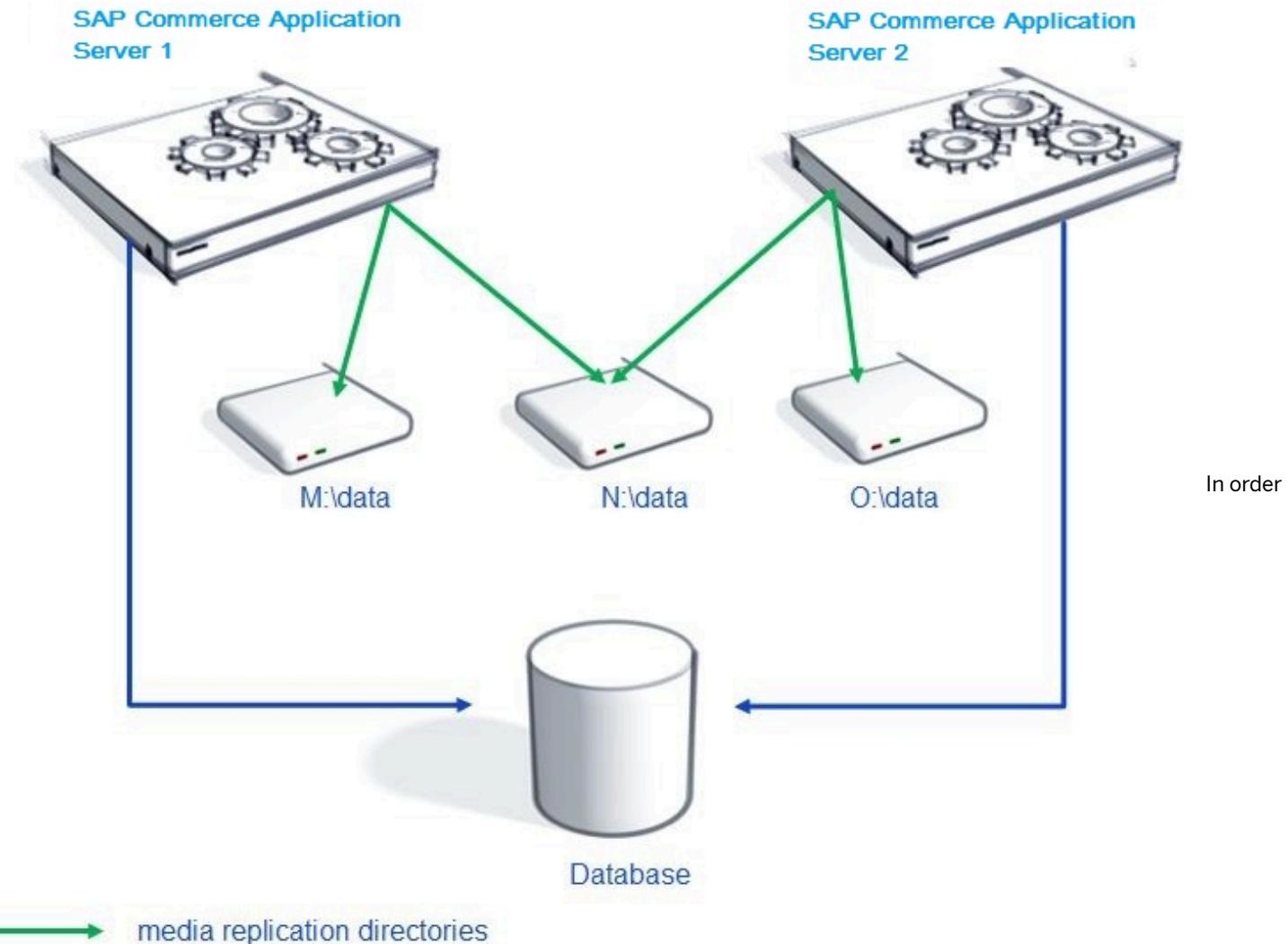
Hence, it is possible to have this property in the **local.properties** file for cluster node 1:

```
cluster.1.media.replication.dirs
```

#### i Note

It is not enough to have the whole configuration in one file. Settings for cluster node 1 need to be configured in **local.properties** file of node 1 and settings for cluster node 2 need to be configured in its own **local.properties** file.

The following figure illustrates two SAP Commerce nodes with different media replication directories. SAP Commerce application server 1 writes the media files to **M:\data** and **N:\data** directories and SAP Commerce application server 2 writes the media files to **N:\data** and **O:\data** directories.



to configure cluster nodes for media replication as in the figure above, you need to have the following settings in the `local.properties` files.

For SAP Commerce node 1, edit its `local.properties` at  `${HYBRIS_DIR} /config`.

`local.properties`

For SAP Commerce node 2, edit its `local.properties` at  `${HYBRIS_DIR} /config`

`local.properties`

## Setting the Node ID for an Existing Cron Job

When creating a cron job, you need to set the ID of the node on which the cron job is expected to run. If you want to change the assigned node id, you can do so by updating the cronjob through Backoffice. For more information, refer to [items.xml](#).

The following code snippet shows how to programmatically move a cron job to another cluster node after the cron job has been created:

```
import de.hybris.platform.servicelayer.cronjob.CronJobService;
import de.hybris.platform.servicelayer.model.ModelService;
import de.hybris.platform.cronjob.model.JobModel;
import de.hybris.platform.cronjob.model.CronJobModel;

//cronJobService, modelService should be injected here

JobModel jobModel = cronJobService.getJob("helloWorldJobPerformable");
print(cronJobService);
print(jobModel.getJobCronJobs());
for(CronJobModel cronJob : jobModel.getJobCronJobs())
{
    print(cronJob.getCode());
    cronJob.setNodeID(Integer.valueOf(10));
    modelService.save(cronJob);
}
```

## Rolling Update on the Cluster

With the rolling update feature, you can update an SAP Commerce cluster without shutting it down.

Performing a system update can potentially mean shutting down your installation for the duration of the update. During this time, your platform is unavailable to end users. However, if you have installed SAP Commerce in a cluster, the rolling update feature only requires one node within the cluster to be shut down at a time. Each node has to be restarted, therefore reliable session replication and request routing is essential.

## Cloning the Type System

During a rolling update it is required for a while that multiple nodes work with different codebases and different type system definitions. To allow old nodes to work correctly we need to perform system update on a separate type system.

The type system used is determined by the `db.type.system.name` property. By default, it is set to `DEFAULT`, which means the **default type system** is used.

To copy the type system you currently use, execute:

```
ant createtypesystem -DtypeSystemName=new_type_system
```

When the codebase on the old node is updated, it becomes a new node. This new node must be started with the new type system.

### i Note

The `createtypesystem` target copies only the type system - related data. Other data such as products or customers are not copied.

## Deleting the Outdated Type System

To delete your outdated type system, in the `<HYBRIS_BIN_DIR>/platform` directory, execute:

```
ant droptypesystem -DtypeSystemName=USER_DEFINED_TYPE_SYSTEM
```

Before you delete your outdated type system, make sure everything works properly. Backup your data for safety.

## Things to Look Out For

### Tasks and Cronjobs

Tasks and cronjobs introduced by the new codebase work properly only on the nodes with the new codebase. These tasks and cronjobs may fail on the nodes with the old codebase. To limit potential issues and to simplify the procedure, shut down the background processing nodes.

### References to Type Models

During a rolling update, when a new type system is created, a type model may have a reference to a type model that isn't available while it is being accessed by the old type system. It may be a valid reference as this second type may be visible after you complete updating the type system.

In case a reference to such an unavailable or non-existing type model instance (for example, the `EnumeratoinValue`) is found, the type system can set the value of the reference in the holding model to `null` in the database. However, as mentioned above, this reference can still be valid as the underlying type may be a part of a pending rolling update codebase change.

To prevent such a reference from being set to `null`, use the `self.healing.suppressForTypeSystemTypes` property. By default, this property is `true`, which prevents the system from setting the value to `null`. Instead, it leaves the reference value in the database, and returns `null` during runtime as a given reference value.

The `self.healing.suppressForTypeSystemTypes.typecodes` property defines a list of typecodes for which to trigger this mechanism. This list contains the type system related types such as `EnumeratoinValue`, `AtomicTypeModel`, or `AttributeDescriptorModel`.

## Perform a Rolling Update

Follow these steps to learn how to perform a rolling update in a clustered environment.

### Context

To perform a rolling update on a cluster, follow the steps:

### Procedure

1. Execute the following command in the `<HYBRIS_BIN_DIR>/platform` directory to copy the current type system:

```
ant createtypesystem -DtypeName=new_type_system
```

2. Update the new type system:

a. Add the `db.type.system.name` property in the `local.properties` file.

```
db.type.system.name=new_type_system
```

b. Execute the following command in the `<HYBRIS_BIN_DIR>/platform` directory:

```
ant updatesystem
```

3. For each node in the cluster, do the following:

a. Shut down the node.

b. Update the codebase for the selected node.

c. Add the `db.type.system.name` property in your `local.properties` file to configure the node to use the new type system.

```
db.type.system.name=new_type_system
```

- d. Start up the node in the cluster.

## Results

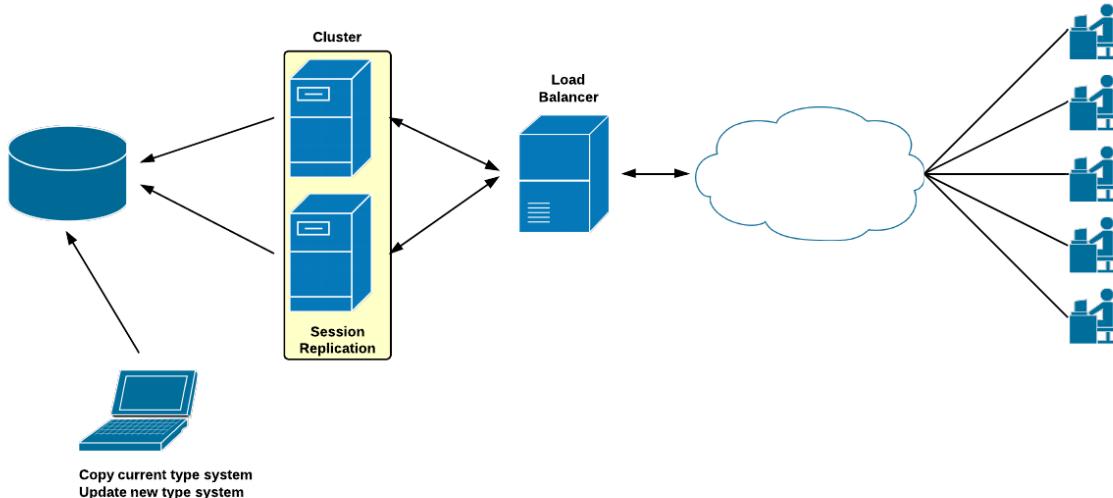
You completed a rolling update procedure.

# Perform a Rolling Update - Example with Two Nodes

Follow these steps to perform an example rolling update in a clustered environment with two nodes.

## Context

The example below shows how to perform the rolling update.



## Procedure

### Updating the First Node

1. Execute the following command to clone the current type system:

```
ant createtypesystem -DtypeSystemName=new_ts
```

The name of the new type system is defined with the `typeSystemName` parameter value. In this example, the name of the new type system is `new_ts`.

2. Change the codebase and the type system in the first node.

3. After changing the codebase, run:

```
ant clean all
```

On the first node you have a new codebase and a new type system definition in the `items.xml` files.

4. Change the value of the `db.type.system.name` property in your `local.properties` file to configure the node to use the new type system:

```
db.type.system.name=new_ts
```

5. Make sure that you can see the following state of the first node:

- the new codebase
- the new type system definition
- the node is configured to use the cloned type system.

6. Execute the following command in the <HYBRIS\_BIN\_DIR>/platform directory:

```
ant updatesystem
```

You should have the node updated.

7. Start the first node.

#### Updating the Second Node

8. Shut down the second node.

9. Change the value of the db.type.system.name property in the local.properties file to configure Platform to use the updated type system:

```
db.type.system.name=new_ts
```

10. Change the codebase and the type system at the second node.

11. Run the following command after changing the codebase:

```
ant clean all
```

On the second node you have a new codebase and a new type system definition in the items.xml files.

12. Start the second node.

## Perform a Rolling Update - Example with Four Nodes

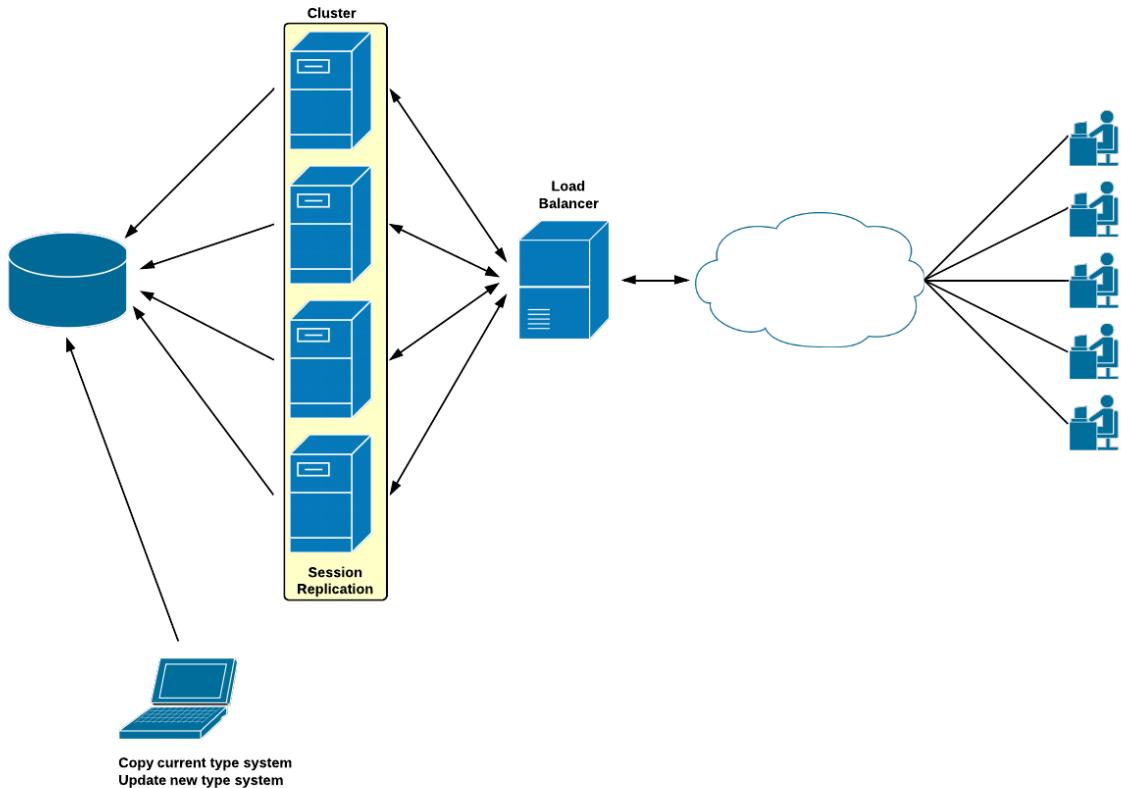
Perform an example rolling update on a cluster with four nodes.

### Context

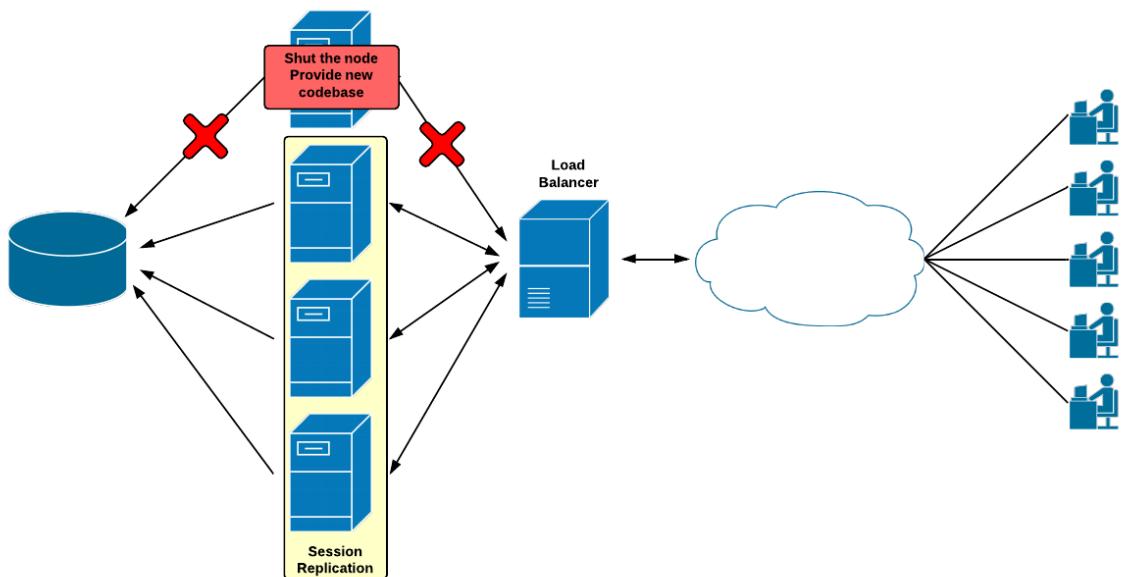
The cluster consists of four nodes working with an old codebase. Sessions are shared between the cluster nodes with a session replication mechanism.

### Procedure

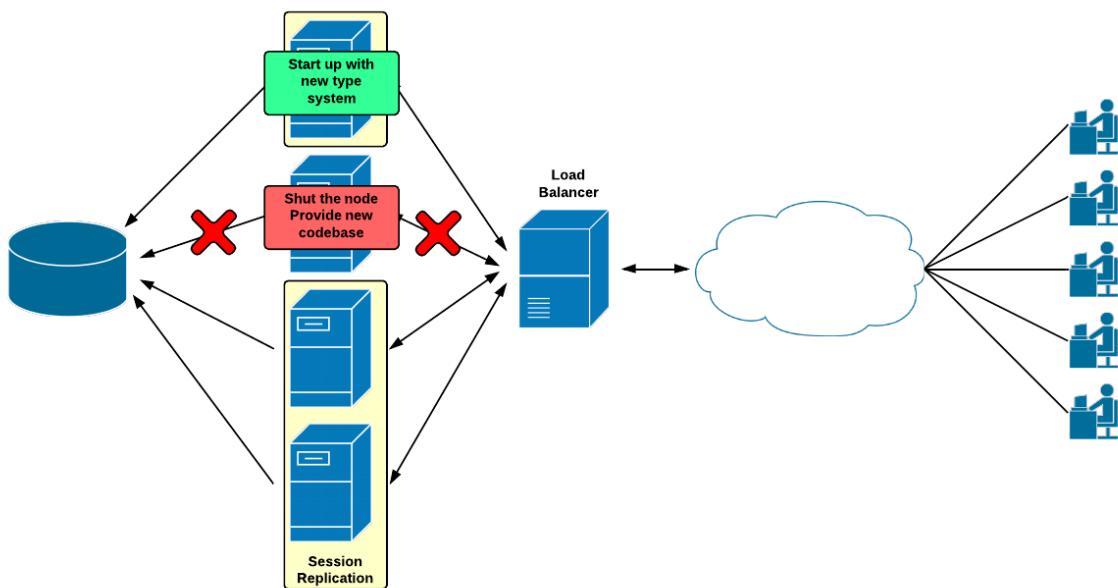
1. You need to copy the type system. To do so, execute **ant createtypesystem -DtypeSystemName=new\_type\_system\_name** command. The name of new type system is defined with the typeSystemName parameter value.



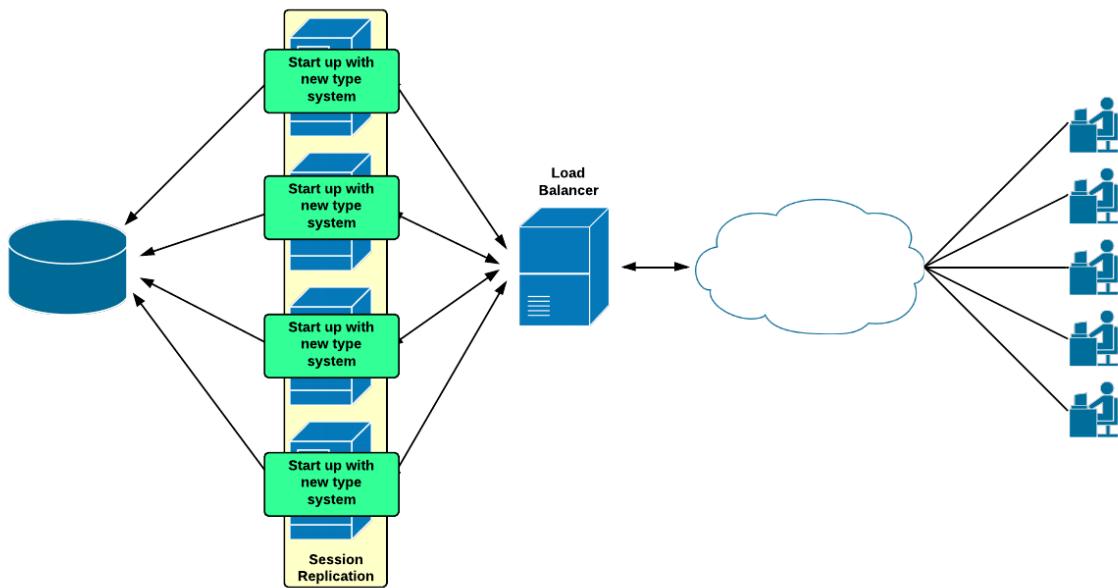
2. Shut down the first node in the cluster.
3. Change the codebase and the type system at the first node.
4. After changing the codebase run `ant clean all` command. Therefore, on the first node you have a new codebase and a new type system definition in the `items.xml` files.



5. When you start the first node with the new type system, it takes part in a session replication.



6. Now repeat steps from 2 to 5 for the rest of your all nodes.
7. All your nodes in the cluster should work with the new type system.

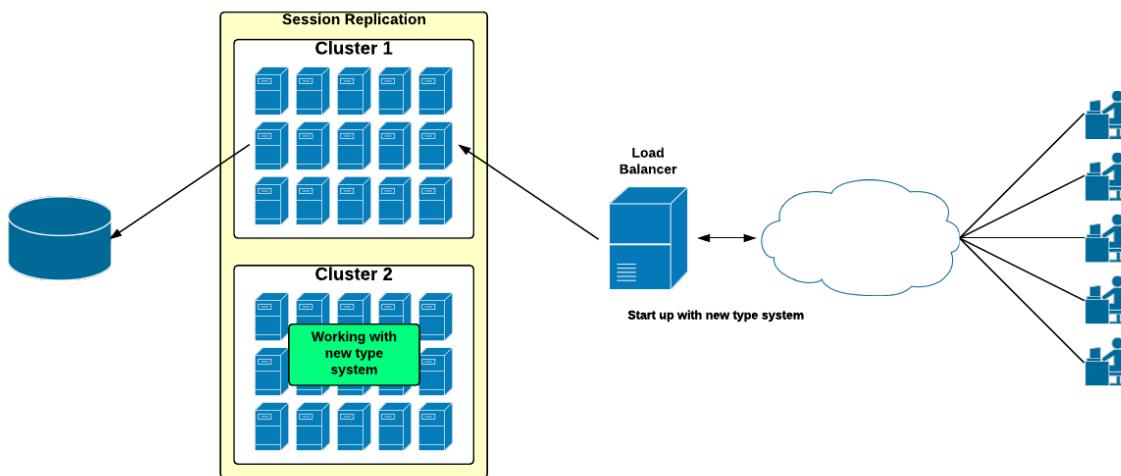


## Switching Between Clusters

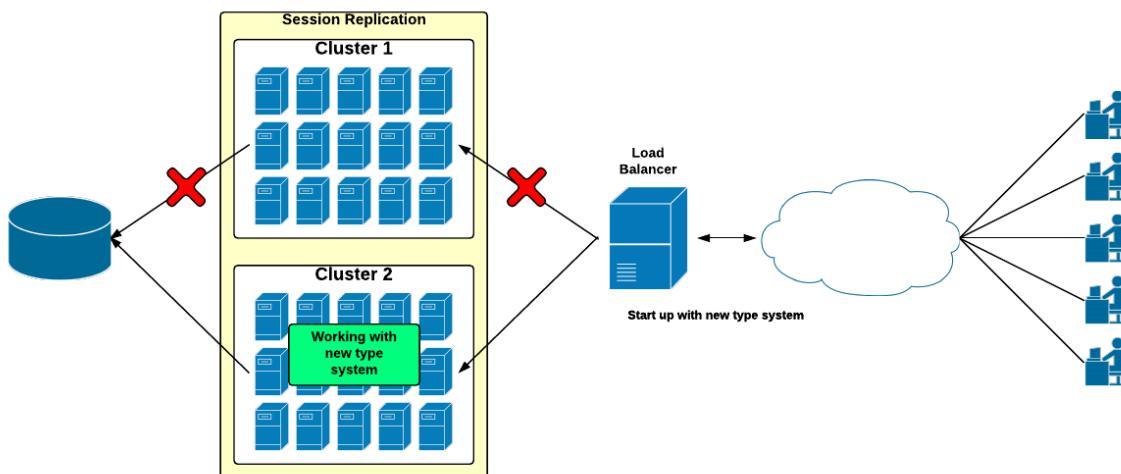
Learn how to switch from a cluster working with an old type system to a cluster with a new type system.

In this scenario the two clusters share sessions by a session replication. There is an old and a new (updated) type system.

Cluster 1 works with the old type system serving all the requests. Cluster 2 is updated with the new codebase and configured to work with the new (updated) type system.



Configure the load balancer to use Cluster 2 instead of Cluster 1. Then, cluster 2 serves all the requests.



## Related Information

[Cluster - Technical Guide](#)

## Cluster Improvements and ID Autodiscovery

Making cluster deployments - especially in cloud and virtualized environments - is easier after we've introduced some improvements in SAP Commerce clustering support.

### Cluster ID Autodiscovery

Cluster **autodiscovery** is a feature that makes deployment less complex and more flexible. All nodes that form a cluster can have an automatically assigned ID number. You don't have to explicitly set the `cluster.id` property on each node.

To enable ID autodiscovery, set `cluster.nodes.autodiscovery` to `true` (it's disabled by default). It makes each new node query a database table to get information about other nodes in the cluster and fetch the next available ID number.

Instances periodically ping each other and update node records in the database. The `cronjob.unlocker.interval.ms` property sets time interval between each heartbeat pulse. Stalled or crashed nodes are removed from a cluster and new nodes can take their IDs. You can set the time after which a node that doesn't respond is considered down, using the `cronjob.unlocker.stale.node.interval.ms` property. Both `cronjob.unlocker.interval.ms` and

`cronjob.unlocker.stale.node.interval.ms` need to be set to the same value in all nodes. The `cronjob.unlocker.interval.ms` property must be smaller than `cronjob.unlocker.stale.node.interval.ms`.

### ⚠ Caution

If there's a network loss, cluster IDs could be wrongly reassigned to new instances.

## ID Autodiscovery Properties

Property	Default value	Description
<code>cluster.nodes.autodiscovery</code>	<code>false</code>	Turns on the cluster node ID autodiscovery.
<code>cluster.nodes.unregisterOnShutdown</code>	<code>true</code>	Allows you to remove a node entry from the <code>CLNodeInfos</code> table when the node is shutting down. When set to false, the entry isn't removed and its ID can be reused after the node is treated as stale.
<code>cronjob.unlocker.interval.ms</code>	<code>300000</code>	Time between pings that update information about nodes.
<code>cronjob.unlocker.stale.node.cutoff.interval.seconds</code>	<code>1800</code>	Allows you to set the cutoff for the stale node lookup. The thread ignores the nodes that are inactive for a period longer than <code>cronjob.unlocker.stale.node.interval.ms</code> minus the value of this property. If set to 0 or less, the thread ignores any nodes and looks for nodes with the last ping since the epoch time.
<code>cronjob.unlocker.stale.node.interval.ms</code>	<code>1800000</code>	Time after which a node is considered to be stale or crashed. Must be bigger than <code>cronjob.unlocker.interval.ms</code> .

## Related Information

[Cluster - Technical Guide](#)

## Collecting and Classifying Running Operations

To suspend SAP Commerce, you need to collect all running operations, and classify them. The feature is based on Platform registering all running threads.

To enable Platform to register non-pool-related threads, use the `RegistrableThread` class as a replacement for the `java Thread` class.

To register a thread with specific operation information, call the `RegistrableThread.registerThread(...)` method available through an instance of the `OperationInfo` class. By default `RegistrableThread` is suspendable.

### ℹ Note

By default, every thread, registered or not, is treated as suspendable. To change this default behavior for a specific thread, use `OperationInfo` to provide information stating that this thread is to be non-suspendable. This is an example:

```
OperationInfo.builder() //  
    .withCategory(Category.TASK) //
```

```
asNotSuspendableOperation() . //
asJunitOperation().build()
```

When a `RegistrableThread` has done its job, unregister it via the corresponding `RegistrableThread.unregisterThread()` method.

For pool-related operations, use the `PoolableThread` class, which is derived from `RegistrableThread`. When idle in a pool, a `PoolableThread` is set to the suspendable state by the `internalRun()` method. When a `PoolableThread` has some work to do, for the time of execution its state is updated via the `PoolableThread.updateOperationInfo()` method.

The `PoolableThread.updateOperationInfo()` method either provides default `OperationInfo` for a non-suspendable thread, or - when `Runnable` also implements `ProcessWithOperationInfo` interface - it takes its `OperationInfo` from `ProcessWithOperationInfo.getOperationInfo()`.

When the system waits to be suspended, the JDBC connection pool returns connections only to the non-suspendable operations that have to finish. For other operations, `SystemIsSuspendedException` is thrown. `SystemIsSuspendedException` is also thrown in two cases while the system is already suspended, that is when you update `OperationInfo` from the suspendable state to the non-suspendable state, or when you try to register a non-suspendable thread.

## Registering Threads: Tutorial 1

Learn how to register threads you created yourself as non-suspendable using `RegistrableThread`, and provide some information about them.

### Context

By default, every thread, registered or not, is treated as suspendable. However, we recommend using `RegistrableThread` every time you create threads within SAP Commerce instead of `Thread(...)` because `RegistrableThread` enables you to easily and explicitly register your thread. As a result you gain full control over all operations. `RegistrableThread` also enables you to easily change the default behaviour and register threads as non-suspendable. It also enables you to provide specific information related to a given thread.

### Procedure

1. Create a `Runnable` to do some "important stuff" for you.

```
Runnable runnable = new Runnable...
```

2. Create a new thread.

```
RegistrableThread thread = new RegistrableThread(runnable);
```

3. Make the thread non-suspendable, and provide information about it.

```
thread.withInitialInfo(OperationInfo.builder() . //
    withCategory(Category.SYSTEM) . //
    withStatusInfo("Doing very important stuff") . //
    asNotSuspendableOperation().build());
```

4. Start the thread.

```
thread.start();
```

### Results

The internal mechanism of `RegistrableThread` takes care of registering and deregistering the thread. The thread is correctly registered as a non-suspendable operation.

## Registering Threads: Tutorial 2

Learn how to register external threads as non-suspendable using `OperationInfo`, and provide information about it.

Whenever you don't have control over a thread creation process, for example in case of http threads coming from a web server, and therefore cannot use `RegistrableThread`, register such a thread manually:

```
RegistrableThread.registerThread(OperationInfo.builder().withCategory(Category.SYSTEM).asNotSuspendable()

// do some business logic here
RegistrableThread.unregisterThread();
```

## Registering Threads: Tutorial 3

Learn how to use the `ProcessWithOperationInfo` interface to register threads as suspendable when required when you use `PoolableThread`.

### Context

In case you use a `PoolableThread`, it is by default registered as a non-suspendable operation for the time of execution of its associated `Runnable` instance. When changing its state to idle, a `PoolableThread` is always registered as suspendable so that the system can be suspended when all `PoolableThreads` are idle.

If you want to change this behavior, you could use the `ProcessWithOperationInfo` interface to set a thread as suspendable whenever required, and to provide some information. Follow the steps to learn how to do it.

### Procedure

1. Implement `Runnable` with some work to be done in the `run()` method, and `ProcessWithOperationInfo` to provide the correct information about the thread.

```
class MyRunnable implements Runnable, ProcessWithOperationInfo {
    @Override
    public OperationInfo getOperationInfo()
    {
        return OperationInfo.builder(). //
            withStatusInfo("Doing very important stuff"). //
            build();
    }

    @Override
    public void run()
    {
        // do some work here
    }
}
```

2. Instantiate your `Runnable` and execute it with `PoolableThread` from the thread pool.

```
MyRunnable runnable = new MyRunnable();
final PoolableThread thread = Registry.getCurrentTenantNoFallback()
```

```
.getThreadPool().borrowThread();
thread.execute(runnable);
```

For the time of execution, the status of the thread is changed. As `PoolableThread` is suspendable in the idle state, this will not change during execution.

## Containerized SAP Commerce

Containerization consists in building Docker images according to a specified configuration, and running those images as software instances.

Platform provides special recipes that enable you to build and run images so that you can see how a containerized SAP Commerce works in action.

Docker is one of the most popular containerization technologies. It automates SAP Commerce deployment because its containers are easily transferrable across different machines running Docker. It also guarantees reliability of application performance.

## Containerizing SAP Commerce

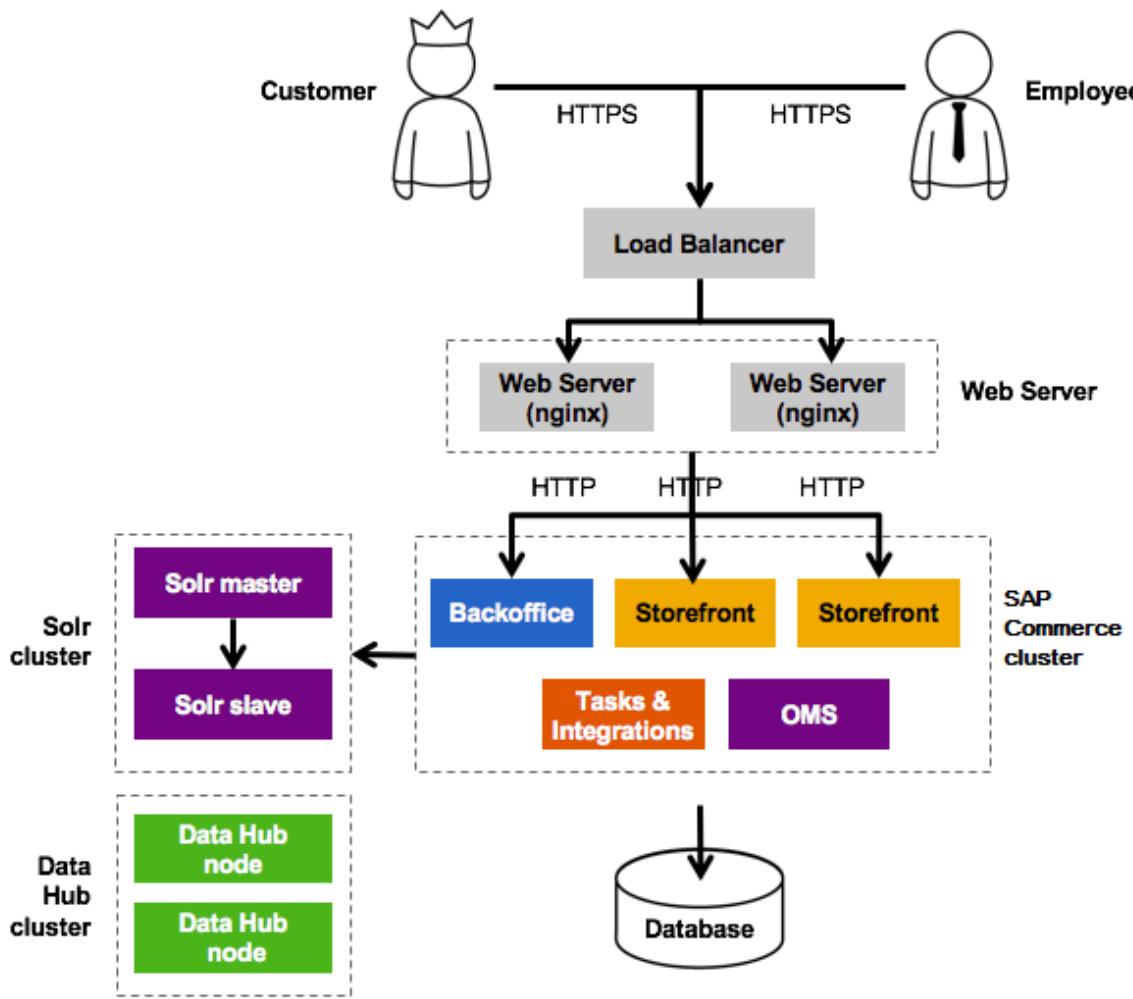
You need to prepare SAP Commerce before you can run it in a Docker container. Containers are running instances of Docker images. Images are built from special file structures mirroring the structures of the components of your SAP Commerce setup.

You can build Docker images for all the SAP Commerce components that require separate deployment, that is:

- Platform
- Data Hub
- Solr

Additionally, you can build Docker images for other mandatory parts of a SAP Commerce cluster:

- Load balancer and Web Server (Apache)
- Database (mainly for testing)



You can build all of the images within a single Installer recipe. You can define a full SAP Commerce cluster configuration in one file. Moreover, you can give all Platform type images multiple modes called aspects in which the images can be started. So it is possible to clearly configure different types of Platform servers without the need to build a separate image for each of them. For more information, see [Writing a Custom Recipe](#).

Once you have built your images, you can start them locally as well as in the production environment, without changing the image. This enables developers to perform production-like testing of such a cluster on their machines.

#### [Platform Containerization Plugin](#)

The Platform Containerization Plugin enhances Installer recipes with a domain-specific language (a DSL) enabling you to describe a SAP Commerce deployment structure. Based on a description created in this way, a set of files is written onto the disk which are required to build Docker images.

#### [Using `createPlatformImageStructure` Ant Task](#)

The `createPlatformImageStructure` ant task enables you to create Platform image structures containing Platform binaries, its configuration, aspect specific properties as well as Docker specific files, such as a `Dockerfile`.

#### [Configuring Memory in Docker Containers](#)

You can configure SAP Commerce memory settings to prevent Docker containers from exceeding memory limits and getting killed as a result.

## Platform Containerization Plugin

The Platform Containerization Plugin enhances Installer recipes with a domain-specific language (a DSL) enabling you to describe a SAP Commerce deployment structure. Based on a description created in this way, a set of files is written onto the disk which are required to build Docker images.

The documentation covers a broad range of topics explaining how to use the Plugin to write a custom containerization recipe. The topics include:

- Applying necessary plugins
- Describing and building images for:
  - Platform
  - Hsql
  - Solr
- Running images in different aspects
- Custom DSL
- Example of a complete containerization recipe
- Hooking into Plugin DSL
- The Plugin API

## Writing a Custom Recipe

A custom containerization recipe describes your setup and structure of all the components that are required to deploy SAP Commerce as a containerized application.

### Prerequisites

Applying both the Installer Platform Plugin and the Installer Platform Containerization Plugin is the starting point and the minimum setup for writing a custom containerization recipe:

```
apply plugin: 'installer-platform-plugin'
apply plugin: 'installer-platform-containerization-plugin'
```

See [Installer Platform Plugin](#) for more information.

## Platform Image

To build a Platform image, you must describe your Platform setup, describe the image structure, and then build the image based on the image structure.

We provide the details in these sections:

- Describing a Platform Setup
- Describing a Platform Image Structure
- Building a Platform Image

### **i Note**

By the `Platform` object, we mean an object of the `Platform` class.

### Describing a Platform Setup

To prepare for building a Platform image, describe first what your Platform setup looks like. Describe the extensions, properties, the database, etc. Use for that purpose the `platform` method shipped with the Plugin.

In the example we describe a simple Platform setup containing `backoffice` as the only required extension. The setup also contains one property that sets legacy persistence to `false` (only as an example):

```
def pl = platform {
    extensions {
        extName 'backoffice'
    }

    localProperties {
        property 'persistence.legacy.mode', 'false'
    }
}
```

## Describing a Platform Image Structure

Now describe the Platform image structure. Use the `deployment` method:

```
def dpl = deployment('mySampleDeployment') {
    platformImage('myPlatform') {
        basedOn pl
    }
}
```

The `basedOn` method takes as a parameter an instance of the `Platform` object. With the declaration, you state that you want to use the `Platform` object created earlier as a basis for your image.

## Building a Platform Image

Next, create a task that builds an image based on the configuration you prepared:

```
def createImageStructure << {
    dpl.createImagesStructure()
}
```

## Image Aspects

After you describe and build an image, you can run it in its full configuration, exactly as it was described. You can also run it in an aspect, choosing from all the available components only those that you want.

You saw earlier how to describe and create a simple Platform image. The image contained exactly the same set of extensions and web applications as that described in the `Platform` object definition. Very often, however, it is useful to build one bigger image containing multiple extensions and start many containers based on that image but in different aspects. For instance, consider one aspect named Only Processing - no webapps, and another one named Only Backoffice, each serving a specific purpose.

### Describing Standard Aspects

Aspects can differ from each other in that they may use different properties. You can describe such a setup using the `aspect` method. Consider this Platform setup as a base for defining example aspects:

```
def pl = platform {
    extensions {
        extensionNames 'backoffice'
    }
}
```

```

localProperties {
    property 'persistence.legacy.mode', 'false'
}
}

```

Now imagine having the following four aspects of that Platform:

- The first aspect enables Backoffice only.
- The second aspect enables Administration Console only and overrides the `persistence.legacy.mode` property to `true`.
- The third aspect enables all the web apps that the Platform instance provides.
- The fourth aspect disables all webapps.

```

def dpl = deployment('mySampleDeployment') {
    platformImage('myPlatform') {
        basedOn pl

        aspect('onlyBackoffice') {
            enabledWebApps 'backoffice'
        }

        aspect('onlyHac') {
            enabledWebApps 'hac'

            localProperties {
                property 'persistence.legacy.mode', 'true'
            }
        }

        aspect('allWebApps') {
            enableAllWebApps()
        }

        aspect('noneWebApps')
    }
}

```

## Using the Admin Aspect

You may decide that you want a specific container to serve administrative purposes. The admin aspect can help you with that. Use it to perform tasks such as, for example, initialization, update, or impex import.

Unlike the standard aspects, the admin aspect doesn't start Tomcat. It executes ant-related tasks only.

To enable an admin aspect, call the `adminAspect()` method in the `platform` closure in the Plugin DSL:

```

def dpl = deployment('mySampleDeployment') {
    platformImage('myPlatform') {
        basedOn pl
        adminAspect {
            property 'persistence.legacy.mode', 'true'
        }
    }
}

```

```

    }
}
}
```

To use the admin aspect, run the admin aspect followed by the command that you want ant to invoke. For example, to perform initialization, run your image with the `admin initialize` parameters:

```
docker run platform admin initialize
```

The container stops after the task finishes.

#### Configuring Extension Web Paths in Aspects

You can configure a specific web path for an extension per aspect. By exposing web applications at different paths, you make it easier to configure load balancer routing.

When you use aspects, it is also possible to map extensions to the same path. For example, assume that you use these two aspects:

- the backend aspect that exposes `backoffice`
- the frontend aspect that hosts `yacceleratorstorefront`

You can now configure the backend aspect to have the `backoffice` extension mapped to the `/` (root) path. At the same time you can map the `yacceleratorstorefront` extension of the frontend aspect to `/` as well. Such mapping is possible because you can start only one aspect in each container. Without aspects, the extensions paths would clash.

#### Configuration

To configure a web extension path, use the `ext_name.webroot` property in the `localProperties` section of an aspect configuration.

The following configuration of the backend aspect sets the `backoffice` path to `/`, and the `hac` path to `/hac_admin`:

```

aspect('backend') {
    enabledWebApps 'backoffice', 'hac'

    localProperties {
        property 'backoffice.webroot', ''
        property 'hac.webroot', '/hac_admin'
    }
}
```

#### Caution

To set an extension path to root, use `backoffice.webroot=` instead of `backoffice.webroot=/`.

## Hsql Image

Follow our example to learn how to describe and create a Hsql database image.

To build an Hsql database image, use the `hsqImage` method. This example is the simplest usage of the `hsqImage` API, as you only give the image a name:

```
def dpl = deployment('mySampleDeployment') {
    hsqlImage('myHsql')
}
```

Your Docker image structure gets created from a default template. The required hsql driver jar file is copied from a current Platform distribution that is known to Installer.

You can, however, adjust the image structure and choose your own jar file:

```
def dpl = deployment('mySampleDeployment') {
    hsqlImage('myHsql') {
        hsqlJarPath '/tmp/hsql-2.3.4.jar'
    }
}
```

You can give it a port so you can expose it to the outside world:

```
def dpl = deployment('mySampleDeployment') {
    hsqlImage('myHsql') {
        properties {
            property 'port', '12345'
        }
    }
}
```

You can also point to your own Docker template directory containing your jar file plus Velocity Docker file template:

```
def dpl = deployment('mySampleDeployment') {
    hsqlImage('myHsql') {
        templateDir '/tmp/myHsqlTemplate'
    }
}
```

### i Note

Such a custom template file must have a `.tmpl` extension. It gets stripped out on copying, and parsed by the Velocity engine. For instance, if a Docker file template name is `Docker tmpl`, in the image it is processed into `Docker`.

## Solr Image

Follow our example to learn how to describe and create a Solr image.

To build a Solr image, use the `solrImage` method:

```
def dpl = deployment('mySampleDeployment') {
    solrImage('mySolr')
}
```

In the `commerce-suite` context, you can use that method safely. You simply give your image a name. The Docker image gets created from a default template. The existing Solr distribution gets copied from the `commerce-suite` package.

You can adjust your Solr and choose your extracted distribution:

```
def dpl = deployment('mySampleDeployment') {
    solrImage('mySolr') {
        solrDistribution '/tmp/solrDistr'
    }
}
```

You can give it a port so that you can expose it to the outside world:

```
def dpl = deployment('mySampleDeployment') {
    solrImage('mySolr') {
        properties {
            property 'port', '12345'
        }
    }
}
```

You can even point to your own Docker template directory containing your distribution file plus the Velocity Docker file template:

```
def dpl = deployment('mySampleDeployment') {
    solrImage('mySolr') {
        templateDir '/tmp/mySolrTemplate'
    }
}
```

### i Note

Such a custom template file must have a `.tmpl` extension. It gets stripped out on copying, and parsed by the Velocity engine. For instance, if a Docker file template name is `Docker tmpl`, in the image it is processed into `Docker`.

## Custom DSL

See our example showing how you can use a custom DSL.

The `deployment(Closure)` API allows you to use a custom DSL. It is especially useful when you have a custom Dockerfile template, some binaries, and you want to fill the template with some properties, and copy into the image the binaries you have. You can handle it all easily using a custom DSL:

```
def dpl = deployment('myDeployment') {

    mySpecialImage('/path/to/your/templateDirectory') {
        property 'foo', 'bar'
        property 'baz', 'blurp'
    }
}

dpl.createImagesStructure()
```

Platform Containerization Plugin takes any file with a `tmpl` extension from `/path/to/your/templateDirectory` and replaces any occurrence of any property found in that template file with a value provided in DSL.

See how we arrived at having in a resulting image a file called `Example.txt` filled in with the properties provided in the example above.

In `/path/to/your/templateDirectory` create an `Example.txt tmpl` file including some content, for instance:

```
Say $foo, $baz
```

After executing the `dpl.createImagesStructure()` method, you get an image with the `Example.txt` file including the following content:

```
Say bar, blurb
```

## Complete Containerization Recipe

The recipe is an example of a complete containerization recipe. Use it as guidelines for your own recipes.

The recipe creates images for Platform, Solr, Hsqldb, and a load balancer:

```
apply plugin: 'installer-platform-plugin'
apply plugin: 'installer-platform-containerization-plugin'

def pl = platform {
    extensions {
        extensionNames 'backoffice'
    }

    localProperties {
        property 'persistence.legacy.mode', 'false'
    }
}

def dpl = deployment('mySampleDeployment') {
    hsqlImage 'myHsql'

    solrImage 'mySolr'

    loadBalancer '/tmp/LoadBalancerTemplate'

    platformImage('myPlatform') {
        basedOn pl

        aspect('onlyBackoffice') {
            enabledWebApps 'backoffice'
        }

        aspect('onlyHac') {
            enabledWebApps 'hac'

            localProperties {

```

```

        property 'persistence.legacy.mode', 'true'
    }

}

aspect('allWebApps') {
    enableAllWebApps()
}

aspect('noneWebApps')
}

task createImagesStructure {
    doLast {
        dpl.createImagesStructure()
    }
}

```

Note that the `loadBalancer '/tmp/LoadBalancerTemplate'` call uses Custom API capabilities.

## Hooking into Plugin DSL

Follow the provided steps to hook into Plugin DSL.

For more information on how to write your own plugin, see the example in the [installer-template-plugin] (<https://stash.hybris.com/projects/DIST/repos/installer-template-plugin/browse>) repository.

After you correctly set up your plugin, add the dependency to `installer-platform-containerization-plugin` in `build.gradle`:

```

dependencies {
    compile 'de.hybris.installer.plugin.platform.containerization:installer-platform-containerization'
}

```

Create a class that extends `AbstractImageHandler`. From this class, implement the `createImageStructure()` method. It is responsible for building an image structure.

In your main plugin class in the `apply(Project)` method you can add your own method to the `Deployment` class and use the previously created handler as a delegate class as follows:

```

void apply(Project project) {

    Deployment.metaClass.yourMethodName = { String name, Closure closure ->
        def handler = new YourHandler(name: name, project: project)
        registerImageHandler(handler)

        closure.resolveStrategy = Closure.DELEGATE_ONLY
        closure.delegate = handler
        closure()
    }
}

```

From now on you are able to write the following piece of code in a recipe:

```
apply plugin: 'installer-platform-plugin'
apply plugin: 'installer-platform-containerization-plugin'

def dpl = deployment {

    yourMethodName('someName') {

    }

}

}
```

If your handler has, for instance, the following method:

```
class YourHandler {
    String myProperty

    void superProperty(String myProperty) {
        this.myProperty = myProperty
    }
}
```

you will be able to write the following piece of code in the recipe:

```
apply plugin: 'installer-platform-plugin'
apply plugin: 'installer-platform-containerization-plugin'

def dpl = deployment {

    yourMethodName('someName') {
        superProperty 'someValue'
    }

}
```

## Platform Containerization Plugin API

See the list of all the methods the Platform Containerization Plugin delivers.

### Available APIs

#### **i Note**

By the term `Platform` object, we mean an object of the `Platform` class.

The APIs include:

- Deployment API
- PlatformImageHandler API
- SolrImageHandler API

- HsqlImageHandler API
- Global Variables

## Deployment API

### Public Methods

- **deployment(Closure)**: a top level method used for describing the whole deployment structure. This method takes **Closure** as a parameter in which you can set all necessary information by using these methods:
  - **platformImage(String, Closure)**: allows you to describe a Platform image. This method takes two parameters: name of the image as a **String**, and **Closure** in which you can set all necessary information about the image
  - **solrImage(String, Closure)**: allows you to describe a Solr platform image. This method takes two parameters: name of the image as a **String**, and **Closure** in which you can set all necessary
  - **solrImage(String)**: allows you to describe a Solr platform image. This method takes one parameter: in which you can set all necessary information about the image.name of the image as a **String**. All settings are taken from a default Docker image template.
  - **hsqlImage(String, Closure)**: allows you to describe a Hsql platform image. This method takes two parameters: name of the image as a **String**, and **Closure** in which you can set all necessary information about the image.
  - **hsqlImage(String)**: allows you to describe Hsql platform image. This method takes one parameter: name of the image as a **String**. All settings are taken from a default Docker image template.
- **createImagesStructure()**: writes down the whole deployment structure into the disk. By default, the structure is written into **Installer work** directory.
- **getOutputDir()**: returns a full path to the output image directory of the current deployment as a **String**.
- **adminAspect()**: enables generation of the admin aspect image that allows you to execute ant tasks such as initialization.

### Protected Methods

- **registerImageHandler(AbstractImageHandler)**: allows you to register a new image handler

## PlatformImageHandler API

### Public Methods

- **basedOn(String)**: allows you to set a base Platform instance from which an image is created. This method is mandatory.
- **tomcatDistribution(String)**: allows you to set a path to the Apache Tomcat distribution folder. This method is mandatory.
- **serverXmlTemplate(String)**: allows you to set a path to the **server.xml** template file.
- **customTomcatFiles**: allows you to set the path to additional, custom files that will be copied into the **tomcat** directory.
- **templatesProperties(Closure)**: allows you to provide additional properties that will be used as a replacement for any **\*.tmpl** files found in an image template directory using the Velocity engine
- **aspect(String, Closure)**: allows you to describe an aspect of an image.

#### i Note

Think about aspects as a kind of role the resultant image is going to serve as.

This method takes two parameters: the name of the image as a **String**, and the **Closure** in which you can set all necessary information using these methods:

- **enabledWebApps(String...)**: allows you to set a comma-separated list of extension names for which webapps must be enabled for the current aspect. If extension names are not provided, all webapps are disabled by default.
- **enableAllWebApps()**: allows you to enable all webapps that are configured in the base **Platform** object by default.

- `localProperties(Closure)`: allows you to set additional properties for an aspect. This method takes a `Closure` as a parameter in which you can set all necessary information in the same form as in a `localProperties` block for the `Platform` object.
- `templatesProperties(Closure)`: allows you to provide additional properties that will be used as a replacement for any `*.tmpl`
- `adminAspect(Closure)`: allows you to add a special admin aspect. This method takes `Closure` as a parameter in which you can set all necessary information in the same form as in the `localProperties` block for the `Platform` object.
- `adminAspect()`: allows you to add a special admin aspect with default settings.
- `createImagesStructure()`: creates an image structure onto a disk.
- `getOutputDirName()`: returns the name of the directory under which an image is written.

## SolrImageHandler API

- `solrDistribution(String)`: allows you to set a path to the Solr distribution folder. If you omit it, plugin tries to find a Solr distribution within commerce suite.
- `templateDir(String) null`: allows you to set a path to the image template directory containing a Docker file. If you omit it, plugin tries to find a template directory within SAP Commerce
- `properties(Closure)`: allows you to set additional properties for a Solr instance. This method takes a `Closure` as a parameter in which you can set all necessary information in the same form as in the `localProperties` block for the `Platform` object.

## HsqlImageHandler API

- `hsqldbJarPath(String)`: allows you to set a path to the hsqldb jdbc driver jar. If you omit it, the Plugin tries to find the driver in the `platform/lib/dbdriver` path.
- `templateDir(String)`: allows you to set a path to the image template directory containing a Docker file. If you omit it, the Plugin tries to find the template directory within SAP Commerce.
- `properties(Closure)`: allows you to set additional properties for a Solr instance. This method takes a `Closure` as a parameter in which you can set all necessary information in the same form as in the `localProperties` block for the `Platform` object.

## Global Variables

The Plugin comes with two global variables which are accessible from the Gradle `project` object:

- `project.outputImagesDir`: the full path to the main directory of output images. By default it is `project.installerWorkDir/output_images`.
- `project.containerizationPluginTmpDir`: the full path to the Plugin temp directory. By default it is `project.installerWorkDir/containerization_plugin_tmp`.

Keep in mind that `project.installerWorkDir` is defined in `installer-platform-plugin` and refers to the `work` subdirectory under `Installer` main directory.

## Using `createPlatformImageStructure` Ant Task

The `createPlatformImageStructure` ant task enables you to create Platform image structures containing Platform binaries, its configuration, aspect specific properties as well as Docker specific files, such as a `Dockerfile`.

You need a Platform image structure to build a Docker image. You can build and run your image in a Docker container using respectively `docker build ...` and `docker run ....`

In this document we show you how to use the `createPlatformImageStructure` ant task to create Platform image structures. We provide examples both for creating image structures and building images.

## Prerequisites

Before you start creating Platform image structures using `createPlatformImageStructure`, run the following commands:

1. Build your project:

```
ant clean all
```

2. Prepare unpacked production binaries:

```
ant production -Dproduction.include.tomcat=false -Dproduction.legacy.mode=false -Dtomcat.legacy
```

The binaries are created in the default `/temp/hybrisserver` directory.

The only required parameter is the external Tomcat directory. Since there is no default defined, you can download and unpack your Apache Tomcat anywhere and pass this path name into the ant task.

## Creating a Platform Image Structure

To create an image structure, you can run `createPlatformImageStructure` without parameters:

```
ant createPlatformImageStructure
```

For other considerations, see the sections that follow.

### Using Parameters

There are a few parameters you can provide for the ant target:

```
ant createPlatformImageStructure -DproductionPackagesDir=<MY_PRODUCTION_PACKAGES_DIR> -DplatformImageDir=<MY_PLATFORM_IMAGE_DIR>
-DserverXmlTemplate=<MY_SERVER_XML> -DplatformImageAspects=<MY_ASPECTS>
```

Available Parameters

Parameter	Required	Meaning	Default
<code>productionPackagesDir</code>	No  If not provided - production packages must exist under the default location	The unpacked production packages location, the input for the whole task, typically generated by ant <code>production</code>	<code>HYBRIS_TEMP_DIR/hybrisserver</code>
<code>platformImageDir</code>	No	The target output folder, which will contain the generated image structure	<code>HYBRIS_TEMP_DIR/platformimage-\${DSTAMP}-\${TSTAMP}</code>
<code>externalTomcatDir</code>	Yes	The unpacked Tomcat directory location	
<code>serverXmlTemplate</code>	No	You can define your own <code>server.xml</code> and	<code>platformhome/bootstrap/resources/containerization/platformimage</code>

Parameter	Required	Meaning	Default
		place the location here. Using your own <code>server.xml</code> template you can customize, for example how you want to set the ports and other <code>server.xml</code> specific settings.	
<code>platformImageAspects</code>	No  If not provided - only the default aspect is generated	You can provide your own aspects to be generated into the image structure.  Provide, however, the location to a file describing the <code>aspectName</code> to <code>aspectPathmapping</code> . See Handling Aspects for details.  There is always the default aspect generated, and your own if you provide it here.	

## Handling Aspects

The `platformImageAspects` parameter enables you to customize your own aspects. You must, however, meet certain requirements.

You must provide as an argument for the `platformImageAspects` parameter a path to the file describing the `aspectName` to `aspectPathmapping`. Such a file may look as follows:

```
aspectMap.properties
aspectABC=/Users/zzz//aspectABC
aspectDEF=/Users/zzz//aspectDEF
aspectXYZ=/Users/zzz//aspectXYZ
```

Assuming you have aspect folders defined as earlier, you must place under these folders some aspect-specific properties:

- `localProperties.properties`: contains `local.properties`, which will be merged under the generated aspect with the default `local.properties`.
- `enabledWebApps.properties`: contains the `enabledWebApps` property listing extensions to be enabled for a given aspect, for example:

```
enabledWebApps=mediaconversion, hac
```

When you enable particular webapps for a given aspect, there will be no Tomcat context files generated under this aspect for any other webapp extension names.

For more information about aspects, see [Image Aspects](#).

## Example

### Creating a Platform Image Structure

The example shows how to generate a Platform image structure under the default location with the aspect configuration as described in the input `aspectsmap.properties` file.

For the purposes of the example we assume that:

- you generated the production packages under the default location
- you unpacked the external Tomcat under `/Users/ixxx/dockerPOC/apache-tomcat-8.0.38`

To create the image structure, run:

```
ant createPlatformImageStructure -DexternalTomcatDir=/Users/ixxx/dockerPOC/apache-tomcat-8.0.38 -DplatformName=platform
```

### Building an Image

The example shows how to build a Platform image and start a Docker container with it. For the purposes of this example we assume that:

- you have created a Platform image structure the way we did above
- `aspectABC` has `mediaconversion` on the `enabledWebApps` list

Build your image by calling from within the generated image structure folder:

```
docker build -t platform .
```

Start the docker container with the Platform image:

```
docker run -it --rm -v /Desktop/dockerPOC/secrets\:/etc/ssl/certs/hybris -v /Desktop/dockerPOC/tmp_me
```

You can start another image using the `aspectABC` aspect:

```
docker run -it --rm -v /Desktop/dockerPOC/secrets\:/etc/ssl/certs/hybris -v /Desktop/dockerPOC/tmp_me
```

Now you have two docker containers running. You can access the web pages on ports 8099 (default aspect) and 8199 (`aspectABC`). In the `aspectABC` aspect you can only access the `mediaconversion` webapp (no Administration Console or others).

## Configuring Memory in Docker Containers

You can configure SAP Commerce memory settings to prevent Docker containers from exceeding memory limits and getting killed as a result.

If you assign a resource limit to a container, the operating system restricts container access to the CPU and/or memory. If a process running inside the container exceeds the memory limit, the container gets killed. You should avoid such situations by properly configuring SAP Commerce memory settings.

You can use the `$CATALINA_MEMORY_OPTS` and `$ADDITIONAL_CATALINA_OPTS` environment variables to configure the memory options in a SAP Commerce image. SAP Commerce sets the maximum and minimum heap size to 2GB by default. Your

configuration, however, should always meet the requirements of your environment.

If you don't specify the `-Xms` `-Xmx` settings explicitly, JVM tries to set these values automatically using the process called ergonomics. For example, the `XX:MaxRAMFraction` property specifies how much of available memory should be assigned for the heap space. We recommend setting memory explicitly instead of relying on automatic configuration. There are many more memory areas used by JVM besides heap, such as Class, Thread, Code, GC, Compiler, Internal and symbol memory areas. For that reason it is problematic to calculate how much memory is enough for your application. When relying on `MaxRAMFraction` (although it is not recommended), make sure that JVM uses the container memory limit instead of the system host. You can enable it by the `-XX:+UnlockExperimentalVMOptions` and `-XX:+UseCGroupMemoryLimitForHeap` properties.

## Data Management with ImpEx

The ImpEx engine matches data to be imported to the SAP Commerce type system. ImpEx allows separate data to be imported into two individual CSV files, one for the matching to the type system and the other file for the actual data. That way, swapping input files is easy. Importing items via XML-based files is not supported.

There are three main fields of use for ImpEx:

1. During Development:

- Importing sample data.
- Testing of business functionalities.
- Creating sample data during SAP Commerce initialization.
- Providing an easy way to create a project initial data.

2. Migration:

- Migrating existing data from one SAP Commerce installation to another (during a version upgrade, for example).

3. In an Operational SAP Commerce:

- Synchronizing data in SAP Commerce with other systems.
- Creating a backup of SAP Commerce data.
- Providing run-time data for CronJobs.

**→ Tip**

ImpEx import is based on the ServiceLayer. This means that actions like `INSERT`, `UPDATE`, and `REMOVE` are performed using `ModelService`, thus the ServiceLayer infrastructure like interceptors and validators is triggered. Read [ImpEx Import Using Jalo or ServiceLayer](#).

## Convention Over Configuration

SAP Commerce uses the *Convention over configuration* principle to simplify or eradicate the need for writing configuration files. An example is the functionality for importing essential and project data as described in [ImpEx Import for Essential and Project Data](#).

## ImpEx Import for Essential and Project Data

The Convention over Configuration principle is adopted in SAP Commerce to simplify and reduce the need for writing configuration files. As a result, ImpEx files for essential data and project data can be prepared without the need for additional data configuration.

## Convention

During the initialization and update processes, the platform looks for ImpEx files in the `<extension_name>/resources/impex` folder. In particular:

- For essential data: The platform scans the `<extension_name>/resources/impex` folders for files with names that match the pattern `essentialdata*.impex` and imports the files during the essential data creation.
- For project data: The platform scans the `<extension_name>/resources/impex` folders for files with names that match the pattern `projectdata*.impex` and imports the files during the project data creation.

The **ImpEx** directory does not exist by default. You must create it and copy files to it.

For example, if you have the following folder structure:

- `resources/impex/essentialdataOne.impex`
- `resources/impex/essDataOne.impex`
- `resources/impex/DataOne.csv`
- `resources/impex/projectdataOne.impex`

The files `essentialdataOne.impex` and `projectdataOne.impex` are located and imported during the essential and project data imports respectively.

### i Note

To import your impex files correctly, make sure that their file names are unique across all extensions during import, for example by adding extension names to impex file names:

```
essentialdata-<extension>-*.impex
projectdata-<extension>-*.impex
```

## Configuration

If you have special folder structures or want to use another folder in the resources, you must override the configuration in your `local.properties` file:

- For essential data, add the property `<extension_name>.essentialdata-impex-pattern`.
- For project data, use `<extension_name>.projectdata-impex-pattern`.

For example, assume that you have the following folder structure:

- `resources/test1.csv`
- `resources/subfolder/test2.csv`
- `resources/impex/subfolder/subfolder/test3.csv`

In this structure, only the `test1.csv` file has the pattern `<extension_name>.essentialdata-impex-pattern=*.csv`. In contrast to the example above, the pattern: `extension_name.essentialdata-impex-pattern=**/*.csv` includes `test1.csv`, `test2.csv` and `test3.csv`.

If you want your configuration to work as the default does, you must set the pattern to `<extension_name>.essentialdata-impex-pattern=impex/essentialdata*.impex`.

The order in which the files are imported depends on the Virtual Machine (VM), because the files are loaded as Java resources. If you want to specify a particular order, you can create one import file that matches your import pattern and some other files, which are imported in the first file, for example:

#### projectdataOne.impex

```
... IMPEX FILE that includes CVS:  
"#% impex.includeExternalData(SampleDataManager.class.getResourceAsStream("/1.csv"), "utf-8", 0,  
"#% impex.includeExternalData(SampleDataManager.class.getResourceAsStream("/2.csv"), "utf-8", 0,
```

Alternatively, you can use an initialization hook to determine the import order. For more information, see [Hooks for Initialization and Update Process](#).

## Related Information

### [Initializing and Updating SAP Commerce](#)

## ImpEx Import Using Jalo or ServiceLayer

SAP Commerce uses ServiceLayer to import data with ImpEx by default. It means that actions like INSERT, UPDATE, and REMOVE are performed using `ModelService`, thus the ServiceLayer infrastructure like interceptors and validators are triggered. When necessary, you can execute import using the Jalo layer.

You can start using the Jalo layer globally by setting `impex.legacy.mode` property to `true`. You can do this in two different ways:

- Using code
- Adding a new property to `local.properties` file

## Modify the Property Using Code

To modify the default value of the `impex.legacy.mode` property, implement the `Config` utility class:

```
// Read old value  
String final legacyModeBackup = Config.getParameter(ImpExConstants.Params.LEGACY_MODE_KEY);  
// Set it to true  
Config.setParameter(ImpExConstants.Params.LEGACY_MODE_KEY, "true");  
// Perform some operations  
// ...  
// Set it again to previous value  
Config.setParameter(ImpExConstants.Params.LEGACY_MODE_KEY, legacyModeBackup);
```

## Modify the Properties File Manually

To use Jalo, add `impex.legacy.mode=true` to the `local.properties` file. Restart your application server.

## Enabling ImpEx Import with the Jalo layer for Specific Import Operation

To use the Jalo layer only for some specific ImpEx import operation with usage of `ImportService`, use the `setLegacyMode` method from the `ImportConfig` class:

```
// Assume you have already a bytes object containing a script content  
byte[] bytes;  
final ImportConfig config = new ImportConfig();  
// Switch on legacy mode
```

```

config.setLegacyMode(Boolean.TRUE);
config.setScript(new StreamBasedImpExResource(new ByteArrayInputStream(bytes), "en"));
// Now use Config object
final ImportResult result = importService.importData(config);

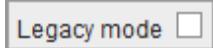
```

Notice that `config.setLegacyMode` method uses a `Boolean` object as a parameter. If you don't use it, then the global setting from `project.properties` file of the `impex` extension is used.

## Test Import Script Using the SAP Commerce Administration Console

You can test your import script with the Jalo layer in SAP Commerce Administration Console:

1. Open SAP Commerce Administration Console.
2. Go to the **Console** tab and select **ImpEx Import** option.
3. Before importing, select **Legacy mode** to use the Jalo layer.



## Related Information

[Administration Console](#)

[ImpEx API](#)

## Analyzing Import Results for Errors

You can analyze import results for errors. To perform error analysis, use the `collectImportErrors` method of the `ImportService` class.

First, run your import - the `collectImportErrors` method requires an `ImportResult` object as a parameter:

```
Stream<? extends ImpExError> collectImportErrors(ImportResult result)
```

The method returns a `Stream` of `ImpExError` objects, so it has lazy behavior.

The `ImpExError` interface is a base interface that describes a general error:

```

public interface ImpExError
{
    String getItemType();

    ProcessMode getMode();

    String getErrorMessage();

    Map<Integer, String> getSource();
}

```

`ImpExError` provides methods that enable you to get an item type code, `ProcessMode`, error message, and line source. `ProcessMode` says whether the error relates to the `INSERT`, `UPDATE`, `REMOVE`, or `INSERT_UPDATE` mode.

There are two more specialized interfaces that extend `ImpExError`:

- `ImpExHeaderError` represents an invalid header error:

```
public interface ImpExHeaderError extends ImpExError
{
    HeaderDescriptor getHeader();

    Exception getException();
}
```

Header related errors are present only if a particular header is invalid (for instance, contains an invalid column name, invalid type code, etc.).

- `ImpExValueLineError` represents a **value line** error:

```
public interface ImpExValueLineError extends ImpExError
{
    ValueLine getLine();

    ImpExLineErrorHandlerType getErrorHandlerType();

    enum ImpExLineErrorHandlerType
    {
        PARTIALLY_PROCESSED, NOT_PROCESSED, PARTIALLY_PROCESSED_WITH_ERROR, NOT_PROCESSED_WITH_ERROR
    }
}
```

Value line related errors are present always for any dumped and not resolved lines - even if they have been partially processed. To determine what kind of value line error you have, use the `ImpExValueLineError#getErrorHandlerType` method. It returns an enum value you can see above.

Here is an example usage of the `ImpEx Error API`:

```
final String inputScript = "INSERT Title;code[unique=true];name\n" //
+ ";title1;name1;\n" //
+ ";title1;name1;\n" //
+ ";title3;name3;\n";
final ImportConfig config = new ImportConfig();
config.setScript(inputScript);
config.setSynchronous(true);
final ImportResult importResult = importService.importData(config);

final Stream<? extends ImpExError> result = importService.collectImportErrors(importResult);

// To print out all ImpExValueLineErrors:
errors.filter(e -> e instanceof ValueLineError).map(ValueLineError.class::cast).forEach(System.out::println);

// To print out all ImpExHeaderErrors:
errors.filter(e -> e instanceof ImpExHeaderError).map(ImpExHeaderError.class::cast).forEach(System.out::println);
```

### i Note

The `collectImportErrors` method returns an empty `Stream` if `ImportResult` has no dumped lines - this means your import was successful.

# ImpEx Syntax

SAP Commerce ships with an integrated text-based import/export extension called ImpEx, which allows creating, updating, removing, and exporting platform items such as customer, product, or order data to and from Comma-Separated Values (CSV) data files - both during run-time and during the SAP Commerce initialization or update process.

## CSV Files

The ImpEx extension uses Comma-Separated Values (CSV) files as the data format for import and export. As there is no formal specification, there is a wide variety of ways in which you can interpret CSV files.

### ImpEx Syntax in CSV Files

An ImpEx-compliant CSV file contains several different kinds of data. The following screenshot is a representation of a sample CSV file with different colors for different kinds of data.

## CSV Files

The ImpEx extension uses Comma-Separated Values (CSV) files as the data format for import and export. As there is no formal specification, there is a wide variety of ways in which you can interpret CSV files.

You can find some sort of quasi-standard in [Common Format and MIME Type for Comma-Separated Values \(CSV\) Files](#) document, as it is followed by most implementations. It is quoted here in excerpts:

1. Each record is located on a separate line, delimited by a line break (CRLF), for example:

```
aaa,bbb,ccc CRLF
zzz,yyy,xxx CRLF
```

2. The last record in the file may or may not have an ending line break, for example:

```
aaa,bbb,ccc CRLF
zzz,yyy,xxx
```

3. There may be an optional header line appearing as the first line of the file with the same format as normal record lines. This header contains names corresponding to the fields in the file and should contain the same number of fields as the records in the rest of the file, for example:

```
field_name,field_name,field_name CRLF
aaa,bbb,ccc CRLF
zzz,yyy,xxx CRLF
```

The presence or absence of the header line should be indicated using the optional header parameter of this MIME type.

4. Within the header and each record, there may be one or more fields, separated by commas. Each line should contain the same number of fields throughout the file. Spaces are considered part of a field and should not be ignored. The last field in the record must not be followed by a comma, for example:

```
aaa,bbb,ccc
```

5. Each field may or may not be enclosed in double quotes (however some programs, such as Microsoft Excel, do not use double quotes at all). If fields are not enclosed with double quotes, then double quotes may not appear inside the fields, for example:

```
"aaa", "bbb", "ccc" CRLF
zzz,yyy,xxx
```

6. Fields containing line breaks (CRLF), double quotes, and commas should be enclosed in double-quotes, for example:

```
"aaa", "b CRLF
bb", "ccc" CRLF
zzz,yyy,xxx
```

7. If double-quotes are used to enclose fields, then a double-quote appearing inside a field must be escaped by preceding it with another double quote, for example:

```
"aaa", "b""bb", "ccc"
```

Due to the simple structure, CSV files are common for data exchange purposes. Compared to XML files, CSV files require less processing and memory resources. For additional details on CSV files, please refer to [Common Format and MIME Type for Comma-Separated Values \(CSV\) Files](#) document.

In a CSV file, by default, one line of entries represents one record of data. In other words: in a CSV file, one line defines one item, that is one customer, one car, one t-shirt, and so on. A single line can contain several attribute values, for example in case of a t-shirt: size, color, and brand. CSV breaks down individual record entries via a so-called delimiter, which is, by default, the comma, hence the name. The following fictitious code sample gives you an example of this:

```
green,42,BlackLabel
red,42,BlackLabel
green,35,BlackLabel
```

## CSV Files in ImpEx

The ImpEx extension relies on the above definition of the CSV format except for the defined special characters, which may be different but can be configured. These characters are:

- Line break: character for separating one line from another.  
Default is the Java system property `line.separator`.
- Column delimiter: isolates values that belong in columns, in other words records.  
Default is the semicolon (:). You can change it using the `csv.fieldseparator` property.
- Enclosing character: character for enclosing a column. Default is the double-quote ". You can change it using the `csv.quotecharacter` property.
- Individual delimiter: these delimiters isolate values that belong in one database field, such as with Collections (here the default is comma ,). The individual delimiters depend on the column translator used and can be configured separately. Refer to [Header](#).

Furthermore, the ImpEx extension provides the multi-line separator symbol for breaking lines only visually (soft line break). The ImpEx extension does not interpret the line break. The support of multi-lines can be disabled using the API.

For more information, see:

- [https://docs.oracle.com/.../getProperties\(\).](https://docs.oracle.com/.../getProperties().) : `getProperties()` documentation
- [Configuration Properties](#)
- [API Documentation and YAML Files](#)

## Managing CSV Files Using Third Party Tools

The default value for these delimiters depends on the system locale settings when using third party tools. Using an ImpEx CSV file on different locale settings has to be handled with care. For example, during the tests a German version of Microsoft Excel 2003 SP2 used commas for individual delimiters and semicolons for column delimiters by default if running on an English version of Microsoft Windows XP SP2. The default delimiters for Microsoft Windows XP versions with an English locale are semicolons for individual delimiters and commas for column delimiters.

The following code snippet, for example, would cause different interpretations, depending on the locale settings:

```
value1,value2,value3;value4;value5,value6
```

German locale	English locale
<p>three values:</p> <ul style="list-style-type: none"> <li>• value1,value2,value3</li> <li>• value4</li> <li>• value5,value6</li> </ul>	<p>four values:</p> <ul style="list-style-type: none"> <li>• value1</li> <li>• value2</li> <li>• value3;value4;value5</li> <li>• value6</li> </ul>

Furthermore, you can encounter serious problems when switching the editing programs because of the different character encoding used. Saving a CSV file in Windows encoding and importing it using **ImpEx** in UTF-8 encoding destroys special characters like umlauts. Always check the character encoding used by the program and configure it as described in [ImpEx API](#).

The most convenient way of dealing with CSV files is using an editor with syntax highlighting or spreadsheet processor. Among such software, SAP recommends the following applications:

- UltraEdit - text editor that contains customizable syntax highlighting.
- Microsoft Office Excel - a powerful commercial spreadsheet processor. Microsoft Excel CSV spreadsheets are limited to a maximum of 65535 lines. This means that if you want to manage CSV files longer than 65535 lines, you have to split those files into parts. If you try to open a file longer than 65535 lines, Excel informs you that the file is beyond the specification. The problem was partially resolved since Microsoft Excel 2003, although the application supports more than 65535 lines only in its native format.
- OpenOffice.org Calc - freeware spreadsheet processor from Sun Microsystems Inc. It is free and also supports a customized import of CSV files (you can use the usual open file method, no extra import is needed). When saving a file as CSV the original encoding is used, or windows-1252 if the file was created from scratch.

## ImpEx Syntax in CSV Files

An ImpEx-compliant CSV file contains several different kinds of data. The following screenshot is a representation of a sample CSV file with different colors for different kinds of data.

The different line types are described in detail below:

- Headers (blue and light green text color in the screenshot), refer to [Header](#).
- Lines of values (**black** text color), refer to [Value Line](#).
- Comments (lavender text color), refer to [Comment](#) section.
- Macro definitions (red text color), refer to [Definition Line - Macro](#).
- BeanShell calls (not shown in the screenshot), which are documented throughout various pages in [ImpEx API](#).
- Lines declaring user rights import (not shown in the screenshot), refer to the *User Rights* section of [ImpEx API](#).

Microsoft Excel - StoreFoundation_clothescatalog.csv				
A69	=			E
6	\$supercategories = supercategories ( code, catalogVersion ( catalog ( id[default = 'clothescatalog'] ), version[default = 'Staged'] ) )			
7	\$baseProduct = baseProduct ( code, catalogVersion ( catalog ( id[default = 'clothescatalog'] ), version[default = 'Staged'] ) )			
8				
9	INSERT_UPDATE Catalog	id[unique=true]	name[lang=de]	name[lang=en]
10		clothescatalog	Bekleidungs-Katalog	Clothing
11				supplier(uid)
12	INSERT_UPDATE CatalogVersion	catalog(id)[unique=true]	version[unique=true]	languages(isocode)
13		clothescatalog	Online	de,en
14		clothescatalog	Staged	de,en
15				true
16	INSERT_UPDATE category	code[unique=true]	name[lang=de]	name[lang=en]
17		CL1100	Jeans	Jeans
18		CL1200	Shirts	Shirts
19		CL2000	Schuhe	Shoes
20		CL2100	Herrenschuhe	Men's shoes
21		CL2200	Freizeitschuhe	Leisure shoes
22		CL2300	Damenschuhe	Lady's shoes
23		topseller	Topseller	Topseller
24				CL2000
25	INSERT_UPDATE category	code[unique=true]	allowedPrincipals(uid)	\$CatalogVersion
26		CL1100	customergroup	clothescatalog:Staged
27		CL1200	customergroup	clothescatalog:Staged
28		CL2000	customergroup	clothescatalog:Staged
29		topseller	customergroup	clothescatalog:Staged
30				
31	INSERT_UPDATE Media	code[unique=true]	\$CatalogVersion	mime
32	# JEANS MEDIAS			realfilename
33		aaa015x04a	clothescatalog:Staged	image/jpeg
34		aaa015x04a_big	clothescatalog:Staged	image/jpeg
35		aaa015x04b	clothescatalog:Staged	image/png
36		aaa015x04b_big	clothescatalog:Staged	image/png

## Header

A header is a single line defining a mapping of the following value lines to the type system. A header applies to all processed value lines until the next header or until the end of file, whichever comes first. You can put any number of headers into a CSV file.

An ImpEx header line has the following structure:

```
mode type[modifier=value];attribute[modifier=value];attribute[modifier=value];attribute[modifier=value];
```

- **Mode:** Specifies what is to be done with the following value lines (insert, update, and so on) - see [Header Mode](#).
- **Type:** Defines the type of item to be processed (category, product, media, type, and so on) - see [Header Type](#).
- **Attribute:** Describes which item attributes the columns are mapped to. The value lines supply the actual values for the items that are translated using the header settings - see [Header Attributes](#)
- **Modifier:** Gives additional information for translating a value record to the mapped type attribute - see [Header and Attribute Modifier](#).

Example for a header row:

```
INSERT_UPDATE category;code[unique=true];name[lang=de];name[lang=en];$supercategories;$thumbnail;desc;
```

This header states that each following value line creates or updates a category instance, until another header occurs.

### i Note

The whole header syntax is case insensitive including the attribute qualifiers.

## Related Information

[The Type System](#)

## Header Mode

The mode of a header specifies what is to be done with the following data.

In case of an import it describes how to map and translate the following value lines to the data model of the platform. In case of an export, it describes how to map and translate the following items to the target CSV file. During export, headers are also written, so exported CSV files can be re-imported via the ImpEx extension.

There are four header modes:

Mode	Comment/Description
INSERT	<p>Creates a new item in SAP Commerce from the processed value line. By default, ImpEx does not check if an item with the specified attributes already exists. Therefore, trying to insert an item with a code that already exists may throw an exception caused by the Commerce core. For that reason, it is recommended to use the unique modifier, where ImpEx prechecks a violation of attribute uniqueness.</p>
UPDATE	<p>Selects an existing item in SAP Commerce from the combination of the values of all columns marked with the unique modifier referred to as key attributes, and sets the item attributes to the values specified in the value line.</p> <p>If there is no item that matches the values of all key attributes, the value line cannot be resolved. The value line is dumped into a temporary file and the ImpEx extension tries to resolve the value line later on. Refer to <a href="#">ImpEx API</a>, section <b>Import Using an Instance</b> for details.</p> <p>Also note that the default value for attributes is <code>null</code>. This means that any attributes that are referenced by the header but have no values specified in the value line are not ignored, but are explicitly set to <code>null</code>. For example, the following ImpEx lines create a customer named "ahertz" with the name attribute set to <code>null</code> as the second line after the header specifies no value for the name attribute.</p> <pre>UPDATE User;uid[unique=true];name;customerID; ;ahertz;Anja Hertz;; ;ahertz;;K2006-C0005;</pre> <p>To set values for both the name and the customerID attributes, use one of the following approaches:</p> <ul style="list-style-type: none"> <li>Specify all values in one single line:</li> </ul> <pre>UPDATE User;uid[unique=true];name;customerID; ;ahertz;Anja Hertz;K2006-C0005;</pre> <ul style="list-style-type: none"> <li>Re-specify all values that have been set in a prior line:</li> </ul> <pre>UPDATE User;uid[unique=true];name;customerID; ;ahertz;Anja Hertz;; ;ahertz;Anja Hertz;K2006-C0005;</pre> <ul style="list-style-type: none"> <li>Use individual headers:</li> </ul> <pre>UPDATE User;uid[unique=true];name; ;ahertz;Anja Hertz;</pre> <pre>UPDATE User;uid[unique=true];customerID; ;ahertz;K2006-C0005;</pre>

Mode	Comment/Description
INSERT_UPDATE	Combines the effects of the INSERT and UPDATE header modes: if an item with the values of all key attributes exists, this is updated. If no item exists that matches the combination of all key attributes, the value line is used to create the item. The ImpEx extension always uses this order: UPDATE, then INSERT.
REMOVE	As in UPDATE mode, the ImpEx extension tries to find an existing item using the key attributes. If such an item exists, it is removed; otherwise a log message of level <b>warning</b> is printed to the log.

## Header Type

The type code selects the type from the SAP Commerce Type System where the following data rows create, update or remove instances.

Optionally, each line can specify a subtype of the type specified in the header at the first column. That subtype is selected for that single line only - in the next line, the type specified in the header is active again. The following code snippet references the User type in the header, but creates an instance of the **Customer** and **Employee** types each.

```
INSERT User;uid[unique=true]
Customer;SampleCustomer
Employee;SampleEmployee
```

This allows specifying an abstract type, which cannot be instantiated, in the header for the INSERT mode, as with the Job type in the following code snippet. In that case, you need to specify a non-abstract type (that is, a type that can be instantiated) in every value line.

```
INSERT Job;code[unique=true]
ImpExImportJob;sampleJob
ImpExImportJob;sampleJob2
```

Although the header references the Job type, two instances of an ImpExImportJob will be created: sampleJob and sampleJob2.

## Header Attributes

Each column followed by the header type at the header line describes an attribute of the header type where the column value of the following value lines are mapped to.

The ImpEx extension checks the specified type and its attributes and uses a Translator to match the value line's entry to the individual attribute. See [Header and Attribute Modifier](#) for more details on Translator.

It is easy to set an attribute value for attributes that are specified as AtomicTypes (such as `java.lang.String` or `java.lang.Integer`). For example, assume the following code snippet:

```
INSERT User;uid
;myUser
```

Here the `uid` attribute is of type `java.lang.String`, so additional definitions are not required. You define the attribute at the header line and write a value line with the plain text of the desired user ID.

## Item Reference Attributes

Setting attribute values is more complex with attributes specified to hold references to items in SAP Commerce.

By default, the ImpEx expects values that represent a primary key:

```
INSERT Product;code;unit
;MyProduct;2821057095693456
#2821057095693456 being the PK for the pieces unit
```

Since primary keys are not usually known externally, ImpEx alternatively allows using **attributes of the referenced item** for lookup. They're provided in brackets immediately after the column attribute name: <attributeOfTypeX>' ('<attributeFromX>(','<attributeFromX>)\* ') '.

For example, the following code snippet specifies the `unit` column to look up Unit items using its `code` attribute:

```
INSERT Product;code;unit(code)
;MyProduct;pieces
```

### i Note

Specifying referred-type attributes behind an attribute definition is called **item expression**.

You can use item expressions in a nested way, in cases where a lookup attribute is a reference to itself.

In the following example the `catalogVersion` attribute refers to a CatalogVersion item that is looked up using its `catalog` and `version` attribute. Within the catalog is a reference to itself (to a Catalog item) that is looked up using the `id` attribute:

```
INSERT Product;code;unit(code);catalogVersion(catalog(id),version)
;testCode;pieces;clothescatalog:Staged
```

If you specify an item expression using several attributes at a header line, you have to separate them with a comma. The values for such an expression at a value line are separated by a colon.

There are ComposedTypes that reference attributes of a very general type; for example, the `owner` attribute of the `Address` type is specified to be of the `Item` type and can therefore accept any item in the SAP Commerce. By consequence, you cannot directly reference an attribute specified for a subtype of `Item`, such as the `uid` attribute of a `Customer` instance, which would allow you to specify a specific customer. To refer to an attribute that belongs to a subtype of a type attribute, use the following syntax:

```
attribute(subtype.subtype_attribute)
```

For example, the following code snippet creates an `Address` type instance whose `owner` attribute (attribute of type `Item`) holds a reference to the admin user using the `Principal`'s type `uid` attribute.

```
INSERT Address;firstname;owner(Principal.uid)
;Hans;admin
```

## Skipping Columns of Data

The ImpEx extension allows skipping of entire columns of data by omitting the attribute for that column in the header.

```
INSERT myProduct;myAttribute;myAttribute2;;myAttribute4
;myValue1;myValue2;myValue3;myValue4
```

`myValue3` is ignored, because there is no header attribute mapping defined for the related column. In the following code snippet, the second column (containing the value `deutsch`) is ignored as there is no header attribute to process it:

```
INSERT Language;code;;active
;de;deutsch;true
```

## Localizing

Localized attributes are described by multiple columns using the `lang` modifier, each one containing the ISO code or PK for a single language.

Refer to [Header and Attribute Modifier](#) section for more information about attribute modifiers.

```
INSERT Product;code[unique=true];name[lang=en];name[lang=de]
;myProduct1;myProduct1's localized name;lokalisierter Name von myProduct1
;myProduct2;myProduct2's localized name;lokalisierter Name von myProduct2
```

The ImpEx extension only overwrites localized values for the languages specified in the column header. Existing localized values for other languages remain unchanged, as the following example illustrates:

```
#create some products with a German and English name
INSERT Product;code[unique=true];name[lang=en];name[lang=de]
;myProduct1;myProduct1's localized name;lokalisierter Name von myProduct1
;myProduct2;myProduct2's localized name;lokalisierter Name von myProduct2

#overwrite the English name -> the German one remains untouched
UPDATE Product;code[unique=true];name[lang=en]
;myProduct1;"my product 1"
;myProduct2;
```

Product	Resulting English localization	Resulting German localization
myProduct1	my product 1	lokalisierter Name von myProduct1
myProduct2		lokalisierter Name von myProduct2

For more information, see [Wikipedia on ISO Country Codes](#).

## Document ID

Especially when importing `partOf` item values, you must reference these items by means other than the usual unique column technique because `partOf` items do not always provide a unique key but only hold their enclosing parent as a foreign key.

The `Customer.addresses` attribute provides an example of this. The customer owns addresses exclusively. The addresses hold the customer PK as a back reference. When using ImpEx, both customer and address have to be imported in separate lines.

Removing obsolete addresses requires that you fill the `Customer.addresses` cell with all currently valid addresses. This allows the `setter` method to remove obsolete ones. You must therefore reference valid addresses using a unique key, which addresses do not provide. For cases where a reference within the ImpEx script is required, ImpEx provides a special header attribute marked with the prefix `&` to specify a virtual ID called `Document ID`. Each imported item line can optionally define such a Document ID, which allows other lines to reference it later using the same attribute qualifier (with the `&` prefix).

Example:

```
INSERT Customer;uid[unique=true];...;defaultPaymentAddress(&addrID);...
;andi;...;addr1;...
```

```
INSERT Address;addrID;owner(Customer.uid);...
;addr1;andi;...
```

The Document ID is case-sensitive, so the following code snippet would not work as the Document ID would not be recognized by the ImpEx extension (the document ID for the Customer type being **&addrID** and the header for the Address type being **&addrid**).

```
INSERT Customer;uid[unique=true];...;defaultPaymentAddress (&addrID);...
;andi;...;addr1;...
```

```
INSERT Address;&addrid;owner(Customer.uid);...
;addr1;andi;...
```

## Special Attributes

Sometimes you need to add data to an item that is not covered by an attribute, for example the media data. In these cases, the syntax of ImpEx provides a special kind of attribute definition called **special** attributes that do not have a mapping to a real attribute of the configured type.

They are marked with the @ symbol and always need the **translator** modifier, because there is no default implementation. Furthermore the specified translator has to be a realization of the **SpecialValueTranslator** interface.

The following code sample shows how to create a media with the data set from the specified file:

```
INSERT_UPDATE Media;code[unique=true];@media[translator=de.hybris.platform.impex.jalo.media.MediaData@
;myMedia;file://c:/myMedia.txt
```

The type **Media** has no **media** attribute, but specifying the **media** attribute as a special attribute instructs ImpEx not to search for such an attribute. Instead, it will call the special translator specified with the **translator** modifier, which performs the import logic.

## Alternative Pattern

For type defining attributes that belong to a generic type, such as the **Address** type **owner** attribute, which is specified to hold an instance of **Item**, the ImpEx extension allows you to specify Alternative Patterns in the header.

An Alternative Pattern specifies more than one attribute and makes the ImpEx extension determine the matching type instance from all the specified attributes. Alternative Patterns are processed from left to right.

The following code snippet, specifies two possible target attributes for the **Address** instances:

- the **Principal** type **uid** attribute
- the **AbstractOrder** type **code** attribute:

```
INSERT Address;firstname;owner(Principal.uid|AbstractOrder.code)
;Hans;admin
;Klaus;0-K2006-C0000-001
```

During import, ImpEx compares the values for the **owner** attributes against all available instances of the **Principal** and **AbstractOrder** types and automatically chooses the instance whose target attribute value (**uid** or **code**, respectively) matches the target value (**admin** and **0-K2006-C0000-001**, respectively).

When importing the first value line ;Hans;admin, the ImpEx extension searches all instances of the **Principal** type to verify if any instance exists where the **uid** attribute is set to **admin**.

- If an instance exists (and there is, by platform default), the new instance of the Address type has its owner attribute set to a reference to that Principal type instance.
- If no instance exists of the Principal type whose uid attribute is set to admin, the ImpEx continues the retrieval by trying to find an instance of the AbstractOrder type whose code attribute is set to admin.
- If there was no instance of the AbstractOrder type whose code attribute is set to admin, the ImpEx extension would try to continue the retrieval on the next specified Alternative Pattern attribute. Since there would be no further Alternative Pattern attribute, the ImpEx extension would dump the line and try to resolve it on the next import pass.

In this sample case, there is a Principal instance whose uid attribute is set to admin by platform default, so ImpEx is able to resolve the value line.

Basically, the same thing happens for the second value line ;Klaus;0-K2006-C0000-001: the ImpEx extension searches all instances of the Principal type if there is any instance whose uid attribute is set to 0-K2006-C0000-001.

- If there was an instance (but there is none, by platform default), then the new instance of the Address type would have its owner attribute set to a reference to that Principal type instance.
- Since there is an instance of the AbstractOrder type whose code attribute is set to 0-K2006-C0000-001 (at least for SAP Commerce installations with the `sampodata` extension enabled), the new instance of the Address type has its owner attribute set to a reference to that AbstractOrder type instance.
- If there was no instance of the AbstractOrder type whose code attribute is set to 0-K2006-C0000-001, then the ImpEx extension tries to continue the retrieval on the next specified Alternative Pattern attribute. Since there would be no further Alternative Pattern attribute, the ImpEx extension would dump the line and try to resolve it on the next import pass.

In this sample case, there is a AbstractOrder instance whose code attribute is set to 0-K2006-C0000-001 by using the `sampodata` extension, so ImpEx is able to resolve the value line right now.

## Header and Attribute Modifier

A modifier is an addition to an entry in a header line that specifies additional processing instructions.

For example:

```
...;attribute[modifier=value];...
```

The ImpEx extension allows specifying several modifiers at once, like this:

```
...;attribute[modifier=value,modifier=value,modifier=value];...
```

or

```
...;attribute[modifier=value][modifier=value][modifier=value];...
```

For example, the following code snippet defines that the value specified for the name attribute is the English localization version ("en" according to [Wikipedia on ISO 639-1](#)) of the name attribute.

```
name[lang=en]
```

### i Note

If languages for which impex data is imported don't exist in the database, import errors are thrown.

When you're not sure if a given language is available in your target system, use the following script to protect your files in this language from being imported:

```
#% if: de.hybris.platform.servicelayer.i18n.util.LanguageUtils.isLanguagePresent("fr")
INSERT_UPDATE Item1;code[unique=true];name[lang=fr]
...
#% endif:
```

To escape a modifier value, use the inverted comma (''). To use an inverted comma in a modifier value, escape the inverted comma itself:

```
attribute[modifier='My modifier''s value']
```

## Header-Related Modifiers

The following table contains a reference of all header-related modifiers. These modifiers apply to the entire header and are specified directly to the type referenced by the header, as in the following code snippet:

```
INSERT_UPDATE ClassificationSystemversion[cacheUnique=true];...
```

Modifier	Allowed values	Description
batchmode Import only	true / false Default is false	<p>In UPDATE or REMOVE mode, the batch mode allows modifying more than one item that matches for a combination of all unique attributes of a value line. So, if for a value line more than one item is found that matches the combination of unique attributes, the attributes specified as non-unique are updated at all found items.</p> <p>[batchmode=true]</p> <ul style="list-style-type: none"> <li>• If the batch mode is not enabled (as by default), the hybris ImpEx extension throws an exception if more than one item matches for a value line.</li> <li>• If the batch mode is enabled, the hybris ImpEx extension modifies any platform item that matches the value line.</li> </ul>
cacheUnique Import only	true / false default is false	<p>If this option is enabled, the CachingExistingItemResolver is used for existing item resolving (in case of update or remove mode) which caches by the combination of unique keys already resolved items. So when processing a value line first, it is tried to find the related item by searching the cache using the unique keys. The usage is only meaningful if an item has to be processed more than one time within a header scope. The maximum size of the cache is not restricted at the moment.</p> <p>[cacheUnique=true]</p>
processor Import only	A ImportProcessor class Default is the DefaultImportProcessor	<p>Unlike a Translator, which handles a certain column of a value line, a Processor handles an entire value line. It contains all business logic to transform a value line into values for attributes of an item in SAP Commerce (such as calls for translator classes, for example) and performs the real setting of the values. In other words: a Processor is passed a value line as input, and it creates or updates an item in SAP Commerce as its output.</p> <p>[processor=de.hybris.platform.impex.jalo.imp.DefaultImportProcessor]</p> <p><b>i Note</b></p>

Modifier	Allowed values	Description
		SAP Commerce doesn't support the Processor modifier in Distributed ImpEx.
impex.legacy.mode (since SAP Commerce version 5.1.1)	true / false. Default: false	This modifier allows enabling the legacy mode (jalo) per header. This way - in case of service layer mode enabled globally, impex will switch dynamically to legacy mode, just like for existing 'allowNull' or 'forceWrite' modifiers. The only difference is - it's set for the type, not for the column. For example:  <code>INSERT_UPDATE myProduct[impex.legacy.mode=true];myAttribute</code>

## Attribute-Related Modifiers

Modifier	Allowed Values	Description
alias Export only	A text such as <b>myAttribute</b> Default: no alias is set	With that modifier you can specify an alias name for an attribute, which makes it easier to reference the attribute in other parts of the impex file. Example: [alias=myAttribute]
allownull Import only	true / false Default: false	If set to true, this modifier explicitly allows null values for the column. It allows null values in mandatory attributes, such as the catalogVersion attribute. Example: [allownull=true]  <b>→ Tip</b> In the Service Layer mode, import may fail if allownull is set. ImpEx will check for mandatory attributes that have null values. After processing a given line, the import will switch back to the SL mode and continue with the next line.
cellDecorator Import only	An AbstractImpExCSVCellDecorator class Default: no decorator is applied.	Specifies a decorator class for modifying the cell value before interpreting it. Example: [cellDecorator=de.hybris.platform.catalog.CSVCellDecorator]
collection-delimiter Restricted to attributes of type Collection.	Any character Default: comma (,)	Allows specifying a delimiter for separating values in a Collection. For example: INSERT_UPDATE Product;collectionAttribute[(,)]
dateformat Restricted to Date and StandardDateRange types.	A date format such as <i>MM-dd-yyyy</i> or <i>dd.MM.yyyy</i> (See the Date and Time Patterns in Java API documentation) Default: <code>DateFormat.getDateInstance(DateFormat.MEDIUM, DateFormat.MEDIUM, impexReader.getLocale())</code>	Specifies the format in which the column specifies Date values. Example: UPDATE myProduct;myAttribute [dateformat='dd.MM.yyyy'] A more complex example would be to implement a ISO 8601 compliant date format, such as yyyy-MM-dd'T'HH:mm:ss.SSSZ. The T in the date format is a separator between the date and time parts. In ImpEx, the single quotes have to be escaped by doubling. Example: UPDATE myProduct;myAttribute[dateformat='yyyy-MM-dd''T''HH:mm:ss.SSSZ'] 0700"
default	A value that matches the current attribute column type.	Sets the default for values of this column. Example: INSERT_UPDATE myProduct;myAttribute[default='0']
forceWrite Import only	true / false Default: false	If set, it tries to write into read-only columns. Success depends on the database system being tried. Example: [forceWrite=true]  <b>→ Tip</b>

Modifier	Allowed Values	Description
		In the Service Layer mode, import may fail if <code>forceWrite</code> is set. In After processing a given line, the import will switch back to the SL mode.
ignoreKeyCase Import only Restricted to attributes of type <code>Item</code>	true / false Default: false	If set, the case of the text that specifies attributes of items for resolution is ignored. Example: [ignoreKeyCase=true]
ignorenull Only for import Restricted to attributes of type <code>Collection</code> .	true / false Default: false	For Collection-type attribute values, this allows ignoring of null values. <ul style="list-style-type: none"> <li>If set to <code>false</code> as by default, a null value in a Collection-related column causes the value line contained the value <code>myValue;null</code>; the Collection to contain the null value, as in <code>myValue;null;myValue2</code>.</li> <li>If set to <code>true</code>, a null value in a Collection-related column causes the value line contained the value <code>myValue;null;myValue2</code>, to not contain the null value, as in <code>myValue;myValue2</code>.</li> </ul> Example: <code>INSERT_UPDATE myProduct;myAttribute[ignorenull=true]</code>
key2value-delimiter Restricted to attributes of type <code>Map</code> .	any character Default: "->"	Specifies the assignment delimiter for a key-value pair. Example: <code>INSERT_UPDATE myProduct;myAttribute[key2value-delimiter=&gt;]</code>
lang Restricted to localized attributes ( <code>Map&lt;String, Language&gt;</code> )	The ISO code or PK of a language defined for the SAP Commerce (such as <code>de</code> , <code>en</code> , or <code>de_ch</code> , for example) Default: session language	Specifies the language for a column containing localized attribute values. Example: <code>INSERT_UPDATE myProduct;myAttribute[lang=de]</code>
map-delimiter Restricted to attributes of type <code>Map</code> .	Any character Default: ";"	Specifies the delimiter between two key-value pairs. Example:  <code>INSERT_UPDATE myProduct;myAttribute[map-delimiter= ] ;myKey-&gt;myValue myKey2-&gt;myValue2;</code>
mode Import only Restricted to attributes of type <code>Collection</code>	append, remove, merge Default: replace (not explicitly specifiable) - an already existing Collection will be replaced	Specifies the modification mode for <code>Collection</code> instances. <ul style="list-style-type: none"> <li><code>mode=append</code>: In the <code>append</code> mode, references to elements of the Collection are updated. You can also use <code>(+)</code> instead of <code>append</code>.  <code>UPDATE Language;isoCode[unique=true];fallbackLang ;l1;(+)f1 ;l1;(+)f2</code></li> <li><code>mode=remove</code>: In the <code>remove</code> mode, references to elements of the Collection are removed. You can also use <code>(-)</code> instead of <code>remove</code>, for example:  <code>UPDATE Language;isoCode[unique=true];fallbackLang ;l1;(-)f1 ;l1;(-)f2</code></li> <li><code>mode=merge</code>: In the <code>merge</code> mode, references to elements of the Collection are merged. You can also use <code>(*)</code> instead of <code>merge</code>, for example:  <code>UPDATE Language;isoCode[unique=true];fallbackLang ;l1;(*)f1 ;l1;(*)f2</code></li> </ul>

Modifier	Allowed Values	Description
		<pre>UPDATE Language;isoCode[unique=true];fallbackL ;l1;(+?)f1 ;l1;(+?)f2</pre> <ul style="list-style-type: none"> <li>For example, in the following code snippet,</li> </ul> <ul style="list-style-type: none"> <li>The myCategory category's <b>supercategories</b> attribute contains mySuperCategory2 categories (<code>mode=merge</code>).</li> <li>In the case of <code>mode=remove</code>, the references to the mySuperCategory2 categories are removed.</li> <li>For the default (no <code>mode</code> modifier specified, implicit <code>mode=merge</code>), the myCategory category's <b>supercategories</b> attribute contains mySuperCategory2, overwriting all previous references.</li> </ul> <p>Example:</p> <pre>\$catalogVersion=catalogVersion(catalog(     \$supercategories=supercategories( code,c         INSERT_UPDATE category;code[unique=true]         ;myCategory;meine Kategorie;mySuperCategory         ;myCategory;meine Kategorie;mySuperCategory</pre>
numberformat Restricted to attributes of type Number	<p>A number format like <code>#.###,##</code> (For further information, refer to the <a href="#">NumberFormat class documentation by Sun Microsystems</a>)</p> <p>Default:</p> <pre>NumberFormat.getNumberInstance(     impexReader.getLocale() )</pre>	<p>Specifies the format the column uses to specify Number values.</p> <p>Be aware that the delimiters for number values depend on the locale selected by the <code>numberformat</code> modifier only. If you need to specify the delimiters explicitly, do so using the BeanShell code.</p> <p>Example:</p> <pre>#% impex.setLocale( Locale.GERMAN );</pre> <p>The following code snippet shows two settings for the <code>numberformat</code> modifier.</p> <p>Example:</p> <pre>INSERT_UPDATE myProduct;myAttribute[numberformat=#.# ;1.299,99 INSERT_UPDATE myProduct;myAttribute[numberformat=#,## ;1,299.99</pre>
path-delimiter Restricted to attribute of type ComposedType	<p>Any character</p> <p>Default: ":"</p>	<p>Defines the delimiter to separate values for attributes using an item separator.</p> <p>Example:</p> <pre>INSERT_UPDATE myProduct;myAttribute(code,id)[path-delimiter=:] ;myCode:myID</pre>
pos	<p>A positive integer</p> <p>Default: column number as positioned in header.</p>	<p>Specifies the column position where the values for this attribute can be found. If this attribute is defined in a header description, it has to be used for all attributes in the header.</p> <p>Example: [pos=3]</p>
translator	<p>An <code>AbstractValueTranslator</code> or a <code>SpecialValueTranslator</code></p> <p>Default varies; each Type has a predefined default Translator.</p>	<p>A <code>Translator</code> class resolves a column of a value line into an attribute value. It can be used to map instances of out-of-the-box <code>AtomicType</code> and <code>ComposedType</code> instances to attribute values using the default <code>Translator</code> classes.</p> <p>Example:</p>

Modifier	Allowed Values	Description
		<code>INSERT_UPDATE myProduct;myAttribute[translator=de.hy</code>
unique	true / false default: false	<p>Marks this column as a key attribute - that is, the value is used to determine values of all key attributes to check whether the item already exists. Two key attributes in a row must be unique. Each row must provide a unique value used to check whether or not an item already exists.</p> <p>Example:</p> <pre>INSERT_UPDATE myProduct;myAttribute[unique=true]</pre> <ul style="list-style-type: none"> <li>If this modifier is set to <b>false</b> as by default, an insert is always performed.</li> <li>If set to <b>true</b>, the column contains key attribute values and a update is performed. If the existing item will be updated by the non-unique values.</li> </ul>

## Value Line

Items are translated into a text representation called a value line completely driven by their business type definition. Generally, one item is represented by one value line - its attribute values by the columns of this row.

In a CSV file for the ImpEx extension, a value line holds a set of data such as a product definition with name, price and description, for example. This set of data spreads over several columns, each of which contains the value for an individual attribute. Within a row, columns are separated by a delimiter - the semicolon ; by default, for example:

```
Value1;Value2;Value3;Value4;Value5;Value6
```

For each value line, an item is created, updated, or removed depending on the mode set by the header. The basic structure of a value line is:

```
value;value;value;value;value
```

Instead of a value, you can also skip entries - as in the following code sample where two values are skipped. A skipped value is replaced by the default value if the currently active header defined one. If no default value is defined, then a **null** value is passed to the translator class. What happens then depends on the translator implementation.

```
value;;;value;value
```

The value `<ignore>` makes the ImpEx extension skip the entry and leave the item value as it is. This is useful in combination with the UPDATE mode, for example. The resolving mechanism also makes use of this value. For more information about the resolving mechanism see the *Resolving Mechanism* section of [ImpEx API](#).

```
value;<ignore>;<ignore>;value;value
```

## Comment

Comments start with the dash # and are ignored during import. Blank lines and lines without values, such as ones occasionally generated by Microsoft Excel like ;;;;;;;, are also ignored.

```
INSERT_UPDATE Product;code[unique=true];varianttype(code);name[lang=en];
;;;;;;;;;;
#;;;;;;;;;;
#BASEPRODUCTS JEANS;;;;;;
#;;;;;
```

## Definition Line - Macro

The ImpEx extension allows you to define macros so that you do not have to type repeating strings into your CSV files, and you can keep your CSV files more manageable.

During import, these macros are parsed and any occurrence of the macro name is replaced by the macro value. You can call macros in headers and in value lines. You can even call macros within macro definitions. If you define two macros with the same key, the latest definition is used.

Macro definitions start with the dollar symbol (\$), and they are referenced by \$macroName. For example:

```
$catalog=catalog(id)
$catalogVersion=catalogVersion($catalog,version)

INSERT Product;code;$catalogVersion
```

Wherever the term \$catalogVersion appears, it is replaced by: catalogVersion(catalog(id),version):

```
INSERT Product;code;catalogVersion(catalog(id),version)
```

## Abbreviations

Although the ImpEx header definition language provides the most flexible way of using custom column translators, header declaration can be rather long.

You could shorten them a bit using the ImpEx alias syntax **\$xyz=...**. You can also use regexp patterns and replacements to shorten frequently-used definitions. The following example shows how to use them to shorten classification columns declarations by giving them a new syntax. Check if the syntax shown in the example below is in your SAP Commerce local.properties file:

```
impex.header.replacement.1 =
    C@(\w+) ...
    @$1[ system='\\$systemName', version='\\$systemVersion'
        translator='de...ClassificationAttributeTranslator']
```

### i Note

The line has been wrapped to make it more readable. You must leave it in one line. Also note that the Java string notation has to be used, that is why there are double '\'.s.

All parameters starting with impex.header.replacement are parsed as ImpEx column replacement rules. The parameter has to end with a number that defines the priority of the rule. This way ambiguous rules could be sorted.

So what is this for? The first part of the property C@(\w+) defines the new abbreviation pattern to be used to declare classification attribute columns. The second part is the replacement text including the attribute qualifier match group \$1. In fact, it contains the original special column declaration. Both parts are to be separated by '...'.

So all that is needed now to declare a classification attribute column is this:

```
$systemName=MySys
$systemVersion=1.0
INSERT_UPDATE Product; code[unique=true]; C@attr1; C@attr2
```

Now the whole translator definition is hidden and you do not need to declare any helper alias for that. Nevertheless both alias definitions are mandatory to tell the classification column which system and version it should take its attribute from.

## Working with ImpEx

You can import and export various types of data to and from SAP Commerce. Learn how to manage any type of data, from catalogues or products, to user data or classification systems.

### [Creating Media Items using ImpEx](#)

Learn how to create an ImpEx cron job that uses a CSV file and a ZIP file to import media by following a simple example.

### [Exporting CMS Content](#)

This example shows you how to use an export script to export CMS content.

### [Exporting the Content of a Catalog Version](#)

This section shows you how to create a script to export the content of a catalog version.

### [Exporting the Content of a Classification System](#)

Describes how to export the content of a classification system using ImpEx.

### [Exporting Users and Addresses](#)

Where this document talks of **export**, it means the process of writing representations of SAP Commerce items into a CSV file. Where this document talks of **import**, it means the process of creating SAP Commerce items from the representations in a CSV file.

### [Importing Data from Multiple Sources](#)

The ImpEx import module can manage the loading of initial or other data from multiple data sources. Such data sources can have various types, formats or standards.

### [Importing Products and Orders](#)

This section gives you a sample file on how to use the ImpEx framework including definitions for common object creations.

### [Importing Users and Usergroups](#)

This section describes how to import users and usergroups using ImpEx.

### [Clearing Model Context](#)

Clearing the model context after every INSERT, INSERT\_UPDATE, UPDATE, and REMOVE ImpEx operation ensures that fresh and valid data is passed on to Interceptors' context.

## Creating Media Items using ImpEx

Learn how to create an ImpEx cron job that uses a CSV file and a ZIP file to import media by following a simple example.

### Code Sample

The following code sample shows how to create an ImpEx cron job that uses a CSV file and a ZIP file to import media. The CSV file (referenced as `inCsvName`) contains the ImpEx-related data such as headers. The ZIP file (referenced as `inMediaName`) contains the media files.

```
/**
 * With this method it's possible to import data and media at the same time e.g. during import.
 * To do so, it's necessary to place the data in a csv file and all desired media in a zip.
 * to pass a reference to these files to the impExCreateMedia(...) method in form of the
 * inCsvName and inMediaName.
 *
 * We cannot use the original files for the CronJob, so we need to create a copy of each one.
 */
public void impexCreateMedia( String inCsvName, String inMediaName, String outCsvName, String outMediaName )
{
    // open a new FileInputStream for the CSV to be imported
    InputStream inCSV = new FileInputStream( inCsvName );

    // by using the method createImpExMedia(...), the ImpexManager creates a new ImpEx Media
    ImpExMedia csvMedia = ImpExManager.getInstance().createImpExMedia( outCsvName );

    // open a CSV file for output
    csvMedia.setDataFromStream( new DataInputStream( inCSV ) );

    // set the MIME type for the csvMedia as defined by the project.properties file
    csvMedia.setMime( Config.getParameter( "mediatype.by.fileextension.csv" ) );

    // open a new FileInputStream to import the ZIP with the images
    InputStream inZIP = new FileInputStream( inMediaName );

    // Create a new ImpExMedia to hold the images
    ImpExMedia zipMedia = ImpExManager.getInstance().createImpExMedia( outMediaName );

    // set the ZIP stream, realfilename and mime type
    zipMedia.setData( new DataInputStream( inZIP ), outMediaName, Config.getParameter( "mediatype.zip" ) );

    // Use the ImpExManager to create a new ImpExImportCronJob
    ImpExImportCronJob c = ImpExManager.getInstance().createDefaultImpExImportCronJob();

    // assign csvMedia to the cronjob
    c.setJobMedia( csvMedia );

    // assign the media ZIP to the cronjob
    c.setMediasMedia( zipMedia );

    // after assigning the output medias to the cronjob, start the import
    ImpExManager.getInstance().importData( c, true, true );
}
```

For more information on cron jobs, see [The Cronjob Service](#).

## Calling the Code Sample

Specify the full path to the files in both the `inCsvName` and the `inMediaName` parameters. A sample call might look like the following:

```
impexCreateMedia("C:\\\\import.csv", "C:\\\\import.zip", "output.csv", "output.zip");
```

## Exporting CMS Content

This example shows you how to use an export script to export CMS content.

## Exporting CMS Content

```
insert_update Website;code[unique=true,allownull=true];emailFrom;emailPassword;emailServer;emailUser;
"##% impex.exportItems( ""Website"" , false );"

insert_update Template;code[unique=true,allownull=true];webSite(code)[unique=true,all];
"##% impex.exportItems( ""Template"" , false );"
```

```

insert_update ParagraphContent;code[unique=true];webSite(code)[unique=true,allownull=false];
"%# impex.exportItems( ""ParagraphContent"" , false );"

insert_update NavigationElement;code[unique=true];content(code,website(code));content
"%# impex.exportItems( ""NavigationElement"" , false );"

insert_update BannerNavigationElement;bannerHTMLSrc[lang=nl];bannerMedia(code,catalog
"%# impex.exportItems( ""BannerNavigationElement"" , false );"

insert_update TextParagraph;code[unique=true];paragraphText[lang=nl];status(itemtype)
"%# impex.exportItems( ""TextParagraph"" , false );"

insert_update ParagraphContent2ParagraphRelation;language(isocode)[unique=true];quali
"%# impex.exportItems( ""ParagraphContent2ParagraphRelation"" , false );"

insert_update Store;name[lang=nl];code[allownull=true,unique=true];online[allownull=1
"%# impex.exportItems( ""Store"" , false );"

```

## Exporting the Content of a Catalog Version

This section shows you how to create a script to export the content of a catalog version.

The script was tested using the **clothescatalog** and **hwcatalog** catalog in version **Staged** created by the sample data of the storefoundation. It exports all content of one catalog version of the storefoundation including categories, products and related version specific types. It does not exports non-version specific types as languages, users, currencies, classification system, etc.

The catalog version can be configured at the start of the script using the two macros:

```
$catalog=clothescatalog
$version=Staged
```

The value of the **\$catalog** macro sets the catalog id. The value of the **\$version** macro sets the version string of the catalog version to export. These two macros will be used throughout the whole script for gathering the items for export.

Basically, the script consists of two components: headers and FlexibleSearch-based export statements.

This document is structured according to the export script:

- the [catalog structure](#) with its categories is exported in case the version does not exist at the destination platform
- The [products](#) and their variants as well as related medias are exported
- The related price, discount and tax [rows](#) are also exported as well as
- [Remaining types](#) like keywords, references, agreements and syncitemjobs.
- Finally, all [relation](#) items are exported.

## Category Structure

```
insert_update Category; catalog(id); catalogVersion(catalog(id),version)[unique=true]; code[unique=true];
"%# impex.exportItems(""SELECT {Cat:pk} FROM {Category as Cat}, {CatalogVersion as CV}, {Catalog

```

## Products and Medias

### Products

The main content of a catalog version are the products. In general, a product is identified by its code and catalog version and can be exported using the following lines:

```
insert_update Product; catalogVersion(catalog(id),version)[unique=true]; code[unique=true]; Europe1P!  
    "%> impex.exportItems("SELECT {P:pk} FROM {Product as P}, {CatalogVersion as CV}, {Catalog as C}
```

Furthermore all relations where products are involved have to be exported:

- Product2KeywordRelation,
  - ProductMediaLink
  - CategoryProductRelation

(See [relations](#) section). All PriceRows, DiscountRows and TaxRows are [exported separately](#).

The previous code snippet is the most simple export which exports all products including all subtypes using one single header. Using this way the additional attributes of subtypes of **Product** will be lost especially the type itself.

Therefore, it is better to write a export for each individual type (**Product**, **SweatShirts**, **Jeans**, **Shoes**, in this case):

```
insert_update Product; catalogVersion(catalog(id),version)[unique=true]; code[unique=true]; Europe1P1
  "%> impex.exportItems("//SELECT {P:pk} FROM {Product! as P}, {CatalogVersion as CV}, {Catalog as C} as CatalogVersion

insert_update SweatShirts; catalogVersion(catalog(id),version)[unique=true]; code[unique=true];
  "%> impex.exportItems("//SELECT {P:pk} FROM {SweatShirts as P}, {CatalogVersion as CV}, {Catalog as C} as CatalogVersion

insert_update Jeans; catalogVersion(catalog(id),version)[unique=true]; code[unique=true]; Europe1P2
  "%> impex.exportItems("//SELECT {P:pk} FROM {Jeans as P}, {CatalogVersion as CV}, {Catalog as C} as CatalogVersion

insert_update Shoes; catalogVersion(catalog(id),version)[unique=true]; code[unique=true]; Europe1P3
  "%> impex.exportItems("//SELECT {P:pk} FROM {Shoes as P}, {CatalogVersion as CV}, {Catalog as C} as CatalogVersion
```

Here all items of type Product excluding subtypes (marked with the ! sign behind the Product type at the FlexibleSearch statement, **Product!**) and the three variant types are exported individually using a separate header and a separate FlexibleSearch statement each. This allows exporting values for customized types with individual attributes.

Media

The next item which is related closely to products is the Media type which is in general catalog version-dependent. Media not contained at the used catalog versions will be not exported and are assumed as existing in the target platform! A media is identified via its code and catalog version. The data of the medias is exported to a additional ZIP-file by using the **MediaDataTranslator**. The necessary relations **CategoryMediaRelation** and **ProductMediaLink** are exported at the [relation section](#).

```
insert_update Media;catalogVersion(catalog(id),version)[unique=true];code[unique=true];@media[transla  
    "%> impex.exportItems(""SELECT {M:pk} FROM {Media as M}, {CatalogVersion as CV}, {Catalog as C}
```

## Price, Tax, and Discount Rows

For a export of a product, the associated rows are essential. So all rows related to products of the used catalog version are selected as well as rows related to the used catalog version. As such rows do not have a unique identifier, rows will be imported with a INSERT statement always because of the missing unique modifier. Updating existing rows is not possible as the rows cannot be accessed individually due to the missing unique identifier and therefore cannot be updated either.

```

insert DiscountRow; catalogVersion(catalog(id),version); currency(isocode); dateRange; discount(code)
  "#% impex.exportItems("""SELECT {DR:pk} FROM {DiscountRow as DR}, {Product as P}, {CatalogVersion as CV}""");
  "#% impex.exportItems("""SELECT {DR:pk} FROM {DiscountRow as DR}, {CatalogVersion as CV}, {Catalog as C}""");

insert PriceRow; catalogVersion(catalog(id),version); currency(isocode)[allownull=true]; dateRange
  "#% impex.exportItems("""SELECT {PR:pk} FROM {PriceRow as PR}, {Product as P}, {CatalogVersion as CV}""");
  "#% impex.exportItems("""SELECT {PR:pk} FROM {PriceRow as PR}, {CatalogVersion as CV}, {Catalog as C}""");

insert TaxRow:catalogVersion(catalog(id).version):currency(isocode):dateRange:endTime:pg(itemtype)

```

```
"#% impex.exportItems("""SELECT {TR:pk} FROM {TaxRow as TR}, {Product as P}, {CatalogVersion as CV}""")"
"#% impex.exportItems("""SELECT {TR:pk} FROM {TaxRow as TR}, {CatalogVersion as CV}, {Catalog as C}""")"
```

## Miscellaneous Types

Another type which has to be exported is Keyword where items of this types are unique using catalog version, language and keyword as identifier. The product association is exported by the Product2KeywordRelation at [relation section](#).

```
insert_update Keyword;catalogVersion(catalog(id),version)[unique=true];keyword[unique=true];language[unique=true]
"#% impex.exportItems("""SELECT {K:pk} FROM {Keyword as K}, {CatalogVersion as CV}, {Catalog as C}""")"
```

Agreements are also catalog version dependent and are identified by its catalog version and id.

```
insert_update Agreement;catalogVersion(catalog(id),version)[unique=true];id[unique=true];Catalog(id)
"#% impex.exportItems("""SELECT {A:pk} FROM {Agreement as A}, {CatalogVersion as CV}, {Catalog as C}""")"
```

The **ProductFeature** type is identified by its product and its classification attribute assignment where it is assumed that the classification is defined in a foreign catalog version and already exists at the target system.

The value attribute of the **ProductFeature** is of type Object, so the **AtomicValueTranslator** is used as default which translate it to/from a base64 serialized string. In case of a migration that is OK in case you have no values of type enumeration. In all other cases this is not suitable and a additional translator is needed which handles the type of the value.

```
insert_update ProductFeature; product(code,catalogVersion(catalog(id),version))[unique=true]; classification[unique=true]
"#% impex.exportItems("""SELECT {PF:pk} FROM {ProductFeature as PF}, {Product as P}, {CatalogVersion as CV}""")"
```

Optionally the already defined synchronization jobs can be exported, so you do not have to configure them again at the target system. An import of them will only be successful if the related catalog version still exists. Here the clothescatalog in staged version is exported. The configured job describes a synchronization with the related online version, so the online version has to exist at target system!

```
insert_update SyncItemJob; code[unique=true]; sourceVersion(catalog(id),version)[unique=true]; targetVersion[unique=true]
"#% impex.exportItems("""SELECT {SIJ:pk} FROM {SyncItemJob as SIJ}, {CatalogVersion as CV}, {CatalogVersion as TV}""")"
```

```
insert_update SyncAttributeDescriptorConfig;attributeDescriptor(enclosingType(code),qualifier)[unique=true]
"#% impex.exportItems("""SELECT {SADC:PK} FROM {SyncAttributeDescriptorConfig as SADC}, {SyncItemJob as SIJ}""")"
```

At last type described here, the **ProductReference**'s are exported. They have no unique identifier because a product can have a reference to another product twice. So here only the INSERT mode can be used for import! Furthermore it has to be considered that a referenced product can be content of a foreign catalog version, so it is assumed that it exists already at the target system.

```
insert ProductReference; active[allownull=true]; description[lang=de]; description[lang=en]; icon(code)
"#% impex.exportItems("""SELECT {PR:pk} FROM {ProductReference as PR}, {Product as P}, {CatalogVersion as CV}""")"
```

## Relations

The hybris Platform basically has two types of relations: 1:1 and 1:n on the one hand and n:m on the other hand.

For export purposes, only n:m relation have to be considered: A relation always is a subtype of Link and therefore has a source and target attribute and a language that has to be considered where a link item is always identified by its source and target attribute. Therefore, the data is stored in the Link table or using an own deployment.

In all other cases it is modeled by using simple references (1:1) or a reference and a jalo collection (1:n).

It is always preferred to export relations using the real link instance and not by exporting the jalo collection attributes at the involved items. The relations that has to be considered at this script are the followings:

```

insert_update CategoryProductRelation;source(code,catalogVersion(catalog(id),version))[unique=true];{
  "#% impex.exportItems("""SELECT {CPR:pk} FROM {CategoryProductRelation as CPR}, {Product as P}, {Catalog
  insert_update ProductMediaLink;source(code,catalogVersion(catalog(id),version))[unique=true];target(
    "#% impex.exportItems("""SELECT {PML:pk} FROM {ProductMediaLink as PML}, {Media as M}, {CatalogVe
  insert_update CategoryMediaRelation;source(code,catalogVersion(catalog(id),version))[unique=true];
    "#% impex.exportItems("""SELECT {CMR:pk} FROM {CategoryMediaRelation as CMR}, {Media as M}, {Catal
  insert_update CategoryCategoryRelation;source(code,catalogVersion(catalog(id),version))[unique=true];
    "#% impex.exportItems("""SELECT {CCR:pk} FROM {CategoryCategoryRelation as CCR}, {Category as Cat}
  insert_update Product2KeywordRelation;source(keyword,catalogVersion(catalog(id),version),language);
    "#% impex.exportItems("""SELECT {PKR:pk} FROM {Product2KeywordRelation as PKR}, {Product as P}, {K
  insert_update Category2KeywordRelation;source(code,catalogVersion(catalog(id),version))[unique=true];
    "#% impex.exportItems("""SELECT {CKR:pk} FROM {Category2KeywordRelation as CKR}, {Category as Cat}
  insert_update Category2PrincipalRelation;source(code,catalogVersion(catalog(id),version))[unique=true];
    "#% impex.exportItems("""SELECT {CPR:pk} FROM {Category2PrincipalRelation as CPR}, {Category as Ca
  insert_update Principal2ReadableCatalogVersionRelation;source(uid)[unique=true];target(catalog(id));
    "#% impex.exportItems("""SELECT {PCR:pk} FROM {Principal2ReadableCatalogVersionRelation as PCR}, {C
  insert_update Principal2WriteableCatalogVersionRelation;source(uid)[unique=true];target(catalog(id));
    "#% impex.exportItems("""SELECT {PCR:pk} FROM {Principal2WriteableCatalogVersionRelation as PCR},

```

## Exporting the Content of a Classification System

Describes how to export the content of a classification system using ImpEx.

```

"#% impex.setLocale( new Locale( ""en"" , """" ) );"

$classificationSystem=SampleClassification
$version=1.0

insert_update ClassificationSystem;&Item;activeCatalogVersion(catalog(id),version);buyer(uid);defaultCatalogVersion(catalog(id),version);
  "#% impex.exportItems("""SELECT {CS:pk} FROM {ClassificationSystem as CS} WHERE {CS:id}='\$classificationSystem'""");

insert_update ClassificationSystemVersion;&Item;active[allownull=true];catalog(id)[unique=true,forceWrite=true];
  "#% impex.exportItems("""SELECT {CSV:pk} FROM {ClassificationSystemVersion as CSV},{ClassificationSystemVersion as CSV} WHERE {CSV:classificationSystem}='\$classificationSystem'""");

insert_update ClassificationClass;&Item;catalog(id)[allownull=true];catalogVersion(catalog(id),version);
  "#% impex.exportItems("""SELECT {CC:pk} FROM {ClassificationClass as CC}, {ClassificationSystemVersion as CSV} WHERE {CC:classificationSystem}='\$classificationSystem'""");

insert_update ClassAttributeAssignment;&Item;attributeType(itemtype(code),code)[unique=true];classificationAttribute(code);
  "#% impex.exportItems("""SELECT {CAA:pk} FROM {ClassAttributeAssignment as CAA}, {ClassificationSystemVersion as CSV} WHERE {CAA:classificationSystem}='\$classificationSystem'""");

insert_update AttributeValueAssignment;&Item;attribute(&Item);attributeAssignment(classificationAttribute(code));
  "#% impex.exportItems("""SELECT {AVA:pk} FROM {AttributeValueAssignment as AVA}, {ClassificationSystemVersion as CSV} WHERE {AVA:classificationSystem}='\$classificationSystem'""");

insert_update ClassificationAttributeUnit;&Item;code[allownull=true,unique=true];conversionFactor;external;
  "#% impex.exportItems("""SELECT {CAU:pk} FROM {ClassificationAttributeUnit as CAU}, {ClassificationSystemVersion as CSV} WHERE {CAU:classificationSystem}='\$classificationSystem'""");

insert_update ClassificationAttribute;&Item;code[forceWrite=true,allownull=true,unique=true];external;
  "#% impex.exportItems("""SELECT {CA:pk} FROM {ClassificationAttribute as CA}, {ClassificationSystemVersion as CSV} WHERE {CA:classificationSystem}='\$classificationSystem'""");

insert_update ClassificationAttributeValue;&Item;code[forceWrite=true,allownull=true,unique=true];
  "#% impex.exportItems("""SELECT {CAV:pk} FROM {ClassificationAttributeValue as CAV}, {ClassificationSystemVersion as CSV} WHERE {CAV:classificationSystem}='\$classificationSystem'""");

insert_update ClassificationKeyword;&Item;catalog(id)[allownull=true,unique=true];catalogVersion(catalog(id),version);
  "#% impex.exportItems("""SELECT {CK:pk} FROM {ClassificationKeyword as CK}, {ClassificationSystemVersion as CSV} WHERE {CK:classificationSystem}='\$classificationSystem'""");

insert_update ClassificationAttributeTypeEnum;code[unique=true];extensionName;name[lang=de];name[lang=en];
  "#% impex.exportItems( ""ClassificationAttributeTypeEnum"" , false );"

insert_update ClassificationAttributeVisibilityEnum;code[unique=true];extensionName;name[lang=de];name[lang=en];

```

```

"%# impex.exportItems( ""ClassificationAttributeVisibilityEnum"" , false );"

insert_update Category2KeywordRelation;&Item;language(isocode)[unique=true];qualifier;source(catalog\
"%# impex.exportItems(""SELECT {CKR:pk} FROM {Category2KeywordRelation as CKR}, {ClassificationKeywo\
insert_update Product2KeywordRelation;&Item;language(isocode)[unique=true];qualifier;source(catalogVe\
"%# impex.exportItems(""SELECT {PKR:pk} FROM {Product2KeywordRelation as PKR}, {ClassificationKeywo\
insert_update CategoryProductRelation;source(code,catalogVersion(catalog(id),version))[unique=true];i\
"%# impex.exportItems(""SELECT {CPR:pk} FROM {CategoryProductRelation as CPR}, {ClassificationClass a\
insert_update CategoryCategoryRelation;source(code,catalogVersion(catalog(id),version))[unique=true];\
"%# impex.exportItems(""SELECT {CCR:pk} FROM {CategoryCategoryRelation as CCR}, {ClassificationClass\
insert_update Category2PrincipalRelation;source(code,catalogVersion(catalog(id),version))[unique=true];\
"%# impex.exportItems(""SELECT {CPR:pk} FROM {Category2PrincipalRelation as CPR}, {ClassificationClas\
insert_update Principal2ReadableCatalogVersionRelation;source(uid)[unique=true];target(catalog(id),ve\
"%# impex.exportItems(""SELECT {PCR:pk} FROM {Principal2ReadableCatalogVersionRelation as PCR}, {Clas\
insert_update Principal2WriteableCatalogVersionRelation;source(uid)[unique=true];target(catalog(id),v\
"%# impex.exportItems(""SELECT {PCR:pk} FROM {Principal2WriteableCatalogVersionRelation as PCR}, {Clas

```

## Exporting Users and Addresses

Where this document talks of **export**, it means the process of writing representations of SAP Commerce items into a CSV file. Where this document talks of **import**, it means the process of creating SAP Commerce items from the representations in a CSV file.

See these topics for details:

- [Creating Headers for Export](#)
- [Modifying Export Headers](#)
- [Exporting Data](#)
- [Modifying the Import Header to Use DocumentIDs](#)
- [Importing Data](#)

## Creating Headers for Export

Follow these steps to create headers for export.

### Procedure

1. Log in to Backoffice.
2. Navigate to **System > Tools > Script Generator**.
3. Clear the **Document ID** check box.
4. Select **Migration** from the **Script type** drop-down menu.
5. Click the **Generate** button.

The Script Generator writes the headers for all types in SAP Commerce into the **Script** field.

6. Copy and paste the list of headers into a text editor.
7. Remove the headers for the types you do not want to export.

The following code snippet shows a sample header (for the **User** type):

```
# ---- Extension: core ---- Type: User ----
"#% impex.setTargetFile( ""User.csv"" );"
```

```
insert_update User;pk;CN;DN;Europe1PriceFactory_UDG(code,itemtype(code));Europe1PriceFactory_UP
"%# impex.exportItems( ""User"" , false );"
```

## Modifying Export Headers

There are two aspects which you may need to edit about the generated headers:

1. The headers as generated contain references to the PK of the items in SAP Commerce (such as **addresses(pk)** or **allDocuments(pk)**, for example). It is very much impossible for any two SAP Commerce installations to have identical PKs for any two items, let alone for all of the items. Therefore, referencing items by PK is not going to work during migration. By consequence, a different way of referencing items than by PK will be necessary, such as via the identifier (code) or a combination of any other attributes.

Also note that items to be referenced by non-PK attributes (such as **isocode**, for example) will have to exist already in the new system.

2. By default, a generated ImpEx header does not export items which are subtypes of the header type. This is specified by the Boolean parameter which is set to **false** by default. To also export items which are subtypes of the header type, set the Boolean parameter from **false** to **true**, such as:

```
"#% impex.exportItems( ""User"" , true );"
```

## Exporting Users

An edited sample header for the **User** type could look like this:

```
# ---- Extension: core ---- Type: User ----
"#% impex.setTargetFile( ""User.csv"" , true );"
insert_update User; pk[unique=true]; uid[unique=true]; addresses(pk); sessionLanguage(isocode); desc
"#% impex.exportItems( ""User"" , true );"
```

## Exporting Addresses

An edited sample header for the **Address** type could look like the following code snippet. The **Address** type is somewhat different from the **User** type for which it is enough to simply export any item.

Addresses can be referenced by two different types: **User** (different residences, for example) and **Order** (where was a certain order delivered to?). On every order, SAP Commerce creates a copy of the address the order was delivered to. However, we are not interested in exporting order-related addresses and only want to export addresses referenced by a user. Therefore, we need to sort out all order-related addresses.

An Address has a field called **duplicate** which indicates a user-related address if set to **false** and indicates an order-related address if set to **true**. To export only user-related addresses, we need to select only addresses with the **duplicate** attribute set to **false**. The following code snippet uses a FlexibleSearch statement to select only addresses with the **duplicate** attribute set to **false**.

```
# ---- Extension: core ---- Type: Address ----
"#% impex.setTargetFile( ""Address.csv"" , true );"
insert_update Address; pk[unique=true]; country(isocode); duplicate; email; firstname; gender(itemtyp
"#% impex.exportItems( ""select {pk}from {Address} where {duplicate}='false'"", Collections.EMPTY_MAP );"
```

## Exporting Data

Follow these steps to export data.

## Procedure

1. Copy and paste the edited headers from the text editor into the **Script** field of **Wizard: Script Generator**.

2. Click the **Save** button.

The **Wizard: Script Generator** creates a Media containing the content of the **Script** field. This media contains the headers as specified in the **Script** field.

The name of the Media will be displayed in the **Media** field, such as **exportscript\_1208437500254**.

3. Close the **Wizard: Script Generator**.

4. In the Backoffice navigation tree, click the **Export** link in the **System > Tools** menu.

5. Select the media created via the **Wizard: Script Generator** in the **ImpEx content** field. The field has auto-completion - just start typing in a part of the name of the media.

The content of the Media is displayed in the **Script** field.

6. Select **(Re)Import Relaxed** from the **Mode** drop-down menu.

7. Click the **Validate** button.

8. Click the **Next** button and the **Start** button.

The Export Wizard starts a CronJob and creates a ZIP file in the progress (**dataexport\_0000000Z-ImpEx-Export.zip**, for example) that contains an **.impex** file containing the headers plus one CSV file for every exported type each.

For example, the **dataexport\_0000000Z-ImpEx-Export** ZIP file contains these three files:

- **Address.csv**
- **importscript.impex**
- **User.csv**

This ZIP file contains the CSV representations of every type to export, plus the header files.

## Modifying the Import Header to Use DocumentIDs

Follow these steps to modify the import header to use DocumentIDs.

### Context

#### **i Note**

This section only applies to imports using the PK as a reference. If you do not have any items to be imported that use the PK as a reference, you can skip this section.

Types like **User** that have a unique identifier (such as the **code** attribute) are rather simple to handle as referencing the identifier will do for migration purposes (both for export and for import). Things are different with types that do not have a unique identifier on top of the PK. To export items without an identifier on top of the PK, you will need to resolve the item's PK and use the PK as a reference during import. For such types, you have to leave the references to the PK in the export headers. For more in-depth details on Document IDs, please refer to [Document ID](#).

### Procedure

1. Once the Wizard has completed, click the **Download** button.

2. Save the file on your system's local hard disk.

3. Unpack the ZIP file.

4. Open the **impexscript.impex** file.

5. Replace the reference to the PK (such as **addresses(pk)**) with DocumentIDs (name with a prefixed &, such as **&pk**, for example), such as:

```
insert_update Address; &pk; country(isocode)[unique=true]; ... ;owner(&pk)[unique=true] "#% imp
```

6. Create a new ZIP file containing the edited **impexscript.impex** file and the CSV files.

7. Make sure the new ZIP file is stored on a volume you can reach from the system to which to import (CD, DVD, network share, USB storage device).

## Importing Data

Follow these steps to import data using Backoffice.

### Procedure

1. In the Backoffice navigation tree, click the **Import** link in the **System > Tools** section.

2. Select the ZIP file containing the CSV files and the header file.

If you did not edit the **impexscript.impex** file, you only have to reference the ZIP file via the **Choose media** field. The field has auto-completion - just start typing in a part of the name of the ZIP file.

If you edited the **impexscript.impex** file, you will need to upload the new ZIP file into an ImpEx Media:

- a. Click the **Upload** button under **Upload new impex**.
- b. Use the file system explorer to navigate to the new ZIP file.
- c. Click the **Create** button.

3. Make sure you are using the relaxed import mode. **Strict import mode** checkbox should be empty.

4. Mark the **Allow code execution from within the file** checkbox.

5. Click the **Start** button.

## Importing Data from Multiple Sources

The ImpEx import module can manage the loading of initial or other data from multiple data sources. Such data sources can have various types, formats or standards.

Although ImpEx handles data in CSV format by default, the approach given here shows how you can combine data from several sources, in different format. The example demonstrates how to prepare an import process that will load product data from the following sources:

- **SAP base product data:** TXT files with data formatted in some fixed order
- **SAP size data:** TXT files with data formatted in some fixed order
- **XLS file with product prices**
- **CSV file with marketing information**

The example has the following constraints:

- Marketing product data only updates those products that are imported from SAP files
- Because the quantity of products from the SAP source is larger than that from the CSV file, any products from SAP that are not updated during the marketing data import should be present in the system, but not in the online store

## Design

Make sure that all data source files are stored in one folder. Implement a class that extends the SAP Commerce internal job class. Create a special instance of CronJob for this job, and extend it with a property that will hold a path that points to this data sources folder. Using a cron job allows you to run an import process at any time from Backoffice. You can also pass the input path as a cron job parameter. Ensure your implementation of the class extending the job class does the following:

- Searches for files that match fixed filters in the input folder
- Triggers specific import modules and passes them the list of files to import
- Logs the progress of the import process

Divide the import process into the following independent modules:

- SAP base data
- CSV marketing data
- SAP size data
- XLS price data

Make sure that these modules are triggered by the main import process in the order that they are listed. Each module then parses each given source file from the list and creates a CSV file (or files) for it. After it creates the CSV files, import them to the system. Define a pack of fixed import headers for each module.

## Implementation Details

### Configurable Import Process

The example defines five boolean project properties. Four of them can enable or disable the import of a specific module during the import process. For instance, setting the property called `debug.import.marketingData` to false disables marketing data import. The fifth flag controls the deletion of temporary CSV files that get created during import. The functionality of those flags gives us the possibility to debug the whole import process if some problem should occur.

### XLS File Import

Use the `org.apache.poi.hssf.*` package. HSSF is the POI project's pure Java implementation of the Excel '97(-2007) file format. It does not support the new Excel 2007 .xlsx OOXML file format, which is not OLE2 based. For more information, see [click here](#).

Open the XLS file and read first row:

```
//InputStream in;

try {
    // try to open xls file
    HSSFWorkbook wb = new HSSFWorkbook(in);

} catch (IOException ioe) {
    log.error( ioe );
}

// get first sheet

HSSFSheet sheet = wb.getSheetAt(0);

// get first row

HSSFRow headerRow = sheet.getRow((short) 0);

// read first cell

HSSFCell firstCell = headerRow.getCell((short) (i));
```

The HSSFCell object can hold values that could be of the type:

- BooleanCellValue
- NumericCellValue
- DateCellValue
- RichStringCellValue

It is important to know what type of value you are trying to get from the cell. Calling the getter `getRichStringCellValue()` on `HSSFCell` that holds a `NumericCellValue` throws an Exception. Before calling the getter, check what type of cell value it holds:

```
// HSSFCell articleCell
if (articleCell.getCellType() == HSSFCell.CELL_TYPE_NUMERIC) {
    articleId = String.valueOf(articleCell.getNumericCellValue());
    articleId = articleId.substring(0, articleId.length() - 2);
} else {
    articleId = articleCell.getRichStringCellValue().getString();
}
```

## Selective Product Visibility

As the requirement is to **not** show those products that have no match in marketing data provided in the CSV file, use the SAP Commerce internal `approvalStatus` flag. The idea is easy:

Set status `nonapproved` for all of the products that are imported from SAP data as default. Set status `approved` for those which are updated during marketing data import.

## Advantages

- Each module produces ImpEx CSV files which can be treated as a log file containing what and how it was imported.
- ImpEx automatically logs all the things that went wrong, giving additional debug information.
- Use of `project.properties` enables you to control the import process and delete the produced CSV files, also doubling as a debug tool. However, you can only change those properties using the admin console.
- As cron jobs are placed in Backoffice by default, you can run the import process easily and at any moment. Backoffice allows you to look in the cron job logs more deeply.
- All of the import work is done by ImpEx instead of implementing it yourself. The only thing to do is to prepare proper ImpEx headers and implementation of logic that will parse and read source files.

## Importing Products and Orders

This section gives you a sample file on how to use the ImpEx framework including definitions for common object creations.

It is not intended to give you an overview or basic information on the ImpEx framework, which can be found in [ImpEx](#).

## The Sample File

```
#  
# Macro definitions (1)  
#  
$catalogVersion=catalogVersion(catalog(id[default='clothescatalog']), version[default='Staged'])[unic  
$prices=europelPrices[translator=de.hybris.platform.europe1.jalo.impex.EuropelPricesTranslator]  
$baseProduct=baseProduct(code, catalogVersion(catalog(id[default='clothescatalog']), version[default=
```

```

#
# Create a category (2)
#
INSERT_UPDATE Category;code[unique=true];$catalogVersion;name[lang=de];name[lang=en];description[lang=de];
;SampleCategory;clothescatalog:Online;Testkategorie;Sample category;Dies ist eine Testkategorie;This

#
# Create some products (3)
#
INSERT_UPDATE Product;code[unique=true];name[lang=en]; name[lang=de];unit(code);$catalogVersion; description[lang=de];
;sampleproduct1;SampleProduct1;Testprodukt1;pieces;clothescatalog:Online;"This is a sample product";
;sampleproduct2;SampleProduct2;Testprodukt2;pieces;clothescatalog:Online;"This is another sample product";
;sampleproduct3;SampleProduct3;Testprodukt3;pieces;clothescatalog:Online;"This is the third sample product"

#
# Some pricerows for our products (4)
#
INSERT_UPDATE Product;code[unique=true];$catalogVersion;$prices;Europe1PriceFactory_PTG(code);
;sampleproduct1;clothescatalog:Online;"1 pieces = 12,00 EUR, 20 pieces = 10,00 EUR";Tax_Full;
;sampleproduct2;clothescatalog:Online;"1 pieces = 11,50 EUR, 30 pieces = 0,99 EUR";Tax_Half;
;sampleproduct3;clothescatalog:Online;"1 pieces = 651,89 EUR, 10 pieces = 599,99 EUR";Tax_Full;

#
# Some variants of sampleproduct1 including Prices (5)
#
INSERT_UPDATE Product;code[unique=true]; $baseProduct;$catalogVersion; approvalStatus(code); Europe1PriceFactory_PTG(code);
;sampleproduct1-00;sampleproduct1;clothescatalog:Online;approved;Tax_Full;11,50 EUR N; pieces;
;sampleproduct1-01;sampleproduct1;clothescatalog:Online;approved;Tax_Full;11,00 EUR N; pieces;

#
# Defining an Order (6)
#
INSERT_UPDATE Order;code[unique=true];user(uid);date[dateformat=dd.MM.yyyy HH:mm];currency(isocode);region;
;SampleOrder1;demo;17.01.2006 10:58;EUR;false;;advance;;false
## impex.getLastImportedItem().setDeliveryAddress(impex.getLastImportedItem().getUser().getDefaultDeliveryAddress());
## impex.getLastImportedItem().setPaymentAddress(impex.getLastImportedItem().getUser().getDefaultPaymentAddress());

#
# Setting OrderEntries on the Order (7)
#
INSERT_UPDATE OrderEntry;order(code)[unique=true];product(code,catalogVersion(catalog(id),version))[1];
;SampleOrder1;sampleproduct1;clothescatalog:Online;1;false;pieces;0;;
;SampleOrder1;sampleproduct2;clothescatalog:Online;2;false;pieces;1;;
;SampleOrder1;sampleproduct3;clothescatalog:Online;3;false;pieces;2;;

#
# Calculating the Order (8)
#
UPDATE Order;code[unique=true]
;SampleOrder1
## impex.getLastImportedItem().recalculate();

```

## Macro Definitions (1)

The first section of the file defines two macros referred to throughout the object definitions:

- **\$catalogversion**

Sets the default catalog to be **clothescatalog:Staged**, in total, and in parts (**default='clothescatalog'** and **default='Staged'**, respectively). The following table visualizes what will happen if you give a value for the catalogVersion attribute using the above definition line and setting some values empty (n/a):

ID parameter	version parameter	Resulting catalog version
MyCatalog	Online	MyCatalog:Online
n/a	Online	clothescatalog:Online
MyCatalog	n/a	MyCatalog:Staged

ID parameter	version parameter	Resulting catalog version
n/a	n/a	clothescatalog:Staged

- **\$prices**

Defines the translator class for specified prices to be `de.hybris.platform.europe1.jalo.impex.Europe1PricesTranslator`.

```
#  
# Macro definitions (1)  
#  
$catalogVersion=catalogVersion(catalog(id[default='clothescatalog']),version[default='Staged'])[unique]  
$prices=europe1Prices[translator=de.hybris.platform.europe1.jalo.impex.Europe1PricesTranslator]
```

## Create a Category (2)

This section creates a sample Category instance **SampleCategory** within the **Online** version of the **clothescatalog** catalog. Note the empty value for the **supercategories** attribute. A Category instance with no value for the **supercategories** attribute is created as a root category.

```
#  
# Create a category (2)  
#  
INSERT_UPDATE Category;code[unique=true];$catalogVersion;name[lang=de];name[lang=en];description[lang  
;SampleCategory;clothescatalog:Online;Testkategorie;Sample category;Dies ist eine Testkategorie;This
```

## Creating Some Products (3)

As creating Product instances is a common field of use for the ImpEx framework, the following code snippet creates three sample Product instances: **sampleproduct1**, **sampleproduct2**, and **sampleproduct3**. All of these sample Product instances are part of the **clothescatalog** catalog in the **Online** version, their **unit** is **pieces**, they belong into the **SampleCategory** category and their **approval** status is set to **approved**.

```
#  
# Creating some products (3)  
#  
INSERT_UPDATE Product;code[unique=true];name[lang=en];name[lang=de];unit(code);$catalogVersion;descri  
;sampleproduct1;SampleProduct1;Testprodukt1;pieces;clothescatalog:Online;"This is a sample product";'  
;sampleproduct2;SampleProduct2;Testprodukt2;pieces;clothescatalog:Online;"This is another sample prod  
;sampleproduct3;SampleProduct3;Testprodukt3;pieces;clothescatalog:Online;"This is the third sample pr
```

Note the **(code)** parameter in the header on the **unit**, **approvalStatus**, and **supercategories** attributes (**unit(code)**, **approvalStatus(code)**, and **supercategories(code)**, respectively). By default, the ImpEx framework expects references to be given as a PK. By specifying the **(code)** parameter, the ImpEx framework resolves the given value from an identifier. If you specify an identifier where the ImpEx framework expects a PK, the import will fail with the following exception:

```
ERROR (master) [ImpExImportJob] pk has wrong format: 'SampleCategory':For input string: "SampleCategory"  
de.hybris.platform.core.PK$PKException: pk has wrong format: 'SampleCategory':For input string: "Sampl  
at de.hybris.platform.core.PK.parse(PK.java:294)  
at de.hybris.platform.impex.jalo.translators.ItemPKTranslator.convertToJalo(ItemPKTranslator.  
at de.hybris.platform.impex.jalo.translators.SingleValueTranslator.importValue(SingleValueTranslat  
at de.hybris.platform.impex.jalo.translators.CollectionValueTranslator.processItem(CollectionValueT  
...  
...
```

## Some Pricerows for Our Products (4)

The SAP Commerce default PriceFactory, Europe1, calculates prices based on PriceRow instances. A PriceRow instance defines a price for a product in a certain amount of units, depending on the user and the date. The following code snippet defines two PriceRow instances for **sampleproduct1**, **sampleproduct2**, and **sampleproduct3** each:

Product	Number of units	Effective price per unit
sampleproduct1	1 through 19	12 EUR
sampleproduct1	20+	10 EUR
sampleproduct2	1 through 29	11.5 EUR
sampleproduct2	30+	0.99 EUR
sampleproduct3	1 through 9	651.89 EUR
sampleproduct3	10+	599.99 EUR

```
#  
# Some pricerows for our products (4)  
#  
INSERT_UPDATE Product;code[unique=true];$catalogVersion;$prices;Europe1PriceFactory_PTG(code);  
;sampleproduct1;clothescatalog:Online;"1 pieces = 12,00 EUR, 20 pieces = 10,00 EUR";Tax_Full;  
;sampleproduct2;clothescatalog:Online;"1 pieces = 11,50 EUR, 30 pieces = 0,99 EUR";Tax_Half;  
;sampleproduct3;clothescatalog:Online;"1 pieces = 651,89 EUR, 10 pieces = 599,99 EUR";Tax_Full;
```

## i Note

### German locale setting

The notation for the PriceRow instances is given in German locale, the US locale-compliant code would look like this:

```
;sampleproduct1;clothescatalog:Online;"1 pieces = 12.00 EUR; 20 pieces = 10.00 EUR";Tax_Full;
```

However, defining prices for products is not always a good idea, as prices from products are not passed on to variants of those products. The following code snippet shows how to create product variants and how to set prices on them.

## Some Variants of Sampleproduct1 Including Prices (5)

```
#  
# Some variants of sampleproduct1 including prices (5)  
#  
INSERT_UPDATE Product;code[unique=true]; $baseProduct;$catalogVersion; approvalStatus(code); Europe1F  
;sampleproduct1-00;sampleproduct;clothescatalog:Online;approved;Tax_Full;11,50 EUR N; pieces;  
;sampleproduct1-01;sampleproduct;clothescatalog:Online;approved;Tax_Full;11,00 EUR N; pieces;
```

Note the **N** suffix to the price setting - it specifies that this price is given as net. The default for a price setting is gross (G).

## Defining an Order (6)

```
#  
# Defining an Order (6)  
#  
INSERT_UPDATE Order;code[unique=true];user(uid);date[dateformat=dd.MM.yyyy HH:mm];currency(isocode);r  
;SampleOrder1;demo;17.01.2006 10:58;EUR;false;advance;false  
## impex.getLastImportedItem().setDeliveryAddress(impex.getLastImportedItem().getUser().getDefaultDel  
## impex.getLastImportedItem().setPaymentAddress(impex.getLastImportedItem().getUser().getDefaultPaym
```

## Setting OrderEntries on the Order (7)

An Order in SAP Commerce consists of individual OrderEntry instances. The Order itself does not have references to amounts of products. The following code snippet adds some OrderEntry instances to **SampleOrder1**.

```
#  
# Setting OrderEntries on the Order (7)  
#  
INSERT_UPDATE OrderEntry;order(code)[unique=true];product(code,catalogVersion(catalog(id),version))[  
;SampleOrder1;sampleproduct1:clothescatalog:Online;1;false;pieces;0  
;SampleOrder1;sampleproduct2:clothescatalog:Online;2;false;pieces;1  
;SampleOrder1;sampleproduct3:clothescatalog:Online;3;false;pieces;2
```

## Calculating the Order (8)

By default, a newly created or recently modified Order instance is not calculated. In other words: the SAP Commerce default PriceFactory Europe1 states that the Order's totals are not correct and re-calculation is necessary. The following code snippet re-calculates the **SampleOrder1** Order:

```
#  
# Calculating the Order (8)  
#  
UPDATE Order;code[unique=true]  
;SampleOrder1  
#%    impex.getLastImportedItem().recalculate();
```

1. **SampleOrder1** is "touched" by the ImpEx framework:

```
UPDATE Order;code[unique=true]  
;SampleOrder1
```

2. **SampleOrder1** is re-calculated by calling the **recalculate()** method on the return value of the static **getLastImportedItem()** method.

```
#%    impex.getLastImportedItem().recalculate();
```

## Importing Users and Usergroups

This section describes how to import users and usergroups using ImpEx.

For ImpEx samples on how to create orders, please refer to [Importing Products and Orders](#).

### Creating Users

For both the Customer and the Employee type, the only mandatory attribute is **uid**. Also note that it is not possible to create a Customer with the same **uid** as an existing Employee, and vice versa.

It is, however, possible to create a Usergroup and a User (Customer or Employee) with the same **uid**. This is because the check for duplicate uids is not done at the Principal type, but only with the User type. This is not an issue in itself; but when an ImpEx script references the Principal.uid attribute, SAP Commerce will not know which Principal to use: the User (Customer or Employee, respectively) or the Usergroup. This will result in an exception as below:

```
ERROR [127.0.0.1] (master) [ImpExImportJob] line 2 at main script: more than one item found for 'testde.hybris.platform.impex.jalo.ImpExException: line 2 at main script: more than one item found for 'testNested: de.hybris.platform.jalo.JaloInvalidParameterException: more than one item found for 'testtemp' at de.hybris.platform.impex.jalo.Importer.importNext(Importer.java:686)  
at de.hybris.platform.impex.jalo.cronjob.ImpExImportJob.doImport(ImpExImportJob.java:241)  
at de.hybris.platform.impex.jalo.cronjob.ImpExImportJob.performJob(ImpExImportJob.java:206)
```

```
at de.hybris.platform.impex.jalo.cronjob.ImpExImportJob.performCronJob(ImpExImportJob.java:16)
at de.hybris.platform.cronjob.jalo.Job.execute(Job.java:1043)
at de.hybris.platform.cronjob.jalo.Job.performImpl(Job.java:677)
at de.hybris.platform.cronjob.jalo.Job.performImpl(Job.java:615)
at de.hybris.platform.cronjob.jalo.Job.perform(Job.java:545)
at de.hybris.platform.impex.jalo.ImpExManager.importData(ImpExManager.java:573)
at org.apache.jsp.import_jsp._jspService(import_jsp.java:198)
at org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:70)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
at org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:374)
at org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:342)
at org.apache.jasper.servlet.JspServlet.service(JspServlet.java:267)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:206)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:206)
at de.hybris.platform.util.RootRequestFilter.doFilter(RootRequestFilter.java:642)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:206)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:206)
at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:233)
at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:191)
at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:128)
at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:102)
at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:109)
at org.apache.catalina.valves.AccessLogValve.invoke(AccessLogValve.java:568)
at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:286)
at org.apache.coyote.http11.Http11Processor.process(Http11Processor.java:845)
at org.apache.coyote.http11.Http11Protocol$Http11ConnectionHandler.process(Http11Protocol.java:447)
at org.apache.tomcat.util.net.JIoEndpoint$Worker.run(JIoEndpoint.java:447)
at java.lang.Thread.run(Thread.java:619)
```

```
Caused by: de.hybris.platform.jalo.JaloInvalidParameterException: more than one item found for 'teste
at de.hybris.platform.impex.jalo.translators.ItemExpressionTranslator.searchItem(ItemExpressi
at de.hybris.platform.impex.jalo.translators.ItemExpressionTranslator.convertToJalo(ItemExpres
at de.hybris.platform.impex.jalo.translators.SingleValueTranslator.importValue(SingleValueTra
at de.hybris.platform.impex.jalo.header.StandardColumnDescriptor.importValue(StandardColumnDe
at de.hybris.platform.impex.jalo.imp.DefaultValueLineTranslator.translateColumnValues(DefaultV
at de.hybris.platform.impex.jalo.imp.DefaultExistingItemResolver.translateUniqueKeys(DefaultE
at de.hybris.platform.impex.jalo.imp.DefaultExistingItemResolver.findExisting(DefaultExistingI
at de.hybris.platform.impex.jalo.imp.DefaultImportProcessor.processItemData_Impl(DefaultImportP
at de.hybris.platform.impex.jalo.imp.DefaultImportProcessor.processItemData(DefaultImportProces
at de.hybris.platform.impex.jalo.imp.ImpExImportReader.readLine(ImpExImportReader.java:457)
at de.hybris.platform.impex.jalo.Importer.doImport(Importer.java:241)
at de.hybris.platform.impex.jalo.Importer.importNext(Importer.java:668)
... 31 more
```

## NESTED EXCEPTION:

```
de.hybris.platform.jalo.JaloInvalidParameterException: more than one item found for 'testtemp1234' usi
        at de.hybris.platform.impex.jalo.translators.ItemExpressionTranslator.searchItem(ItemExpressi
        at de.hybris.platform.impex.jalo.translators.ItemExpressionTranslator.convertToJalo(ItemExpres
        at de.hybris.platform.impex.jalo.translators.SingleValueTranslator.importValue(SingleValueTra
        at de.hybris.platform.impex.jalo.header.StandardColumnDescriptor.importValue(StandardColumnDe
        at de.hybris.platform.impex.jalo.imp.DefaultValueLineTranslator.translateColumnValues(DefaultV
        at de.hybris.platform.impex.jalo.imp.DefaultExistingItemResolver.translateUniqueKeys(DefaultE
        at de.hybris.platform.impex.jalo.imp.DefaultExistingItemResolver.findExisting(DefaultExistingI
        at de.hybris.platform.impex.jalo.imp.DefaultImportProcessor.processItemData_Impl(DefaultImportP
        at de.hybris.platform.impex.jalo.imp.DefaultImportProcessor.processItemData(DefaultImportProce
        at de.hybris.platform.impex.jalo.imp.ImpExImportReader.readLine(ImpExImportReader.java:457)
        at de.hybris.platform.impex.jalo.Importer.doImport(Importer.java:241)
        at de.hybris.platform.impex.jalo.Importer.importNext(Importer.java:668)
        at de.hybris.platform.impex.jalo.cronjob.ImpExImportJob.doImport(ImpExImportJob.java:241)
        at de.hybris.platform.impex.jalo.cronjob.ImpExImportJob.performJob(ImpExImportJob.java:206)
        at de.hybris.platform.impex.jalo.cronjob.ImpExImportJob.performCronJob(ImpExImportJob.java:16
        at de.hybris.platform.cronjob.jalo.Job.execute(Job.java:1043)
        at de.hybris.platform.cronjob.jalo.Job.performImpl(Job.java:677)
        at de.hybris.platform.cronjob.jalo.Job.performImpl(Job.java:615)
        at de.hybris.platform.cronjob.jalo.Job.perform(Job.java:545)
        at de.hybris.platform.impex.jalo.ImpExManager.importData(ImpExManager.java:573)
        at org.apache.jsp.import_jsp._jspService(import_jsp.java:198)
        at org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:70)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
        at org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:374)
        at org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:342)
        at org.apache.jasper.servlet.JspServlet.service(JspServlet.java:267)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.ja
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:206)
        at de.hybris.platform.util.RootRequestFilter.doFilter(RootRequestFilter.java:642)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.ja
```

```
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:206)
at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:233)
at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:191)
at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:128)
at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:102)
at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:109)
at org.apache.catalina.valves.AccessLogValve.invoke(AccessLogValve.java:568)
at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:286)
at org.apache.coyote.http11.Http11Processor.process(Http11Processor.java:845)
at org.apache.coyote.http11.Http11Protocol$Http11ConnectionHandler.process(Http11Protocol.java:447)
at org.apache.tomcat.util.net.JIoEndpoint$Worker.run(JIoEndpoint.java:447)
at java.lang.Thread.run(Thread.java:619)
```

ERROR [127.0.0.1] (master) [ImpExManager] Import has caused an error, see logs of cronjob with code=6

## Customer

```
INSERT_UPDATE Customer;uid[unique=true];customerID[unique=true];name;description;sessionLanguage(isocppetersonson;K2006-C0001;Peter Petersonson;a good customer from denmark;de;EUR;customergroup;1234
```

## Employee

```
INSERT_UPDATE Employee;UID[unique=true];writableCatalogVersions(catalog(id),version);readableCatalog  
;customerservice;hwcatalog:Online;clothescatalog:Online;Customer service demouser.Can edit order and
```

## Setting Encrypted Passwords for Users

To set encrypted passwords for users, you need to specify a special translator class (the `UserPasswordTranslator`, for example). By using `UserPasswordTranslator` it is not possible to run encryption during import. For example, you cannot specify a plain text password and have the password automatically encrypted during import. Instead, if you specify an encryption algorithm, you need to specify the password's hash value in that algorithm.

```
INSERT_UPDATE User; uid[unique=true]; @password[translator=de.hybris.platform.impex.jalo.translators.;test;bla;customergroup
```

The following example stores passwords directly by specifying the desired encoding together with the encoded password

```
INSERT Employee; uid[unique=true]; @password[translator=de.hybris.platform.impex.jalo.translators.Use  
; max ; *:plainPassword
```

However, the `ConvertPlaintextToEncodedUserPasswordTranslator` translator encrypts plain text passwords.

```
INSERT_UPDATE Customer;uid[unique=true];@password[translator=de.hybris.platform.impex.jalo.translator];foo;md5:plaintext password
```

See also the API Doc on the [UserPasswordTranslator](#) and [ConvertPlaintextToEncodedUserPasswordTranslator](#)

## Adding a User to Several Usergroups

By specifying several, comma-separated usergroups, you can add a user to several usergroups in one single step. For example, the following ImpEx script adds the Customer **ppeterson** to both the **customergroup** and **bestcustomergroup** usergroups:

```
INSERT_UPDATE Customer;uid[unique=true];customerID[unique=true];groups(uid);password  
;ppeterson;K2006-C0001;customerqgroup,bestcustomerqgroup;1234
```

# Creating Usergroups

The only mandatory attribute for a usergroup is **uid**.

```
INSERT_UPDATE UserGroup;UID[unique=true];locname[lang=de];locname[lang=en];description;groups(uid);re
;print_test_group;Print Testgruppe;Print test group;Group for print example publication;;de,en;de,en;
```

## Creating Links Between Usergroup and User

In case of using the import script given above for importing a **User**, the related groups the user is assigned is set by using the jalo-attribute **groups** of the **User** type. Setting references part of a link by using the related jalo-attributes can cause performance problems because of additionally performed business logic. A better approach is to set the references directly by importing instances of the **PrincipalGroupRelation** (and not using the **groups** attribute at the **User** type).

In the first step change the import script for the user (here demonstrated for Customer type) to:

```
INSERT_UPDATE Customer;uid[unique=true];customerID[unique=true];name;description;sessionLanguage(isoc
;ppetersonson;K2006-C0001;Peter Petersonson;a good customer from denmark;de;EUR;1234
```

In the second step insert the relation instances to the **PrincipalGroupRelation**

```
INSERT_UPDATE PrincipalGroupRelation;source(uid)[unique=true];target(uid)
;ppetersonson;customergroup;
```

To avoid unneeded import cycles, ensure that you first import the users and usergroups before importing the relations between them.

## Creating Addresses

The only mandatory attribute for an Address is **owner**. An Address must be assigned to a user, and changing the user is not possible after the Address has been created.

There are two special characteristics about Addresses:

- Addresses have no unique attribute per se
- Addresses can be non-unique even in combination of all attributes

In the course of the SAP Commerce factory default ordering process, a copy is made of both the delivery address and the payment address. If a user does several orders, there may be a large number of Address copies for that user, and these addresses may actually all be the same, except for the creation date. To make sure that addresses will be created correctly even if several of them are essentially the same, do not use the **INSERT\_UPDATE** header mode, but **INSERT**.

**INSERT\_UPDATE** tries to locate an existing SAP Commerce item before creating a new one, but it will fail if several identical Addresses exist and no unique one can be determined. **INSERT** will simply create a new Address.

```
INSERT Address;owner(Customer.uid)[unique=true];streetname[unique=true];streetnumber[unique=true];pos
;ppetersonson;Drittnünggetn;13a;103 25;false;Stockholm;at;true;true;true;false;Peter;Petersonson;pet
```

For more information, see:

- [Users in Platform](#)
- [Ordering Process](#)

## Clearing Model Context

Clearing the model context after every **INSERT**, **INSERT\_UPDATE**, **UPDATE**, and **REMOVE** ImpEx operation ensures that fresh and valid data is passed on to Interceptors' context.

In the default SAP Commerce configuration, the model context is not cleared after INSERT, INSERT\_UPDATE, UPDATE, and REMOVE ImpEx operations. To enable such cleanup, add the following property to your local.properties:

```
impex.clearing.model.context.after.processing.each.item.enabled=true
```

### i Note

Clearing the model context can negatively impact the performance of SAP Commerce.

For example, after the following ImpEx import command:

```
INSERT_UPDATE Language;isocode[unique=true];active;
;de;true;

INSERT_UPDATE Country;isocode[unique=true];name[lang=de];name[lang=en];active;
;DE;Deutschland;Germany;true;

INSERT_UPDATE Company;uid[unique=true];buyer;manufacturer;supplier;carrier;country(isocode);locname[1
;exampleCompany;true;false;true;DE;hybris GmbH;hybris GmbH;

INSERT_UPDATE Address;owner(Company.uid)[unique=true];streetname[unique=true];streetnumber[unique=true];
;exampleCompany;exampleStreetName;1;D-1234;exampleTown;DE;
;exampleCompany;exampleStreetName;2;D-1234;exampleTown;DE;
;exampleCompany;exampleStreetName;3;D-1234;exampleTown;DE;
;exampleCompany;exampleStreetName;4;D-1234;exampleTown;DE;
;exampleCompany;exampleStreetName;5;D-1234;exampleTown;DE;
```

With the following Interceptor for AddressModel:

```
public class AddressValidateInterceptor implements ValidateInterceptor<AddressModel>
{
    @Override
    public void onValidate(final AddressModel addressModel, final InterceptorContext ctx) throws InterceptorException
    {
        final ItemModel owner = addressModel.getOwner();
        if (owner instanceof CompanyModel)
        {
            final int addressesSize = ((CompanyModel) owner).getAddresses().size();
            LOG.info("Assigned addresses to Company:" + addressesSize);
        }
    }
}
```

SAP Commerce logs the correct data if the model context cleanup is enabled:

```
Assigned addresses to Company:0
Assigned addresses to Company:1
Assigned addresses to Company:2
Assigned addresses to Company:3
Assigned addresses to Company:4
```

## Using ImpEx with Backoffice or SAP Commerce Administration Console

You can use the impex extension via the SAP Commerce API, Backoffice or SAP Commerce Administration Console.

For details, see:

- [Import](#)
- [Export](#)

# Import

For importing data to platform via Backoffice, you have to create and configure a CronJob of type ImpExImportCronJob.

The configuration of such a CronJob and the import result attributes are described in the next paragraph. To make the configuration of such a CronJob easier, Backoffice provides a wizard of type ImpExImportWizard. For more information, see [Import Wizard](#).

Another possibility for importing data, which is also a graphical alternative, is the usage of the ImpEx Web. This alternative is intended for development only and can only be used by administrators. For more information, see [Import Through Administration Console](#).

For systems with a large number of products, SAP Commerce offers a multithreaded import. For details, see [Multithreaded Import](#).

## Import CronJob

To start an import using an ImpExImportCronJob, log into Backoffice and navigate to **System > Background Processes > CronJobs**. A cronjob collection browser opens. In the browser, open the **ImpEx Import Cronjob** wizard.

The screenshot shows the SAP Backoffice interface with the following navigation path: Home > System > Background Processes > CronJobs. The left sidebar has a 'System' section expanded, showing sub-options like OAuth, CORS Filter, Advanced Configuration, Tools, Output Documents, Workflow Administration, Validation, Scripting, Business Processes, Background Processes, and CronJobs. The 'CronJobs' option is highlighted with a yellow box. The main content area displays a list of CronJob types, each with a small icon and a 'Code' or 'Jc' label. The 'ImpEx Import Cronjob' option is highlighted with a yellow box. The toolbar at the top right includes a search bar, a filter icon, and a 'CSV' export button. The status bar at the bottom indicates '0 ITEMS SELECTED' and 'No items selec'.

In the wizard, you can fill the basic parameters. Save your cronjob, and open it. The details are explained in the next sections

## i Note

This section only discusses those attributes that are specific to the **impex** Import CronJob. Attributes common to CronJobs are not discussed.

## Administration Tab

The **Administration** tab contains some important administrative options. Here you can specify all ImpEx specific settings for starting an import cronjob. The most important attribute is the mandatory **Import Media**. It specifies the media containing the source data in ImpEx format. If you have your import data packed to a ZIP file (for example as a result of an export), you can also specify a media containing this ZIP file. In that case, make sure that the **Import File in ZIP** attribute specifies the import file within the archive. When starting the cronjob, the archive is unpacked, the import file is assigned to the **Import Media** attribute and all other files of the archive to the **External Data** collection.

The **External Data** collection specifies additional medias containing **ImpEx** files referenced from the main import media using a bean shell command like **includeExternalDataMedia**. You can find more information in [ImpEx API](#), section **Inclusion of Data**.

Using the **Archive of Medias**, you can specify a ZIP-archive containing files for a media import, for example, picture files using the translator. Furthermore, you can specify the root directory within this archive where the related files reside.

For setting the locale to use for parsing the import data you can use the **locale setting** attribute. The **mode** attribute specifies the strictness mode for the import where each mode has different restrictions concerning the consistency of underlying import data (see [ImpEx API](#), section **Validation Modes**).

You can enable the ServiceLayer in Impex by changing the **Legacy mode** option to **No**.

Filter Tree entries  SEARCH LOCK UNLOCK

### ImpEx-Import : fooCronjob - UNKNOWN - UNKNOWN

REFRESH SAVE

LOG TASK RUN AS TIME SCHEDULE SYSTEM RECOVERY ADMINISTRATION

#### ESSENTIAL

Code	Current status	job	Last result
fooCronjob	NEW	ImpEx-Import	N/A
Timetable	Last start time	Enabled	Last end time
Not scheduled		<input checked="" type="radio"/> True <input type="radio"/> False	

#### METADATA

PK	Type	Time created	Time modified
8796256928245	ImpExImportCronJob	Mar 5, 2018 1:32:06 PM	Mar 5, 2018 1:32:06 PM
Owner			
<input type="text"/> ...			

#### CHANGES

Last changes

emailNotificationTemplate,enable...
<input type="text"/> ...

#### UNBOUND

Encoding of unresolved data	Retry import of unresolved data	Enable code execution	Enable code execution in external data
UTF-8	<input checked="" type="radio"/> True <input type="radio"/> False	<input checked="" type="radio"/> True <input type="radio"/> False	<input type="radio"/> True <input checked="" type="radio"/> False
Enable the consideration of ImpEx specific-syntax in external data too	Log changes in hMC	External data	Import media
<input type="radio"/> True <input checked="" type="radio"/> False	<input type="radio"/> True <input checked="" type="radio"/> False	<input type="text"/> ...	<input type="text"/> ...
Locale setting	Number of processes	Archive of medias	Basedir within archive
de	1	<input type="text"/> ...	<input type="text"/> ...
Mode	Unresolved data	Unzip media archive temporarily	Imported items
Import (Strict)	<input type="button" value="+ Create new ImpEx Media"/>	<input type="radio"/> True <input checked="" type="radio"/> False	<input type="text"/>
Working media	Import file in ZIP		
<input type="button" value="+ Create new ImpEx Media"/>	<input type="text"/>		
ActiveCronjobHistory	Documents	Assigned Cockpit Item Templates	Changes
<input type="text"/> ...	<input type="button" value="+ Create new Output Document"/>	<input type="text"/> ...	<input type="button" value="+ Create new Change descriptor"/>
Comments	CronjobHistoryEntries	Error mode	Maximum number of rows
<input type="text"/> ...	<input type="button" value="+ Create new CronjobHistory"/>	<input type="text" value="Ignore"/>	1000
Request Abort		Request abort step	
<input type="radio"/> True	<input checked="" type="radio"/> False	<input type="radio"/> N/A	<input type="radio"/> True <input checked="" type="radio"/> False <input type="radio"/> N/A
Is blocked for processing			
<input type="radio"/> True <input checked="" type="radio"/> False <input type="radio"/> N/A			
<input type="text"/> ...	<input type="text"/> ...		
Once-only executable	Delete (after termination)	Retry execution, if the job can not be executed yet	Alternative Data Source ID
<input type="radio"/> True <input checked="" type="radio"/> False	<input type="radio"/> True <input checked="" type="radio"/> False	<input checked="" type="radio"/> True <input type="radio"/> False	<input type="text"/>

Figure: Sample ImpEx CronJob window in Backoffice.

Option	Description/Comment
Import Media	<p>A media containing the data for import. Within the data you can include medias set at <b>External Data</b> using the bean shell command <code>includeExternalDataMedia</code>. You have to <b>Enable Code Execution</b> for doing such an include.</p> <p>Instead of plain data in <b>impex</b> format you can specify a ZIP file directly, for example, a result of an export, marking the main import file using the <b>Import File in ZIP</b> attribute.</p>
Import File in ZIP	<p>If media at <b>Import Media</b> is a ZIP file, you can specify here the main import file <code>importscript.impex</code>. When starting the cronjob, this file is assigned to <b>Import Media</b>, the remaining files are added to the <b>External Data</b> collection.</p> <p>Importing a ZIP file with an impex only works if one of the files inside the ZIP file - the main import file - is named <code>importscript.impex</code>. Do not use other names for that file.</p>
External Data	<p>Holds a set of data medias in ImpEx format that can be referenced within <b>Import Media</b> using the bean shell command <code>includeExternalDataMedia</code>.</p>
Archive of Medias	<p>For specifying a ZIP archive containing files for a media import using the <b>MediaDataTranslator</b>.</p>
Basedir within archive	<p>Specifies the root directory within the <b>Archive of Medias</b>.</p>
Unzip media archive temporarily	<p>The media archive is unzipped while importing to get better performance.</p>
Locale setting	<p>Specifies the used locale setting for the data import. Important for data like numbers or dates where the separator symbols are different per locale.</p>
Mode	<p>Specifies the validation mode for the import where each has a different kind of strictness related to header validations, and so on. See <a href="#">ImpEx API</a>, section <i>Validation Modes</i>.</p>
Enable code execution	<p>Enables the execution of bean shell code within the <b>Import Media</b>. If not enabled, bean shell code is ignored. You have to enable it for using the include mechanism of external data.</p>
Enable the consideration of ImpEx specific-syntax in external data too	<p>If included <b>external data</b> contains ImpEx syntax-like comments, headers and so on, this option has to be enabled. Otherwise the ImpEx syntax is not interpreted and used as a normal value line (which causes errors).</p>
Enable code execution in external data	<p>If included <b>external data</b> contains bean shell code, this option has to be enabled. Otherwise the code is ignored. For enabling this option you also have to enable the <b>Enable the consideration of ImpEx specific-syntax in external data too</b> option, otherwise the bean shell code is interpreted as a value line and therefore is not recognized as bean shell code.</p>
Number of processes	<p>SAP Commerce optionally supports multithreading for ImpEx imports. This attribute specifies the number of threads for the ImpEx CronJob to use. Defaults to 1 (which uses one single thread only and, by consequence, does not use multithreading). The more threads ImpEx is allowed to use, the faster the import completes. However, as ImpEx scripts may have implicit dependencies, some ImpEx</p>

Option	Description/Comment
	imports fail using multithreading and succeed using one single thread only. For details, see <a href="#">Multithreaded Import</a> .
Once-Only Executable	If set, the cronjob can be performed only once. If you disable this flag, remember to delete the reference to the working media at restart of a cronjob, otherwise the work media are used again instead of the newly configured import file.
Delete (after termination)	The cronjob is deleted after successful termination.
Error Mode	What should be done on an error, FAIL the job or IGNORE the error? PAUSE mode is not supported by an import job

## Log Tab

At the log tab you can find at first all log information and at second information about possible unresolved value lines (caused by item references that could not be resolved).

The screenshot shows the SAP Fiori Log Tab interface. At the top, there is a search bar with the text "foo" and a yellow "SEARCH" button. Below the search bar is a navigation menu with items like Home, Inbox, System, Catalog, Multimedia, User, Order, Price Settings, Internationalization, Marketing, and Cockpit. Under the Cockpit section, there is a "SAVED QUERIES" section with a note "No queries". The main content area is titled "ImpEx-Import : fooCronjob - UNKNOWN - UNKNOWN". It has tabs for LOG, TASK, RUN AS, TIME SCHEDULE, SYSTEM RECOVERY, and ADMINISTRATION, with LOG being selected. The LOG section contains several configuration fields: "Code" (fooCronjob), "Current status" (NEW), "Job" (ImpEx-Import), "Last result" (N/A), "Timetable" (Not scheduled), "Last start time" (empty), "Enabled" (True), "Last end time" (empty). Below this is the "LOG" section with fields for "Log to database" (True), "Log level database" (ERROR), "Logs Count" (5), and a note "For performance reasons job logs are not displayed here. Use Search" with a "FIND JOB LOGS FOR CRONJOB" button. There are also fields for "Logs Operator" (AND), "Logs Days Old" (14), "Log to file" (True), "Log level file" (INFO), "Files Count" (5), "Files Operator" (AND), "Files Days Old" (14), "Log files" (+ Create new Log File), and an "Entire log" field. The bottom section is titled "JOB STEPS" and shows fields for "Current step" and "Processed steps".

Option	Description/Comment
Entire Log	Displays the first 500 log entries of database (not the logs stored at file).
Log to Database	Enables the logging to database.
Log Level Database	Sets the log level for database logging.
Log to File	Enables logging to file.

Option	Description/Comment
Log Level File	Sets the log level for file logging.
Log Files	References the created log files (not database log).

## Import Wizard

To start an import, log in to Backoffice and navigate to **System > Tools > Import**. A wizard opens that allows you to set the basic parameters for a CronJob that imports data to Platform.

### Configuration Tab

The main tab of this wizard specifies the source medias for the import and some basic settings.

ImpEx import X

IMPEX IMPORT Configuration > IMPEX IMPORT Advanced configuration > IMPEX IMPORT Results

Choose media:

Upload new ImpEx:

UPLOAD CREATE RESET

Locale settings:

Import file in ZIP:

CANCEL NEXT

Option	Description/Comment
Choose media	A media containing the data for import. Within the data you can include medias set at <b>External Data</b> using the bean shell command <b>includeExternalDataMedia</b> (You have to <b>Enable Code Execution</b> for doing such an include).  Instead of plain data in ImpEx format you can specify a ZIP archive directly (e.g. a result of an export) marking the main import file using the <b>Import File in ZIP</b> attribute.
Upload new Impex	Allows to upload new impex files
Locale setting	Specifies the used locale setting for the data import. Important for data like numbers or dates where the separator symbols are different per locale.

Option	Description/Comment
Import file in ZIP	If media at <b>Import Media</b> is a ZIP file, you can specify here the main import file <b>importscript.impex</b> . When starting the cronjob, this file is assigned to <b>Import Media</b> , the remaining files are added to the <b>External Data</b> collection.  Importing a ZIP file with an impex only works if one of the files inside the ZIP file - the main import file - is named <b>importscript.impex</b> . Do not use other names for that file.

## Advanced Configuration Tab

At the **Advanced configuration** tab you can configure an archive containing the media binary data for import and set some advanced settings.

**ImpEx import** X

IMPEX IMPORT Configuration > IMPEX IMPORT Advanced configuration > IMPEX IMPORT Results

Media-Zip:

Upload new media:

**UPLOAD**   **CREATE**   **RESET**

Distributed Mode:

Strict import mode:

Execute asynchronously:

Log audit trial:

Allow code execution from within the file:

Error mode:

Number of processes:

**BACK**   **CANCEL**   **START**

Option	Description/Comment
Media-Zip	For specifying a ZIP archive containing files for a media import using the <b>MediaDataTranslator</b> .
Distributed mode	Enables import in distributed modes

Option	Description/Comment
Strict import mode	Specifies the validation mode for the import where each has a different kind of strictness related to header validations etc. See ImpEx API, section Validation Modes.
Execute asynchronously	
Log audit trial	Enables the tracing of item modifications. Changes can be revised at the Administration tab of the related item. Attention: Enabling can cause a performance problem for mass data!
Allow code execution from within the file	Enables the execution of bean shell code within the <b>Import Media</b> . If not enabled, bean shell code is ignored. Has to be enabled for using the include mechanism of external data.
Error mode	<p>Allows you to set the error mode of the Import CronJob, that is, how the Import CronJob deals with occurring errors. Possible values:</p> <ul style="list-style-type: none"> <li>• Fail Aborts the import at the location where the error occurred.</li> <li>• Pause The import is suspended at the location where the error occurred to allow manual intervention. An operator can then manually resume or abort the import, depending on error assessment.</li> <li>• Ignore Continues import as if no error had occurred.</li> </ul>
Number of processes	SAP Commerce optionally supports multi-threading for ImpEx imports. This attribute specifies the number of threads for the ImpEx CronJob to use. Defaults to 1 (which uses one single thread only and, by consequence, does not use multi-threading). The more threads ImpEx is allowed to use, the faster the import completes. However, as ImpEx scripts may have implicit dependencies, some ImpEx imports fail using multi-threading and succeed using one single thread only.

## Result Tab

Under the **Results** tab the status is shown with some logs of the currently performed import.

## ImpEx import



IMPEX IMPORT Configuration > IMPEX IMPORT Advanced configuration > IMPEX IMPORT Results

ImpEx file:

00000003

Cron Job:

ImpEx-Import : 000003UW - RUNNING - UNKNOWN

Status:

Async import started

ImpEx logs:

[Empty box]

**DONE**

By clicking on a cronjob, you may get some additional information about the import, or download some logs.

## Import Through Administration Console

You can perform ImpEx import through SAP Commerce Administration Console.

1. Open the SAP Commerce Administration Console.
2. Go to the **Console** tab and select **ImpEx Import** option.
3. The **Impex Import** page displays.

4. You can perform import in two ways:

- In the **Import content** section paste a script and click the **Import content** button.
- In the **Import script** section, choose a script from a different location and click the **Import file** button.

#### → Tip

You can enable the ServiceLayer in ImpEx by disabling the **Legacy mode**. To do it, untick the relevant box. The default value is **Legacy Mode** (the box is ticked by default).

## Multithreaded Import

To support systems with large numbers of products, SAP Commerce offers a multithreaded import through the SAP Commerce **impex** extension functionality.

Multithreaded import enables you to use as many cores as you like to speed up the import of a given file. You can configure the number of cores for each import job separately.

Our tests on 16 core machines have proved almost linear scaling if the database can handle the load.

#### i Note

It is not possible to ensure a strict writing order in a multithreaded import process. Therefore, try to have only one line per item - this ensures that only one thread is processing it. If you have multiple lines per item and the attributes overlap, it is possible that one thread can overwrite the effects of the other thread, and might end up with wrong data.

## Configure Multithreaded Import

You can configure the number of threads (also referred to as worker threads) as follows:

- in SAP Commerce Administration Console, see [Configuring in SAP Commerce Administration Console](#)
- in Backoffice, see [Configuring in Backoffice](#)
- via API, see [Configuring in API](#)
- in the `local.properties` configuration file as the default, see [Configuring in the local.properties File](#)

## Configuring in SAP Commerce Administration Console

You can configure a multithreaded import in the SAP Commerce Administration Console.

### Procedure

1. Open the SAP Commerce Administration Console.
2. Go to the **Console tab** and select the **Impex Import** option.  
The **Impex Import** page displays.
3. Set the maximum number of threads to use in the **Max. threads** field.

You have the same field in the **Import content** and the **Import script** sections.

A screenshot of a user interface showing a numeric input field labeled "Max. threads" with the value "1". To the right of the input field are two small buttons, one with an upward arrow and one with a downward arrow, used for incrementing or decrementing the value.

#### i Note

The **Max. threads** setting overrides the default value of worker threads only for the duration of the import. After the import has finished, the **Max. threads** is reset to the default value.

## Configuring in Backoffice

You can configure a multithreaded import in Backoffice.

### Procedure

1. Log in to Backoffice.
2. In the explorer tree, navigate to **System > Tools > Import** to open the **ImpEx Import** wizard.

In the wizard, on the **Advanced configuration** tab, you can configure the number of processes:



## ImpEx import

IMPEX IMPORT Configuration > IMPEX IMPORT Advanced configuration > IMPEX IMPORT Results

Strict import mode:

Execute asynchronously:

Log audit trial:

Allow code execution from within the file:

Error mode:  
Fail

Number of processes:  
16

**BACK** **CANCEL** **START**

3. Start the import if your configuration is ready.

## Configuring in API

### ImpEx ServiceLayer API

In the SAP Commerce ServiceLayer there is a possibility to set up multithreaded import by configuring special **ImportConfig** object and use this object as parameter for **ImportService#importData** method as follows:

```
// assume that you have already bytes object containing script content
byte[] bytes;

final ImportConfig config = new ImportConfig();
// change maximum threads number to 8
config.setMaxThreads(8);
config.setScript(new StreamBasedImpExResource(new ByteArrayInputStream(bytes), "en"));

// now use Config object
final ImportResult result = importService.importData(config);
```

### ImpEx Jalo API

Technically, the multi-threaded import is run by these types:

- **ImpExImportCronJob**
- **ImpExImportJob**

To use multi-threaded import, you therefore have to use a combination of these CronJob components:

- An **ImpExImportCronJob** or a subtype such as the **ClassificationImportCronJob**.
- An **ImpExImportJob** or a subtype such as the **ClassificationImportJob**.

You can define the maximum number of threads in these different places:

- Using the **maxThreads** attribute of the **ImpExImportCronJob**. The value defined here has the highest priority. Optional.
- Using the **maxThreads** attribute of the **ImpExImportJob**. The value defined here has medium priority and is overridden by a value defined in the **ImpExImportCronJob**. Optional.
- Using a system property, **impex.import.workers** (see [Configuring in the local.properties File](#)). The value defined here has the lowest priority and is overridden by values defined in the **ImpExImportCronJob** and the **ImpExImportJob**. This is the default value for all **ImpExImportCronJob** and **ImpExImportJob** instances in the system.

The maximum number of threads for an individual **ImpExImportJob** is therefore defined in this order:

1. The **ImpExImportJob** uses the maximum thread number value of the **ImpExImportCronJob**.
2. The **ImpExImportJob** uses the maximum thread number value of the **ImpExImportJob** itself.
3. The **ImpExImportJob** uses the value of the **impex.import.workers** property; see [Configuring in the local.properties File](#).

## Configuring in the local.properties File

Within the `local.properties` configuration file you can configure the default number of ImpEx worker threads by setting the property `impex.import.workers`.

You can always disable the multithreaded ImpEx import by setting the number of workers to **1**.

Example: Add the line below to your `local.properties` file in order to use eight worker threads as the default for all your multithreaded ImpEx imports.

```
impex.import.workers=8
```

### → Tip

You can assign an expression to the `impex.import.workers` property. In this expression you can use `#cores` constant which will be substituted by a number of cores available on your system. Default value is set to the doubled number of available cores:

`local.properties`

```
impex.import.workers=#cores * 2
```

Because you can set this only once you should take into account that effective number of workers depends on the target hardware setup. Bear in mind that the expression should be based on your database setup. If the database is slow, you should increase the number of workers accordingly. Otherwise you should keep it close to the default value. For example:

- Doubled number of available cores plus additional three workers

`local.properties`

```
impex.import.workers=#cores * 2 + 3
```

- Number of available cores incremented by one

## local.properties

```
impex.import.workers=1 + #cores
```

# Export

To export data from Platform to CSV-files via Backoffice, you have to create and configure a cronjob of type **ImpExExportCronJob**.

You first need an export script that you can generate using the Script Generator. The configuration of such an export **CronJob** and the export result attributes are described later. For making the configuration of such a **CronJob** more easy, Backoffice provides a **Wizard** of type **ImpExExportWizard**. For details, see [Export Wizard](#).

Another possibility for exporting data offering a graphical interface, is the ImpEx Web. This alternative is intended for development only and can only be used by administrators. For details, see [Export Through Administration Console](#).

# Script Generator

With the script generator, you can generate a complete export script or a header library.

To start the script generator, log in to Backoffice and navigate to **System** **Tools** **Script Generator** . A wizard opens that allows you to generate a complete export script or a header library needed for an Export Wizard started from a search result.

# ImpEx import



## IMPEX IMPORT Configuration

Script type:

Migration

Document ID:



Incl. system types:



Languages:

...

Media:

...

Script:

CANCEL

DONE

The options available include:

Option	Description/Comment
Script Type	Select the type of the resulting script. You can use <b>Header library</b> only at the wizard open when using the export action on a search result list. It contains only header definitions. <b>Migration</b> generates a whole export script that you can use at the usual export wizard.
Media	References the generated media containing the generated script after clicking the <b>Save</b> button.
Script	The script generated after clicking the <b>Generate</b> button.
Document ID	If enabled, the document ID mechanism is used for references that can not be specified uniquely by using a combination of attributes not equal to PK. If disabled, the PK attribute is used.

Option	Description/Comment
Incl. system types	The <code>impepx</code> extension adds an attribute <code>system_type</code> to the <code>ComposedType</code> type using that you can mark a type as system type. If you enable the flag only for those types, an export statement is generated that is not marked as system type.
Languages	Here you can restrict the languages for which a localized attribute gets an corresponding header attribute. Assume you have 100 languages installed and you select only one. Then only for this single language does each localized attribute get a header attribute generated, otherwise 100 are generated for each.

## Export CronJob

To start an export using a `ImpExExportCronJob`, log in to Backoffice and go to **System** **Background Processes** **CronJobs** and open the cronjob wizard:

Filter Tree entries

SEARCH

1 / 2 63 items

	Job definition
Cart Removal Cronjob	solrIndexerJob
Cart To OrderCronJob	backofficeSolrIndexerDeleteJob
Clean up Cronjob	cleanUpFraudOrderJob
Cleanup...	oldCartRemovalJob
Planned catalog version...	orderStatusUpdateCleanerJob
CompositeCronJob	productExpressUpdateCleanerJob
Create audit report cron job	sync powertoolsContentCatalog:Staged-
Cs Ticket Stagnation Cron Job.	sync powertoolsProductCatalog:Staged-
Personalization process...	sync powertoolsContentCatalog:Staged-
Personalization results...	sync powertoolsProductCatalog:Staged-
Excel Import Cron Job	sync powertoolsContentCatalog:Staged-
Export Data CronJob	sync powertoolsContentCatalog:Staged-
Geocoding Cron Job	sync powertoolsProductCatalog:Staged-
ImpEx Export Cronjob	sync powertoolsContentCatalog:Staged-
ImpEx Import Cronjob	sync powertoolsProductCatalog:Staged-
Media Folder Structure...	sync powertoolsContentCatalog:Staged-
Media process job detail	sync powertoolsProductCatalog:Staged-
Move Medias CronJob	sync powertoolsContentCatalog:Staged-
Old Cart Removal Cronjob	sync powertoolsProductCatalog:Staged-
Old payment subscription...	sync powertoolsContentCatalog:Staged-
Order Scheduled Cronjob	sync powertoolsProductCatalog:Staged-
Order Status Update Cleaner...	sync powertoolsContentCatalog:Staged-
OrderTemplate To Order Cronjo b	sync powertoolsProductCatalog:Staged-
Product Express Update...	sync powertoolsContentCatalog:Staged-
Remove catalog version	sync powertoolsProductCatalog:Staged-
RemoveItemsCronJob	sync powertoolsContentCatalog:Staged-
Remove Orphaned Files CronJob	sync powertoolsProductCatalog:Staged-
Rule Engine Cron Job	sync powertoolsContentCatalog:Staged-
Cs Session Events Removal...	sync powertoolsProductCatalog:Staged-
Site Map Media Cron Job	sync powertoolsContentCatalog:Staged-

The screenshot shows the SAP Fiori interface with the Administration tab selected. On the left, there's a sidebar with 'Internationalization' and 'Marketing' sections, and a 'SAVED QUERIES' section indicating 'No queries'. The main area displays a list of cronjobs:

- Cronjob for solr indexer
- Solr Index Optimization...
- SolrQueryStatisticsCollecto...
- Solr update stopwords CronJob
- Solr update synonyms CronJob
- Planned item synchronization
- Uncollected Orders Cronjob
- Workflow

On the right, there are several status messages in blue text:

- sync powertoolsProductCatalog:Staged-...
- sync powertoolsContentCatalog:Staged-...
- sync powertoolsProductCatalog:Staged-...
- sync powertoolsContentCatalog:Staged-...
- sync powertoolsProductCatalog:Staged-...

In the wizard, you can fill in the fields for the basic parameters (for example the code). Save your cronjob and open it. The parameter details are explained below.

## Administration Tab

The **Administration** tab contains important administrative options:

Administration ▾

foo SEARCH

**ImpEx-Export : fooExportCronjob - UNKNOWN - UNKNOWN**

trash refresh undo redo print REFRESH SAVE

LOG TASK RUN AS TIME SCHEDULE SYSTEM RECOVERY ADMINISTRATION

### ESSENTIAL

Code	Current status	Job definition	Last result
fooExportCronjob	NEW	ImpEx-Export	N/A
Timetable	Last start time	Enabled <span>?</span>	Last end time
Not scheduled	<span>calendar</span>	<input checked="" type="radio"/> True <input type="radio"/> False	<span>calendar</span>

### METADATA

PK	Type	Time created	Time modified
8796256993781	ImpExExportCronJob	Mar 5, 2018 3:26:12 PM	Mar 5, 2018 3:26:12 PM
Owner	<span>...</span>		

### CHANGES

Last changes

emailNotificationTemplate,single...
<span>...</span>

### UNBOUND

Comment character <span>?</span>	Converter <span>?</span>	Code <span>?</span>	Target (Data) <span>?</span>
#	<span>...</span>	dataexport_fooExportCronjob	<span>...</span>
Encoding <span>?</span>	Export	Export-Template <span>?</span>	Fieldseparator <span>?</span>
UTF-8	<span>...</span>	<span>...</span>	;
Exported Items <span>?</span>	Overall Items <span>?</span>	Filtered items <span>?</span>	ImpEx content <span>?</span>
<span>...</span>	<span>...</span>	<span>...</span>	<span>...</span>
Code <span>?</span>	Target (Media) <span>?</span>	Mode <span>?</span>	Escape character <span>?</span>
mediasexport_fooExportCronjob	<span>...</span>	(Re)Import Strict	"
Report	Export as	ActiveCronJobHistory	Documents
<span>...</span>	<input type="radio"/> True <input checked="" type="radio"/> False	<span>...</span>	<span>+</span> Create new Output Docume <span>...</span>
Assigned Cockpit Item Templates	Changes	Comments	CronJobHistoryEntries
<span>...</span>	<span>+</span> Create new Change descrip <span>...</span>	<span>...</span>	<span>+</span> Create new CronJobHistory <span>...</span>
Error mode	Maximum number of rows	Request Abort	Request abort step
<span>...</span>	1000	<input type="radio"/> True <input checked="" type="radio"/> False <input type="radio"/> N/A	<input type="radio"/> True <input checked="" type="radio"/> False <input type="radio"/> N/A

Ignore

T000

TRUE

RAISE

TRUE

RAISE

TRUE

RAISE

TRUE

Is blocked for processing

 True
  False
  N/A
**COPIES**

Dependent catalog versions

...

Source catalog versions

...
**ADDITIONAL ATTRIBUTES**

Once-only executable

 True
  False
Delete (after termination) ?
 True
  False
Retry execution, if the job can not be  
executed yet
 True
  False

Alternative Data Source ID

Here are the details:

Option	Description/Comment
ImpEx Content	The export script media used for defining the export. It has to contain a sequence of header lines intercepted by bean shell commands for defining the set of items to export using the related header line. For further information refer to the ImpEx reference.
Mode	Selects the validation mode to use for export.
Export Template	Selects a media containing the export template, which is a kind of ImpEx script containing only header lines. This media is merged to an ImpEx script containing headers, and the related set of item PK's to export by an export wizard. So it does not make sense to configure it directly at the cronjob instance.
Code (Target Data)	The code of the resulting media containing the exported data.
Field Separator	Field separator to use for exported data.
Escape Character	Escape character to use for exported data.
Comment Character	Comment character to use for exported data.
Code (Target Media)	The code of the resulting media containing the exported medias.
Converter	Converter to use for preprocessing the resulting medias.
Target (Data)	The media containing the exported data as ZIP file.
Target (Media)	The media containing the exported medias as ZIP file.
Export	The export container holding the advanced settings as well as the resulting export medias.
Report	Resulting media of used Converter.
Once-Only Executable	If set, the cronjob can be performed only once. If you disable this flag do not forget to delete the reference to the working media at restart of a cronjob, otherwise the work media are used again instead of the newly configured import file.

Option	Description/Comment
Delete (after termination)	The cronjob is deleted after successful termination.
Error Mode	What should be done on an error, FAIL the job or IGNORE the error? PAUSE mode is not supported by an export job.

## Log Tab

At the log tab, you can find all about status information.

The screenshot shows the SAP Fiori Log Tab interface. At the top, there's a search bar with a yellow 'SEARCH' button and a navigation bar with icons for user, refresh, and power. Below the header, the title 'ImpEx-Export : fooExportCronjob - UNKNOWN - UNKNOWN' is displayed, along with 'REFRESH' and 'SAVE' buttons. A navigation bar below the title includes tabs for LOG, TASK, RUN AS, TIME SCHEDULE, SYSTEM RECOVERY, and ADMINISTRATION, with 'LOG' being the active tab.

The main area is divided into sections:

- ESSENTIAL**: Contains fields for Code ('fooExportCronjob'), Current status ('NEW'), Job definition ('ImpEx-Export'), Last result ('N/A'), Timetable ('Not scheduled'), Last start time, Enabled status ('True'), and Last end time.
- LOG**: Contains settings for Log to database ('False'), Log level database ('WARNING'), Logs Count ('5'), Logs Operator ('AND'), Logs Days Old ('14'), Log to file ('INFO'), and Log files ('Create new Log File'). It also includes a note: "For performance reasons job logs are not displayed here. Use Search:" and a 'FIND JOB LOGS FOR CRONJOB' button.
- JOB STEPS**: Shows Current step, Processed steps, and Pending steps, each represented by a list of items with ellipsis buttons.

Option	Description/Comment
Entire Log	Displays the first 500 log entries of the database (not the logs stored at the file).
Log to Database	Enables the logging to database.
Log Level Database	Sets the log level for database logging.
Log to File	Enables logging to file.

Option	Description/Comment
Log Level File	Sets the log level for file logging.
Log Files	References the created log files (not database log).

## Export Wizard

Backoffice offers a wizard that enables you to configure and start an export.

To start an export, log into Backoffice and navigate to **System > Tools > Export**. A wizard opens that allows you to set the basic parameters for a **CronJob** that creates the exported data.

### Configuration Tab

The main tab of this wizard specifies the source medias for the import and some basic settings.

The following screenshot shows the configuration tab of the wizard. In this tab, you have to select or upload an export script media that can be either generated by the Script Generator or created manually. At the Script text box, you can see the loaded script and edit it, or you start typing a script from scratch. Remember that you always have to click the **Save** button after you change the script in the text box to write it back to the selected file (if no file is selected, a new one is created). After you have optionally validated the script using the **Validation** button, you can click **Continue** to go to the **Advanced configuration** tab or you click the **Start** button.

### ImpEx Export

X

IMPEX EXPORT Configuration > IMPEX EXPORT Advanced configuration > IMPEX EXPORT Results

Script:

VALIDATE
SAVE

ImpEx Media:

...

Execute asynchronously:

Mode:

CANCEL
NEXT

Available options:

Option	Description/Comment
Script	Displays the content of the selected <b>ImpEx Content</b> media. You also can edit the content of the text box, where afterwards the <b>Save</b> button has to be clicked for saving the edited text to the <b>ImpEx Content</b> media (if no media is selected a new one is created).
ImpEx Media	The export script media to use for defining the export.
Mode	Selects the validation mode to use for export.

## Advanced Settings Tab

After clicking the **Next** button, you get to the **Advanced Configuration** tab. Here you can configure the resulting medias containing the exported data and decide to convert a csv file into a zip file.

Option	Description/Comment
Encoding	Encoding to use for exported data
Field Separator	Field separator to use for exported data
Escape Character	Escape character to use for exported data
Comment Character	Comment character to use for exported data
Export as zip	Allows you to optionally convert the CSV file to a ZIP file after export.

## Result Tab

Once you click **Start**, the CronJob starts and the following **Result** tab appears. Here you have a direct link to the performed cronjob as well as the resulting medias.

## ImpEx Export



**IMPEX EXPORT**  
Configuration

**IMPEX EXPORT**  
Advanced configuration

**IMPEX EXPORT**  
Results

Export status:

Sync import failed

Performed CronJob:

ImpEx-Export : 000003UX - FINISHED - FAILURE

ImpEx export logs:

Exported Zip File:

**DOWNLOAD**

Exported Media Zip:

**DOWNLOAD**

**DONE**

## Export Through Administration Console

You can perform export using Impex export page in SAP Commerce Administration Console.

### i Note

The alternative to exporting data of Platform described here is only for development systems and **has** to be disabled in productive systems. Because it is only an easy way of testing an ImpEx script for development without any configuration options, it is not preferred and supported at all.

To navigate to **ImpEx Export** page, in the SAP Commerce Administration Console click on the **Console**. Select the **ImpEx Export** option. It guides you to the page that is structured in a file-based export section and a content-based export section, analogous to the import case.

Using the file-based export at the top of the page, you can simply select an **ImpEx** script covered in a file by clicking the **Browse** button. Afterwards you have to click the **Export File** button and a **CronJob** based export is started with the given file as the input source. The logs of the **CronJob** are written to the result page and to the console. If the export was successful, a green text signals this, if not - a red result text is displayed. The error cause can be found in the logs. The stack trace of the occurred error can only be found at the console. The result text contains a link to the created result medias.

The second way of exporting is the use of the content-based export at the bottom of the page. Here, the **ImpEx** script has to be entered directly at the text box. You can select the validation mode to use for export and next to the **Export Content** button you have the possibility to validate the script first by clicking the **Validate Content** button.

## ImpEx Media

An ImpEx media represents in general a CSV/impex file or a ZIP archive containing CSV/impex files. They are used only by the ImpEx extension for import and export processes.

Because these medias contain CSV data, it provides some additional attributes specific to CSV data.

Properties for ImpEx processing

Import file in ZIP:	<input type="text"/>	Escaping character:	<input type="text"/>
Separator character:	<input type="text"/> ;	Encoding:	<input type="text"/> windows-1252
Comment character:	<input type="text"/> #	Lines to skip:	<input type="text"/> 0
Remove on success:	<input type="radio"/> Yes <input checked="" type="radio"/> No		

◀ ▶

Option	Description/Comment
Import file in ZIP	In case the media contains a ZIP archive, this attribute selects the archive entry containing the main import script (used for import).
Separator character	Character used in this file for separating the data cells.
Escaping character	Character used in this file for escaping.
Comment character	Character used in this file for introducing comments.
Encoding	Sets the encoding used in this file.
Remove on success	If set, the media is deleted in case it was part of an import and the import was successful.
Lines to skip	Number of lines which have to be skipped before parsing the first CSV-line (in case you have a CSV-header at first line for example).

## Header Library

The header library types extend the **ImpExMedia** type but does not provide any additional features. It is just a kind of marker saying that it is a header library. They should only be used at the first tab of the export wizard that is opened when using the export action at a search result list.

A header library is a kind of export script only containing header definitions. Only by merging it with a list of items using the export action a real export script can be produced from.

## ImpEx Export Media

This kind of **ImpExMedia** marks the result of an export. There is no additional feature set or functionality.

## Related Information

[Performing Multithreaded Catalog Synchronization in Backoffice](#)

## ImpEx Import - Best Practices

Importing data is a common project task. If data being imported surpasses a certain amount, the import time becomes a factor to consider. For example, the performance of some business methods is not suitable to import mass data. The reason is that some checks or implemented service code—although very useful for normal requirements—do not have a performance improved for importing mass data. A typical case is importing Customers into a system, thus this article refers to this example.

## Importing via Code

Generally, it is faster to create items through the `newInstance()` method and pass an attribute map with the values.

### Create Customers

When creating a customer using service layer all you need to do is create model instance, then fill it with proper attributes and save model:

```
final CustomerModel customer = modelService.create(CustomerModel.class);
customer.setUid("newCustomer");
customer.setGroups(Collections.emptySet());
modelService.save(customer);
```

### Avoid Methods with Checks

The `Customer` instance has to be added to a group, normally the `customerGroup`. One possibility to do this is by the `addMember()` method on the `Customer` item. This method, however, is checking if the customer is already added to the group, a feature most people don't need in an import situation.

Therefore, the following syntax is not suggested:

```
customerGroup.addMember(<customer>)
```

You should instead insert the relation data directly through the LinkManager (manager for relations). For example, by adding the user to the `PrincipalGroupRelation`.

```
LinkManager.getInstance().createLink(
    de.hybris.platform.jalo.security.Principal.PRINCIPAL_GROUP_RELATION,
    <customer>,
    <customerGroup>, 0)
```

## Importing via ImpEx

In the case of using ImpEx, items typically are created with good performance. Improvement can be achieved when setting the group relation. The best way here is also to separate the user import from setting the group relation.

In the first step, find the commonly used:

```
INSERT_UPDATE Customer;uid[unique=true];groups(uid)
    ;testuser;customerGroup;
```

When you find it, exchange it with the following piece of code:

```
INSERT_UPDATE Customer;uid[unique=true];groups[ignorenull=false,default=]
    ;testuser;;
```

Remember to keep the groups column empty. In other words, just separate the processes **User import** and **fill the relation**. The reason for that is that the `createItem` method of the Customer otherwise automatically inserts the created Customer into the `customerGroup`, which is implicitly done by an `addMember()` call.

In the second step then, insert the relations values in the **PrincipalGroupRelation**:

```
INSERT_UPDATE PrincipalGroupRelation;source(uid)[unique=true];target(uid)
;testuser;customergroup;
```

For technical details on using ImpEx as well as an explanation of the modifiers used in the example above, see [Impex Extension - Technical Guide](#).

## ImpEx Distributed Mode

The Distributed ImpEx engine enables you to import Platform items from huge and complex external files (for example, files that contain many dependencies between items), and at the same time it delivers exceptional performance. For more information, see [About Distributed ImpEx](#).

### [About Distributed ImpEx](#)

Distributed ImpEx leverages the existing ImpEx framework to parse and analyze input, and dump unresolved value lines. It also leverages ServiceLayer for persistence, as well as TaskEngine to process single batches of data.

### [Executing Import Programmatically](#)

Enabling data import in the distributed mode programmatically works similarly as in classical ImpEx.

### [Executing Import from Administration Console](#)

To import data in the distributed mode using Administration Console, use the same Administration Console page that the classical ImpEx uses.

### [Executing Import from Backoffice](#)

To import data in the distributed mode, use the standard [ImpEx import wizard](#).

### [Executing Import on Selected Node Groups](#)

Distributed ImpEx uses TaskEngine internally, which was designed to work well in a cluster environment. This enables you to choose which node group to execute import on.

### [Execution Results and Logs](#)

Backoffice enables you to search for logs from a given CronJob.

### [Using ServiceLayer Direct](#)

Distributed ImpEx allows you to use ServiceLayer Direct.

### [Removing Stale Items](#)

SAP Commerce allows you to remove stale output items that have been created in a successful Distributed ImpEx process.

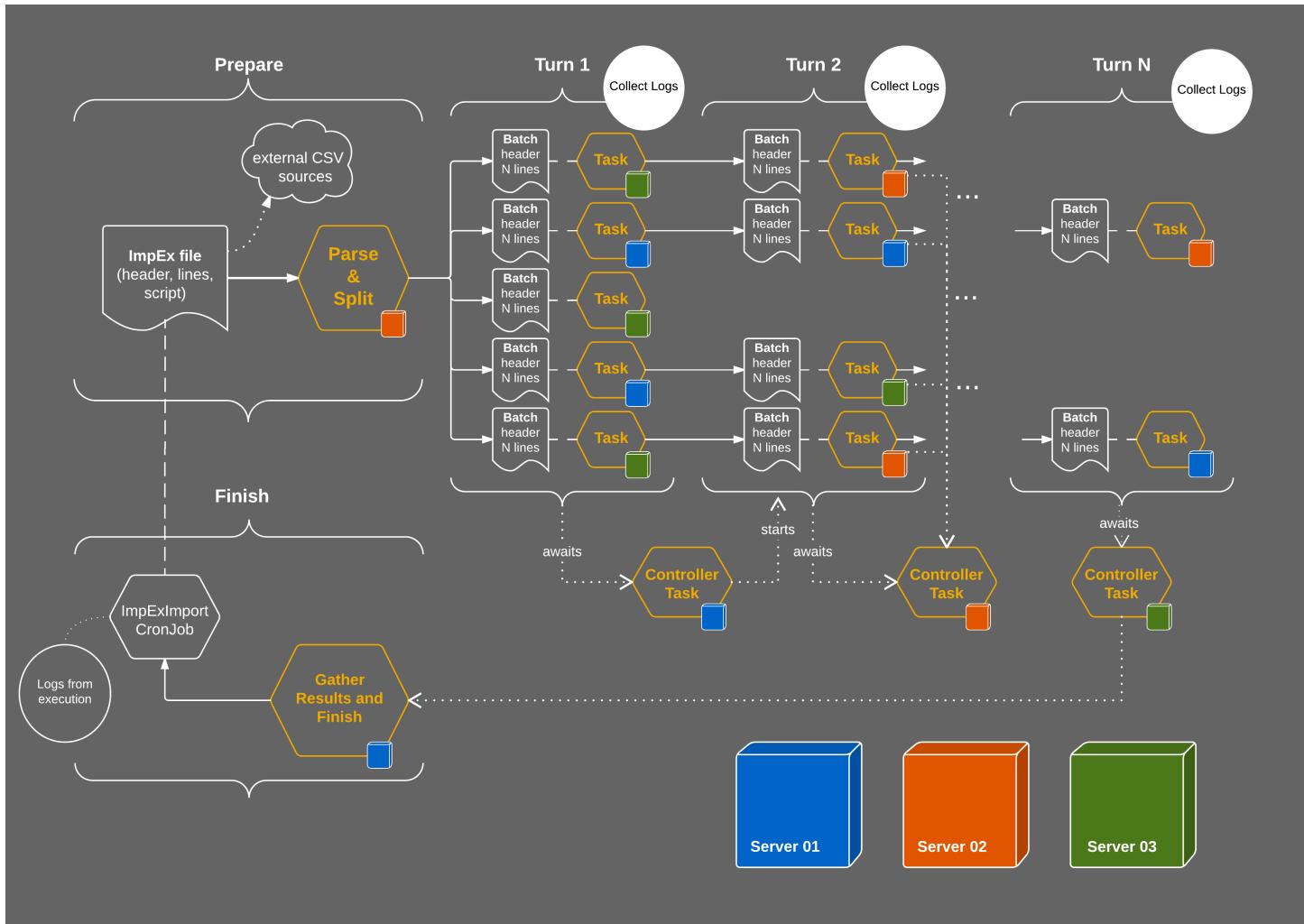
## About Distributed ImpEx

Distributed ImpEx leverages the existing ImpEx framework to parse and analyze input, and dump unresolved value lines. It also leverages ServiceLayer for persistence, as well as TaskEngine to process single batches of data.

Importing data using Distributed ImpEx (the **distributed mode** for short) consists of three phases:

1. **Prepare and split** phase
2. **Single task execution** phase
3. **Finish** phase

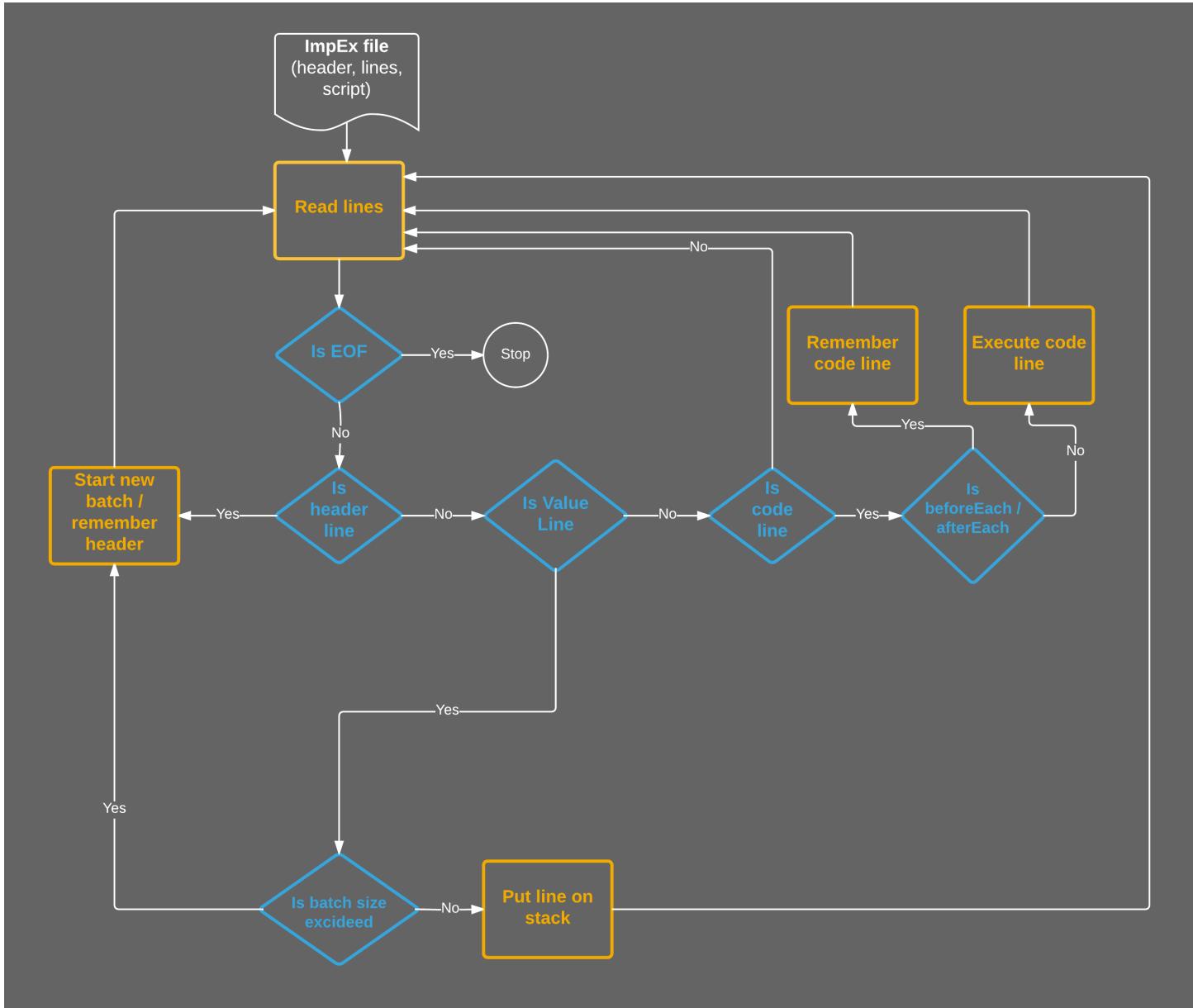
The whole process looks as follows:



## Prepare and Split Phase

In the **Prepare and Split** phase, a large ImpEx file is read line by line and split into batches.

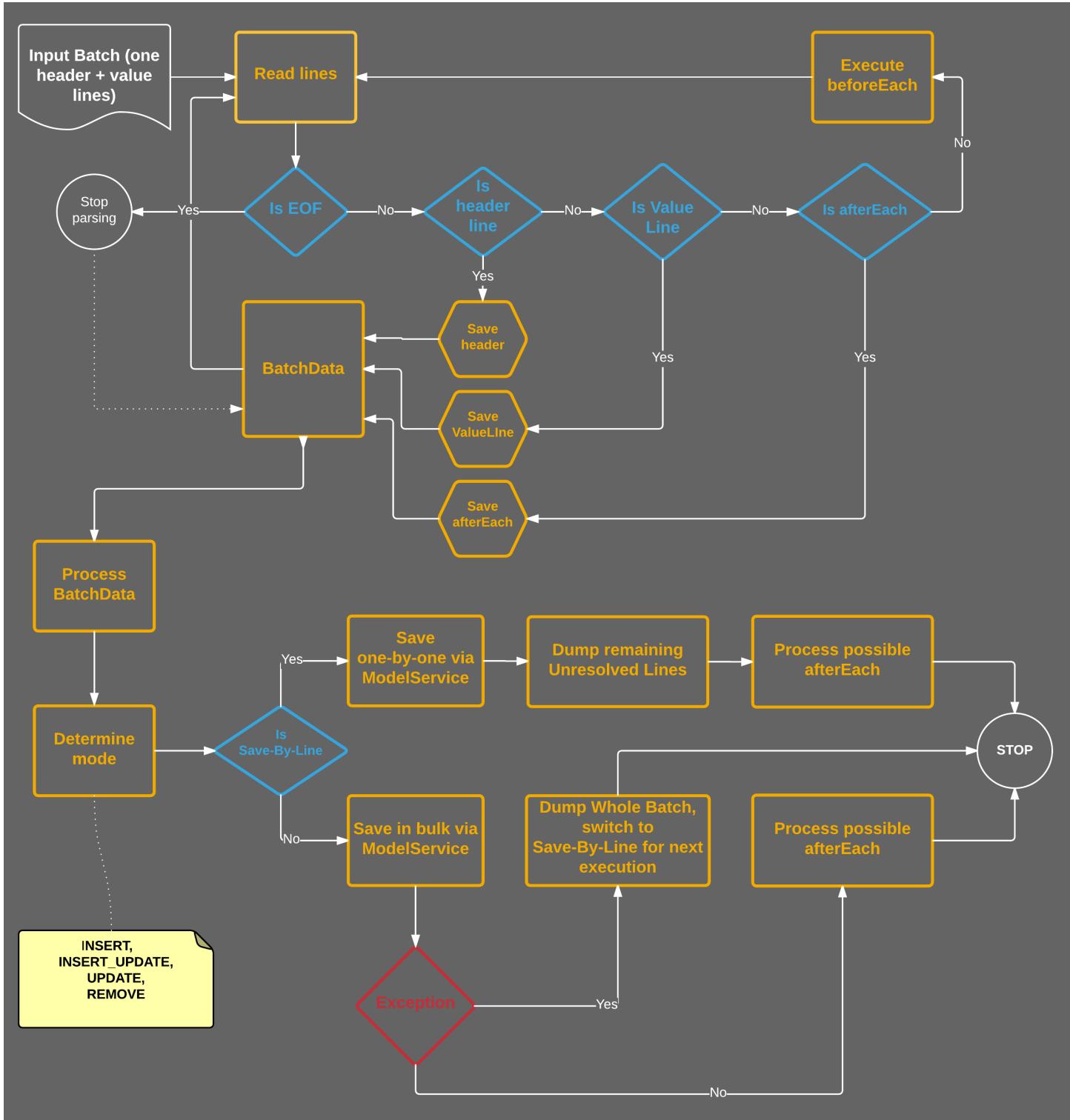
Each header starts a new batch. Value lines are added to a batch until the batch size reaches its limit. You can configure a batch size with the `<impeкс.distributed.batch.size>` property. The default value is 100. All macro definitions declared in an input file are executed, so single batches contain translated values. All possible scripting lines are also executed except for `beforeEach:` and `afterEach:` statements. Those are passed on to single batches. If a batch size is exceeded, a new batch is started with the last read header. The whole process is shown in the diagram:



## Single Task Execution Phase

In the **Single Task Execution** phase, TaskEngine executes each batch individually. That means that a single task is executed in one thread but, on the whole, all batches are spread across the entire cluster. Thus they are executed in parallel.

Execution of a single task in the distributed mode doesn't differ from the classical, single threaded ImpEx process. A batch always contains one header only, and a number of value lines, plus possibly the `afterEach:` and `beforeEach:` expressions. An input batch is in the form of a **String**. Hence, it is read line by line, parsed, and finally converted to **Models**. Models are then persisted into the database. Note that the purpose of the process is to try to transform every value line into Models and finally persist them in one go. If this fails, the process switches to the **save-by-line** mode. The whole process looks as follows:



## Finish Phase

In this phase the whole process ends. A clean-up work is done (for example possible `<documentId>`'s are removed) as well as the status and the result of a resulting CronJob is set.

## Limitations

Distributed ImpEx was built with strong backward compatibility. Almost all features known in the classical ImpEx work in the distributed mode. Only some aren't supported. An input script can contain any number of lines and headers as in a classical ImpEx file. If you want to use scripting inside an input file, note that there are some limitations:

- `if`: and `endif`: scripting statements aren't supported

- `beforeEach`: and `afterEach`: scripting statements are executed in a single batch execution phase
- `beforeEach`: and `afterEach`: scripting statements must be placed after the header definition
- Remaining scripting statements are executed in the splitting phase
- Headers that contain any flags related to Jalo persistence aren't supported
- The Processor modifier is not supported

## Parameters

### Querying the Database for a Single Impex Line

The `impex.distributed.query.for.each.line` flag forces the distributed impex mechanism to read an impex file line by line and retrieve items from the database for the current line only.

When necessary, Platform can switch to reading impex lines one by one on its own but if you think that it is better for Platform do use this behavior from the beginning of an operation, set the flag to `true`.

## Executing Import Programmatically

Enabling data import in the distributed mode programmatically works similarly as in classical ImpEx.

To configure import, use `ImportService` and `ImportConfig` objects, and enable the `distributedImpex` flag:

```
// assuming we have an ImpExResource object that points to an import file on classpath
final ImpExResource importFile;
final ImportConfig config = new ImportConfig();
// enable Distributed ImpEx
config.setDistributedImpexEnabled(true);
config.setScript(importFile);

// perform import
final ImportResult importResult = importService.importData(config);
```

Since we use the `ImportService` and `ImportConfig` API, below we provide the meaning of some of the `ImportConfig` object flags. By default, the `synchronous` flag of `ImportConfig` is set to `true`. In classical ImpEx, this flag is used to inform the underlying CronJob to run as synchronous or asynchronous. Distributed ImpEx is an asynchronous process by its nature. It splits work into batches and uses TaskEngine to process them all on available nodes. Setting this flag to `true` instructs `ImportService` to try to wait for a process to finish and then return `ImportResult`. Because Distributed ImpEx was built to import large volumes of data, it is recommended to set this flag to `false`.

The following flags of an `ImportConfig` object are useful only for the distributed mode, and are ignored in the classical mode:

Flag	Explanation
<code>distributedImpexEnabled</code>	Enables Distributed ImpEx.
<code>distributedImpexProcessCode</code>	Enables you to set a custom process code for execution.
<code>distributedImpexLogLevel</code>	Enables you to set Log Level for Distributed ImpEx.
<code>sldForData</code>	Enables you to set the Service Layer Direct mode for import.

Flag	Explanation
nodeGroup	Enables you to set a cluster node group that handles the import process.
removeOnSuccessForDistributedImpex	Removes stale output items that have been created during a successful synchronous Distributed ImpEx process.

The following flags of an `ImportConfig` object are ignored in the distributed mode:

Flag	Explanation
maxThreads	Using this flag is pointless since TaskEngine processes all the work.
removeOnSuccess	Distributed ImpEx creates a CronJob instance but only as a container for logs and additional information such as status, so for the time being the CronJob isn't removed after the process has finished.
hmcSavedValuesEnabled	Not used.
dumpingEnabled	Not used.
failOnError	Using this flag is pointless here. Distributed ImpEx works on the whole cluster so having one batch that has failed doesn't affect the process of importing the remaining batches.
legacyMode	Distributed ImpEx only works in the Service Layer mode.
mainScriptWithinArchive	At the moment of writing this document, Distributed ImpEx doesn't allow you to use zip archives as input.

## Executing Import from Administration Console

To import data in the distributed mode using Administration Console, use the same Administration Console page that the classical ImpEx uses.

### Context

Choose a file that includes data you want to import and start importing.

### Procedure

1. Log into Administration Console.
2. Hover the cursor over **Console** to roll down a menu bar.
3. In the menu bar, click **ImpEx Import**.

**ImpEx Import** page displays.

4. Switch to the **Import script** tab.
5. Click **Choose file** and load your file.
6. Tick the **Distributed mode** option.

This page provides ImpEx import functionality. You can import a script file or paste a script and validate it before the import.

**Note**  
Legacy mode  
Impex import works on Service Layer. If you select this option, then Jalo Layer is used.

**Info**  
Fullscreen mode  
Press F11 when cursor is in the editor to toggle full screen editing. Esc can also be used to exit full screen editing.

See also in the hybris Wiki

- [impex Extension - Technical Guide](#)

7. Click **Import**.

You should get a message about import status.

## Available Options

The following options are available in Administration Console for the distributed mode only:

Option	Description
Distributed mode	Enables the distributed mode.
Direct persistence	Enables the Service Layer Direct mode. For more information, see <a href="#">Using ServiceLayer Direct</a> .

## Executing Import from Backoffice

To import data in the distributed mode, use the standard [ImpEx import wizard](#).

### Context

Choose a file that includes data you want to import and start importing.

### Procedure

1. Log into Backoffice.
2. Open [ImpEx import wizard](#).

a. Click **System**.

a. Click **Tools**.

a. Click **Import**.

**ImpEx import wizard** opens.

3. Choose the data file to upload.

The screenshot shows the 'ImpEx import' configuration interface. At the top, there are three tabs: 'ImpEx Import Configuration' (highlighted in blue), 'ImpEx Import Advanced configuration', and 'ImpEx Import Results'. Below the tabs, the 'Choose media:' section contains a search bar and a list item '00000001 - default catalog : Online'. At the bottom of this section are four buttons: 'upload' (dark blue), 'create' (light gray), 'preview' (light gray), and 'reset' (light gray). The 'Locale settings:' section below it has a search bar. Under 'Import file in ZIP:', there is a large input field with a dropdown arrow and two buttons at the bottom right: 'Cancel' (gray) and 'Next' (blue).

a. Click **upload** and choose your file.

b. Click **create**.

4. Click **Next** to switch to the **Advanced Configuration** tab.

5. Select **Distributed Mode**.

**ImpEx import**

**ImpEx Import Configuration**    **ImpEx Import Advanced configuration**    **ImpEx Import Results**

**Media-Zip:**

**Upload new media:**

**Distributed Mode:**

**Direct persistence:**

**Node Group:**

**Log Level:**

**Strict import mode:**

**Execute asynchronously:**

**Allow code execution from within the file:**

6. Click **Start**.

The view switches to **ImpEx Import Results**.

7. Click **Done**.

## Available Options

In Backoffice, the following options are accessible from the **ImpEx import wizard**:

Option	Description
Direct persistence	Enables ServiceLayer Direct. For more information, see <a href="#">Using ServiceLayer Direct</a> .
Node Group	Enables you to set a cluster node group that handles the import process.
Log Level	Enables you to set Log Level for Distributed ImpEx
Strict import mode	When <b>enabled</b> , this option causes import to work in the strict mode (default is relaxed).

Option	Description
Execute asynchronously	When <b>disabled</b> , this option causes the wizard to try to wait for the process to finish.
Allow code execution from within the file	Enables code execution in the data import file.

## Executing Import on Selected Node Groups

Distributed ImpEx uses TaskEngine internally, which was designed to work well in a cluster environment. This enables you to choose which node group to execute import on.

Assume you have 2 nodes:

- **NodeBackend** with the following node group settings:

```
cluster.node.groups=backend
```

- and **NodeFrontend** configured as follows:

```
cluster.node.groups=frontend
```

As long as you don't set the `nodeGroup` property of `ImportConfig`, you are able to execute import on any node - in this case on **NodeBackend** or **NodeFrontend**.

However, if you specify a node group in `ImportConfig`, you allow Distributed ImpEx to execute the import process only on a node with a matching node group:

```
final ImportConfig config = new ImportConfig();
config.setDistributedImpexEnabled(true); //enable distributed impex
config.setNodeGroup("backend");
```

In the provided example only **NodeBackend** is able to pick import process for execution. **NodeFrontend** cannot perform any import.

You may as well use the [ImpEx import wizard](#) to specify a node group.

## Execution Results and Logs

Backoffice enables you to search for logs from a given CronJob.

### Context

For backward compatibility, an instance of `ImpExImportCronJobModel` is available as a result of data import execution. This CronJob contains all the logs from a given execution, as well as its status.

#### Caution

An instance of this CronJob **must not** be executed or scheduled for further execution.

To look up logs from a particular Distributed ImpEx import execution, follow the procedure.

## Procedure

1. Log into Backoffice.
2. Look up a CronJob from a given import execution:
  - a. Click **System**.
  - a. Click **Background Processes**.
  - a. Click **CronJobs**.

You can see a list of CronJob instances from particular import executions.

3. Click the CronJob instance you're interested in.

The CronJob's editor area opens. Here you can find the status of the CronJob execution, as well as a list of items containing the logs:

## Using ServiceLayer Direct

Distributed ImpEx allows you to use ServiceLayer Direct.

### i Note

The examples provided here apply to Distributed ImpEx only. All the SLD settings are ignored in the classical ImpEx mode.

## Enabling ServiceLayer Direct for All Imports

You can enable ServiceLayer Direct in the distributed mode for the whole process of importing data, for all imports. To enable SLD (direct persistence) and the distributed mode, configure an **ImportConfig** object as follows:

```
final ImportConfig config = new ImportConfig();
config.setDistributedImpexEnabled(true); //enables distributed impex
```

```
config.setSldForData(true); //enables direct persistence mode
```

### i Note

By default, the global persistence mode (`persistence.legacy.mode`) is used if you don't enable the direct persistence mode.

You can enable SLD in Backoffice or Administration Console. The Backoffice ImpEx import wizard contains the **Distributed Mode** option. After you choose it, the **Direct persistence** option becomes available. Note that when you choose **Distributed Mode**, some classical ImpEx input fields become unavailable (as not relevant for the distributed mode). You can find the same options in Administration Console.

## Switching from Legacy Mode to ServiceLayer Direct for One Header

You can have the whole distributed import process working in the legacy mode and at the same time force it to switch to the direct persistence mode for a selected batch, or header. To use that feature, set the `sld.enabled` modifier in a header of a chosen import file to `true`:

```
INSERT_UPDATE Title[sld.enabled=true];code[unique=true]
;foo_sld_forced_by_header
;bar_sld_forced_by_header
```

Setting the modifier to `sld.enabled=true` means that both titles are to be imported using the SLD mode, even if the global switch, or the flag in `ImportConfig` is set to the legacy mode. Compare with the `persistence.legacy.mode` flag mentioned above.

## Removing Stale Items

SAP Commerce allows you to remove stale output items that have been created in a successful Distributed ImpEx process.

You can configure SAP Commerce for programmatically run import to remove stale output items of the `ImpexImportCronJob`, `DistributedImportProcess`, `ImportBatch`, `ImportBatchContent` type that have been created during successful synchronous Distributed ImpEx processes. For more information, see [Executing Import Programmatically](#).

You can also use the following property if you haven't configured the `removeOnSuccessForDistributedImpex` property of the `ImportConfig` object. When you enable this property, stale items created during import, for example in Administration Console and Backoffice, are also removed.

```
impex.import.service.distributed.synchronous.removeonsuccess.enabled=true
```

This property is disabled by default.

Use the following property for `ImportBatchContent` items that have been created during a successful synchronous or asynchronous Distributed ImpEx process:

```
impex.distributed.importbatchcontent.removeonsuccess=true
```

This property is enabled by default. This property ensures that stale `ImportBatchContent` items created during import, for example in Administration Console or Backoffice, are also removed.

## ImpEx FAQ

This document provides information about issues associated with the **impex** extension.

## How Can I Get More Information on Why an ImpEx Script Doesn't Work?

I've got an ImpEx script that fails to execute every now and then, but I do not get any error messages on the server log. How can I get more information on why the script doesn't work?

### Technical Background

The **impex** extension uses CSV-based files, that is ImpEx scripts, to create instances of types in SAP Commerce. Depending on the CSV file contents, running an import with the **impex** extension may fail with an exception or may fail "silently", that is, the import does not work, but no exception is thrown.

A way of getting a more extensive log of what an ImpEx script does is to find the related (import/export) cronjob. A CronJob has a log file which keeps track of what happened during the CronJob execution.

### Solution

This section gives a tutorial on creating a CronJob that runs an ImpEx script that imports data so that the ImpEx script execution is logged.

1. Log into Backoffice.
2. Navigate to **System** **Background Processes** **Cronjobs**.
3. Find your cronjob, for example **fooCronjob**.
4. Click the **Log** tab to check the CronJob execution log.

If the log does not provide sufficient information, please also refer to the log file(s) listed by the **Log files** attribute.

## Exception: Item Reference for Attribute? Does Not Provide Enough Values at a Position?

The exception says that you have to give a composed item reference value but a part of this composed value is missing.

Assume you have the following example script where the used catalog version already exists:

```
INSERT_UPDATE category; code[unique=true]; supercategories(code,catalogVersion(catalog(id[default='cl
; CL2000;           ; clothescatalog:Staged;shoes;
; CL2100; CL2000; clothescatalog:Staged;
```

Then you have to pay attention when defining the default values for the **catalogVersion** within the **supercategories** definition. It does make a difference whether you define the **default='clothescatalog'** for the **id** attribute of the referenced catalog or the **catalog** attribute directly:

### will work

```
...;supercategories(code,catalogVersion(catalog(id[default='clothescatalog']),version[default='Staged'
```

### will not work

```
...;supercategories(code,catalogVersion(catalog(id)[default='clothescatalog']),version[default='Staged'
```

ImpEx tries to compose a value from the default values provided. If a default for **catalog** and **version** is provided, it gets the default for the entire **catalogVersion** attribute. With this information, it tries to create a value for the **supercategories**

attribute together with the specified value for the code attribute. If no value is provided, as shown in the example above for category with code '2100', the resulting item expression value is not valid.

If you define the default value for the id attribute, you do not provide a default for the catalog and so no default catalogVersion can be composed.

## Why Are Bean Shell Statements Especially Before\_Each Statements Not Dumped for the Second Run?

It is an unsolvable problem to decide if a bean shell statement has to be dumped or not and so they are not dumped at all. If you want to use statements which modify the last imported item, you should disable the dumping functionality for being sure the code is executed always the item is imported (during second run the code is not executed).

If you use the bean shell functionality of ImpEx, you can make simple statements like a `System.out.println` which could be dumped without problems, because it has no side effects. If you write unrepeatable statements (something which modifies items and can only be executed one time without side effects) a dumping has serious problems, because the code is executed more than one time. Because ImpEx can not guess which kind of code you have written and if it is repeatable or not, code is not dumped for a second run at all. So if you want to have bean shell code, you should disable the dumping functionality for being sure the import is only tried one time (avoiding a second run where the code lines is not executed).

## Why Is the First Line of My Included File Not Considered During Import?

The CSV request for comment preserves the first line for column comments. ImpEx assumes as default that an included file does only contain data where it is normal, to define at the first line the column qualifiers as comment. So by default the first line of included files is ignored. This default value can be found at the attribute `linesToSkip` of the `ImpExMedia` type. So if you include a file using a `includeExternalDataMedia` method and you want to consider the first line for import, just set this attribute of your included media to 0.

## What Possibilities Do I Have to Give Special Logic for Column Translation?

If you need business logic not implemented in the default translator class specified for an attribute, the following options are available:

- Implement a custom translator. Refer to the [Writing Own Translator](#) section of [ImpEx API](#).
- Implement a custom special translator. Refer to the [Writing Own Special Translator](#) section of [ImpEx API](#).
- Implement a cell decorator Refer to the [Writing Own Cell Decorator](#) section of [ImpEx API](#).
- Use a Bean Shell code. Refer to the [BeanShell](#) section of [ImpEx API](#).
- Implement your own import processor.

## Import Using Backoffice Works But Not Using `createEssentialData`!

**Problem:** When I try to import data using the Backoffice Tools/Import all works well, and import data are present in the system. But when I put them to the extension's manager, the `createEssentialData()` and `createProjectData()` methods data are not imported during initialization.

**Solution:** Check if you have located the data files at the **resources/jar** folder of your extension. If this is the case maybe there is a file at the **resources/jar** of another extension with the same file name. In this case one of the two files is used when calling `getResourceAsStream`.

This is because in case there are multiple files with the same name, even if in different extensions, it is somewhat arbitrary which file is opened.

## JaloItemNotFoundException at Second Pass While Import

```
de.hybris.platform.jalo.JaloItemNotFoundException: item 149469962411550720 not found[HY-0]
    at de.hybris.platform.core.WrapperFactory$2.compute(WrapperFactory.java:365)
    at de.hybris.platform.cache.AbstractCacheUnit.privateGet(AbstractCacheUnit.java:208)
    at de.hybris.platform.cache.AbstractCacheUnit.get(AbstractCacheUnit.java:109)
    at de.hybris.platform.jalo.JaloItemCacheUnit.getCached(JaloItemCacheUnit.java:76)
    at de.hybris.platform.core.WrapperFactory.getCachedItem(WrapperFactory.java:375)
    at de.hybris.platform.jalo.JaloSession.getItem(JaloSession.java:1116)
    at de.hybris.platform.impex.jalo.imp.DefaultExistingItemResolver.findExisting(DefaultExistingItemResolver.java:100)
    at de.hybris.platform.impex.jalo.imp.DefaultImportProcessor.processItemDataImpl(DefaultImportProcessor.java:140)
    at de.hybris.platform.impex.jalo.imp.DefaultImportProcessor.processItemData(DefaultImportProcessor.java:116)
    at de.hybris.platform.impex.jalo.imp.ImpExImportReader.readLine(ImpExImportReader.java:457)
    at de.hybris.platform.impex.jalo.Importer.doImport(Importer.java:239)
    at de.hybris.platform.impex.jalo.Importer.importNext(Importer.java:651)
    at de.hybris.platform.impex.jalo.cronjob.ImpExImportJob.doImport(ImpExImportJob.java:236)
    at de.hybris.platform.impex.jalo.cronjob.ImpExImportJob.performJob(ImpExImportJob.java:201)
    at de.hybris.platform.impex.jalo.cronjob.ImpExImportJob.performCronJob(ImpExImportJob.java:166)
    at de.hybris.platform.cronjob.jalo.Job.execute(Job.java:1043)
    at de.hybris.platform.cronjob.jalo.Job.performImpl(Job.java:677)
    at de.hybris.platform.cronjob.jalo.Job.access$1(Job.java:648)
    at de.hybris.platform.cronjob.jalo.Job$JobRunnable.run(Job.java:566)
    at de.hybris.platform.util.threadpool.PoolableThread.run(PoolableThread.java:86)
```

**Problem:** During import all value lines consisting of references which can not be stored are dumped and processed again in a second pass. If the item can be created already without using the unresolved reference it is created and the PK of the created item is stored to at the dump file. When processing the dump file at the second pass the PK is resolved to the already created item, but here the exception is thrown because it is removed already.

**Solution:** There is another type imported which also has unresolved references. These references can be resolved at the second pass and be set to the related items. Unfortunately the business logic of the attribute setters removes the item of our type.

Example script which causes such exception is:

```
insert_update Website;code[unique=true,allownull=true];rootNavigationElements(code,website(code));
"#% impex.includeExternalDataMedia( ""Website.csv"" , ""UTF-8"" , ';' , 1, -1 );"

insert_update BannerNavigationElement;code[unique=true];parent(code,website(code));webSite(code)[unique];
"#% impex.includeExternalDataMedia( ""BannerNavigationElement.csv"" , ""UTF-8"" , ';' , 1, -1 );"
```

Here the Website lines are dumped because not all root navigation elements are available at this time. The BannerNavigationElement lines are also dumped because the parent navigation elements are not available. For each line, an item is already created because no essential data is missing.

The second pass then starts and the unresolved attributes of the websites are set. The `setRootNavigationElements` method is called and removes the navigation elements already assigned to the website. That are the already created navigation elements.

## Impex Translators with Service Layer

When using translators, note that the prepare interceptors are called before translators. If you require that the prepare interceptor works with translated values, you must implement a solution where the prepare interceptor would be called again after the values are translated.

## Data Report and Audit

The Generic Data Report and the Generic Audit allow you to collect raw data about items and present it in a report.

With the Generic Data Report, you can create reports reflecting the current state of the data. With the Generic Audit, you can show how that data has changed over time.

The topics covered include:

### [Preparing a Report Configuration](#)

To generate a report, create a special configuration to inform Platform which data about items, attributes, and relations from your SAP Commerce you want to collect and show in your report.

### [Generating a Generic Report](#)

After you create and extend a report configuration, you can generate the final report.

### [Field Name Localization](#)

Platform resolves field name localizations for data report views and allows you to customize the resolving algorithm.

### [Generic Audit](#)

Generic audit tracks every persistence action, including creation, modification, and deletion for specified types. The audit is stored as a change log that allows you to see how an item changed over time.

## Preparing a Report Configuration

To generate a report, create a special configuration to inform Platform which data about items, attributes, and relations from your SAP Commerce you want to collect and show in your report.

The configuration uses the XML format. You can create it from one or more XML files, covering one or more extensions. Each of your extensions can provide one or more XML configuration files. Dependent extensions can extend and contribute to the existing configurations provided by base extensions. During an initialization or a system update, all configurations are merged by a special name attribute. They are eventually stored in the database.

## Configuration File Naming Convention

Name your configuration files according to the <nameOfTheExtension>-<uniqueNameOfConfiguration>-audit.xml convention. For example, if your report configuration covers user-related items from a foobar extension, then you can choose a descriptive name for the <uniqueNameOfConfiguration> part - for example userdata. You **must** use the name of your extension for the <nameOfTheExtension> part. As a result, your configuration file name is foobar-userdata-audit.xml.

Store your configuration file in the resources directory of a given extension, in this example - inside the resources directory of the foobar extension.

## Configuration File Merging Mechanism

During an initialization or a system update, all extensions are scanned for report configuration files in the extension dependency order, and existing configurations are merged. Configuration files are merged by the same values of the name attribute found inside the configuration files. So, if your extension depends on another extension, and that extension provides a configuration, your extension can contribute to this configuration. For example, extension B depends on extension A, and extension A provides an a-someconfig-audit.xml configuration. Extension B can provide a b-someconfig-audit.xml configuration and extend the a-someconfig-audit.xml configuration. If both configuration files have the same name values, they are merged, and stored in the database.

For more information about extending configurations, see [Contributing to a Report Configuration](#).

## Creating Report Configuration

Start each configuration with <audit-report-config />, and provide a configuration name.

```
<audit-report-config name="sampleConfig">
</audit-report-config>
```

Write each configuration around one particular item type that your report logic starts from. To produce the whole report, provide a chosen item type as a report `rootType`, for example the `User` type.

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
</audit-report-config>
```

Define how you want to present your root type in the final report view.

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
    <type code="User" displayName="User">
      </type>
    </types>
  </audit-report-config>
```

The type is described through the `<type />` tag. The tag has a `displayName` attribute that enables you to set up a human-readable name for your type. In this case, you will see the name `User` in the report view.

To localize your `User` type name, use the `displayKey` attribute and provide a property key for it. In your `local.properties` file, provide a localization as the key value.

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
    <type code="User" displayKey="type.user.name">
      </type>
    </types>
  </audit-report-config>
```

For more information about how localized field names are resolved in a report view, see [Field Name Localization](#).

So far, the report provides only an empty definition of your item type. To see some actual values in your report, you can add atomic attributes to your configuration.

Use the `<atomic-attributes />` container tag with the `<atomic-attribute />` tag to provide a configuration for each atomic attribute that you want to add to your report.

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
    <type code="User" displayName="User">
      <atomic-attributes>
        <atomic-attribute qualifier="name" displayName="name"/>
        <atomic-attribute qualifier="uid" displayName="uid"/>
      </atomic-attributes>
    </type>
  </types>
</audit-report-config>
```

The `qualifier` attribute of the `<atomic-attribute />` tag refers to a real attribute qualifier in the type definition.

## i Note

The `<atomic-attribute />` tag allows you to set up a localized name of an attribute through the `displayKey` tag attribute. For simplicity, this tutorial uses `displayName` only.

You can now produce your report for the `User` type. The report contains information about the `name` and `uid` attributes.

If you crawl from the `rootType` item (`User` in this case), you can get information about its related items. For example, your report can show information about all addresses, groups, and media items the `User` type has. To do it, use the `<reference-attributes />` container tag to collect all references to other items the `User` type may have. This sample code shows how to collect addresses:

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
    <type code="User" displayName="User">
      <atomic-attributes>
        <atomic-attribute qualifier="name" displayName="name"/>
        <atomic-attribute qualifier="uid" displayName="uid"/>
      <atomic-attributes>
        <reference-attributes>
          <reference-attribute qualifier="defaultPaymentAddress" displayName="defaultPaymentAdr
        </reference-attributes>
      </atomic-attributes>
    </type>
  </types>
</audit-report-config>
```

By using the `type` attribute in `<reference-attribute />` tag, you define what type your referenced attribute is.

Add the `Media`, and `PrincipalGroup` references.

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
    <type code="User" displayName="User">
      <atomic-attributes>
        <atomic-attribute qualifier="name" displayName="name"/>
        <atomic-attribute qualifier="uid" displayName="uid"/>
      <atomic-attributes>
        <reference-attributes>
          <reference-attribute qualifier="defaultPaymentAddress" displayName="defaultPaymentAdr
          <reference-attribute qualifier="profilepicture" displayName="profilepicture" type="Me
            <reference-attribute qualifier="groups" displayName="groups" many="true" type="Princ
          </reference-attributes>
        </atomic-attributes>
      </type>
    </types>
</audit-report-config>
```

Now provide a more specific configuration for each of those types, that is for `Address`, `Media`, and `PrincipalGroup`. Do it as you did for the `User` type.

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
```

```

<type code="User" displayName="User">
    <atomic-attributes>
        <atomic-attribute qualifier="name" displayName="name"/>
        <atomic-attribute qualifier="uid" displayName="uid"/>
    <atomic-attributes>
    <reference-attributes>
        <reference-attribute qualifier="defaultPaymentAddress" displayName="defaultPaymentAdc
        <reference-attribute qualifier="profilepicture" displayName="profilepicture" type="Me
        <reference-attribute qualifier="groups" displayName="groups" many="true" type="Princi
    </reference-attributes>
</type>

<type code="Address">
    <atomic-attributes>
        <atomic-attribute qualifier="streetname" displayName="Street"/>
        <atomic-attribute qualifier="town" displayName="City"/>
    </atomic-attributes>
    <reference-attributes>
        <reference-attribute qualifier="title" displayName="title" type="Title"/>
    </reference-attributes>
</type>

<type code="Media">
    <atomic-attributes>
        <atomic-attribute qualifier="code" displayName="Profile picutre"/>
    </atomic-attributes>
</type>

<type code="PrincipalGroup">
    <atomic-attributes>
        <atomic-attribute qualifier="maxbruteforceLoginAttempts" displayName="MaxBruteForceLo
    </atomic-attributes>
</type>
</types>
</audit-report-config>

```

The Address type, just like the User type, has a reference to another item called Title. You can add a configuration for Title, too.

```

<audit-report-config name="sampleConfig">
    <given-root-type>User</given-root-type>
    <types>
        <type code="User" displayName="User">
            <atomic-attributes>
                <atomic-attribute qualifier="name" displayName="name"/>
                <atomic-attribute qualifier="uid" displayName="uid"/>
            <atomic-attributes>
            <reference-attributes>
                <reference-attribute qualifier="defaultPaymentAddress" displayName="defaultPaymentAdc
                <reference-attribute qualifier="profilepicture" displayName="profilepicture" type="Me
                <reference-attribute qualifier="groups" displayName="groups" many="true" type="Princi
            </reference-attributes>
        </type>
    </types>
</audit-report-config>

```

&lt;/type&gt;

```

<type code="Address">
  <atomic-attributes>
    <atomic-attribute qualifier="streetname" displayName="Street"/>
    <atomic-attribute qualifier="town" displayName="City"/>
  </atomic-attributes>
  <reference-attributes>
    <reference-attribute qualifier="title" displayName="title" type="Title"/>
  </reference-attributes>
</type>

<type code="Media">
  <atomic-attributes>
    <atomic-attribute qualifier="code" displayName="Profile picutre"/>
  </atomic-attributes>
</type>

<type code="PrincipalGroup">
  <atomic-attributes>
    <atomic-attribute qualifier="maxbruteforceLoginAttempts" displayName="MaxBruteForceLc
  </atomic-attributes>
</type>

<type code="Title">
  <atomic-attributes>
    <atomic-attribute qualifier="code" displayName="code"/>
    <atomic-attribute qualifier="name" displayName="name"/>
  </atomic-attributes>
</type>
</types>
</audit-report-config>

```

You have successfully prepared your report configuration. You can generate your report, or you can contribute to your configuration using a configuration from some other extension.

## Contributing to a Report Configuration

You can contribute to your existing configuration file from a configuration file of another extension, by adding, overriding, or removing elements of your existing configuration.

The other configuration file allows you to add atomic attributes, or references. You can also add types to your existing configuration, replace, or even remove types you already configured.

Imagine that you saved some `sample-sampleConfig-audit.xml` configuration inside some `sample` extension. Now you create a new, dependent extension called `dependent`. You can use the type definitions of the `dependent` extension to extend the `sample-sampleConfig-audit.xml` configuration.

To contribute to an existing configuration, start from creating a `dependent-sampleConfig-audit.xml` configuration file inside the `dependent` extension.

**dependent-sampleConfig-audit.xml**

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
    </types>
<audit-report-config name="sampleConfig">
```

**→ Remember**

Configurations found in the system across the whole, configured set of extensions are only merged if their name attribute values match.

**Examples**

To add a new atomic attribute to the User type of the current configuration, **append** this attribute to the existing User type. Use the mode attribute of the `<type />` tag with append as the value.

**dependent-sampleConfig-audit.xml**

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
    <type code="User" displayName="User" mode="append">
      <atomic-attributes>
        <atomic-attribute qualifier="description" displayName="description" />
      </atomic-attributes>
    </type>
  </types>
<audit-report-config name="sampleConfig">
```

**i Note**

The default value of the mode attribute is append. It is not necessary to set it up explicitly.

To completely replace a definition of a type, use the **replace** mode:

**dependent-sampleConfig-audit.xml**

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
    <type code="User" displayName="User" mode="replace">
      <atomic-attributes>
        <atomic-attribute qualifier="description" displayName="description" />
      </atomic-attributes>
    </type>
  </types>
<audit-report-config name="sampleConfig">
```

To remove a type, use **remove**:

**dependent-sampleConfig-audit.xml**

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <type code="Title" mode="remove" />
<audit-report-config name="sampleConfig">
```

## i Note

We don't recommend removing a rootType item definition.

# Virtual Attributes

Use virtual attributes to define in a report configuration a dependency to another item even if it doesn't contain the proper reference attribute.

For virtual attributes, use the `<virtual-attributes />` container and the `<virtual-attribute />` tags. Since a `virtual-attribute` doesn't exist in an item definition from the type system point of view, you can only access a referenced type via a FlexibleSearch query. In a best case scenario, you can imagine that a referenced item has a foreign key field which keeps the PK of the current item. As an example, we use the `User` and `Address` types. `Address` contains the `owner` property. `owner` is a type of `Item`, thus it can be an `Address` as well. In the code, you can see how to define such a virtual attribute.

```
<audit-report-config name="sampleConfig">
  <given-root-type>User</given-root-type>
  <types>
    <type code="User" displayName="User">
      <virtual-attributes>
        <virtual-attribute expression="owner" type="Address" many="true" displayName="owned address" />
      </virtual-attributes>
    </type>
  </types>
</audit-report-config>
```

The example shows that there is no `qualifier` attribute because a virtual attribute doesn't exist in the type system. We use an attribute expression to provide a property name in the dependent `Address` type that contains the PK of the `User` type.

For a dependent type that has a reference to a parent type through an attribute of a type other than PK, you can use `VirtualReferenceValuesExtractor`. In our example configuration that uses `VirtualReferenceValuesExtractor`, there is a new custom `CustomForeignKeyTest` type. It has the `key` attribute. The `key` attribute consists of `Order.code + "_" + OrderEntry.entryNumber`. It is really important that this foreign key represents one source (parent) item instance.

```
<type code="OrderEntry" displayName="OrderEntry">
  <atomic-attributes>
    <atomic-attribute qualifier="entryNumber" displayName="entryNumber"/>
    <atomic-attribute qualifier="info" displayName="info"/>
    <atomic-attribute qualifier="quantity" displayName="quantity"/>
  </atomic-attributes>
  <reference-attributes>
    <reference-attribute qualifier="product" displayName="product" type="Product"/>
    <reference-attribute qualifier="unit" displayName="unit" type="Unit"/>
  </reference-attributes>
  <virtual-attributes>
    <virtual-attribute expression="key=beanName(orderCodeAndEntryValuesExtractor)" type="CustomForeignKeyTest" many="true" displayName="customChildren"/>
  </virtual-attributes>
</type>
<type code="CustomForeignKeyTest" displayName="CustomForeignKeyTest">
  <atomic-attributes>
    <atomic-attribute qualifier="code" displayName="code"/>
    <atomic-attribute qualifier="key" displayName="key"/>
  </atomic-attributes>
</type>
```

The `CustomForeignKeyTest` type is defined as a virtual attribute of the `OrderEntry` type. Instead of a simple qualifier name `key`, we now have a `key=beanName(orderCodeAndEntryValuesExtractor)` expression. In the parentheses, we set the name of a Spring bean that implements `VirtualReferenceValuesExtractor` - the `orderCodeAndEntryValuesExtractor` bean.

During the extraction of type instances for a report, you have access to the audit records of types that are higher in the tree structure of your report config (in our case we only have `User`, `Order`, and `OrderEntry` audit records).

Back to the bean definition. The API is defined as:

```
public interface VirtualReferenceValuesExtractor
{
    <AUDITRECORD extends AuditRecordInternal> List<AUDITRECORD> extractValues(AuditRecordInternalProv
}
```

It contains one method that needs implementation - `extractValues`. The two arguments it requires are:

- `AuditRecordInternalProvider` allows you to query for `AuditRecords` of a target type. It has only the `List<AUDITRECORD> queryRecords(Set<Object> values)` method. The method accepts a collection of values queried for a given qualifier of a given target type (in our example the qualifier is `key`, the target type is `CustomForeignKeyTest`, and the values will be a set of Strings equal to `order.code + "_" + orderEntry.entryNumber`)
- `AuditTypeContext` is the context for a target type. It contains properties with target type code, base (parent) type code, base (parent) PKs, and language ISO codes. In our example, the type is `CustomForeignKeyTest`, the `baseType` is `OrderEntry`, and `basePKs` contain a set with all `OrderEntry` PKs that belong to a `User` defined in the report).

It also contains a reference to the internal index of `AuditRecords` that have been already extracted for the report (every type that is higher in hierarchy order than our `CustomForeignKeyTest` in the audit configuration), and methods to get those records from that index, for example `getPayloads(final PK pk)`, `getPayloads(final String type)`, `getPayloads(final String type, final PK pk)`, and `getPayloadsForBasePKs()`.

It is important to note that the implementation operates on generic types. Depending on the report generation phase, first we gather all the records for the report, and in the second phase we use audit events created out of those records to generate report snapshots. In the first phase, we use `AuditRecords`, and in the second - `AuditEvents`. They both extend `AuditRecordInternal`.

Here is a sample bean implementation:

```
public class OrderCodeAndEntryValuesExtractor implements VirtualReferenceValuesExtractor
{
    @Override
    public <AUDITRECORD extends AuditRecordInternal> List<AUDITRECORD> extractValues(
        final AuditRecordInternalProvider<AUDITRECORD> provider, final AuditTypeContext<AUDITRECORD>
    {
        final Set<AUDITRECORD> orderEntries = ctx.getPayloadsForBasePKs();

        final Set<Object> values = new HashSet<>();
        for (final AUDITRECORD entree : orderEntries)
        {
            final String entryNumber = entree.getAttribute("entrynumber").toString();

            final String orderPKStr = entree.getAttribute("order").toString();
            final PK orderPK = PK.parse(orderPKStr);
            final Set<AUDITRECORD> orders = ctx.getPayloads(orderPK);
            for (final AUDITRECORD order : orders)
            {
                values.add(order.getAttribute("code") + "_" + entryNumber);
            }
        }
    }
}
```

```

    }
    return provider.queryRecords(values);
}
}

```

We use the context to get all audit records for every `OrderEntry` that will be in the report - that is the gathering phase. In the report snapshot generation phase, there will only be one `OrderEntry` at a time of a given report tree node.

We iterate in a loop over every `OrderEntry` to get its `entryNumber` (use lowercase attribute names). Depending on the audit type, the `getAttribute(final String key)` method gets a before- or after-operation value of a given attribute. If you need a specific value of the attribute before or after the operation, you can use the `getAttributeBeforeOperation(final String key)` or `getAttributeAfterOperation(final String key)` methods. Take a look into the `AuditRecordInternal` interface for more API, and `AuditRecord` interface for a default behavior of the `getAttribute` method, depending on the audit type.

Every `AuditRecord` payload stores references for parent types (if they have any). In our case, `OrderEntry` stores a parent Order PK in the `order` attribute - serialized to a String. By using context, we get all the `OrderAuditRecords` for that PK.

We iterate over every `Order` audit change and combine **order code**, `_`, and **order entry number**, and store it in the set of values.

To the `provider`, we pass String values that serve as query parameters. We use the `provider` to pass a collection of order code and order entry numbers for which we make the queries. The query returns all `CustomForeignKeyTest` items whose key fields have the values we query for.

There is a special `ReferencesResolver` that uses a predefined `FlexibleSearch` query to look for instances of the `Address` type, and for any records satisfying our configuration. See how to provide a custom `ReferenceResolver` implementation.

```

<audit-report-config name="sampleConfig">
    <given-root-type>User</given-root-type>
    <types>
        <type code="User" displayName="User">
            <virtual-attributes>
                <virtual-attribute type="SomeType" many="true" displayName="some type">
                    <resolves-by expression="some_custom_expression_known_to_underlying_resolver_impl"
                </virtual-attribute>
            </virtual-attributes>
        </type>
    </types>
</audit-report-config>

```

In the example above we have added a new `virtual-attribute` that refers to `SomeType` and uses a custom resolver with a `fooBar` Spring Bean ID. It also uses a custom expression that is understandable for this specific `ReferenceResolver` implementation. Provide your implementation upfront by implementing the `ReferencesResolver` interface.

```

public interface ReferencesResolver
{
    Collection<ResolveResult> resolve(AuditRecord baseRecord, Type typeToResolve, ResolvesBy resolver)

    interface ResolveResult
    {
        PK getItemPk();

        String getTypeCode();
    }
}

```

```

        Collection<AuditRecord> getRecords();

    }

}

```

## Relation Attributes

To collect data with many-to-many relations, use the `relation-attributes` container tag in your XML configuration.

The `relation-attributes` container tag can contain many `relation-attribute` tags. Each tag can define relations that are followed when you generate your report.

A `relation-attribute` tag should contain the names of the relation you want to follow (the name should be the same as in an `....-items.xml` configuration) and the type that is resolved after following the relation. Additionally, you can set a name for `displayName` to display it as a description in your report.

Attribute	Required	Description
<code>relation</code>	Yes	The name of the relation in <code>....-items.xml</code> that should be used when following a relation.
<code>type</code>	Yes	The type that is resolved after following the relation.
<code>displayName</code>	No	The name to be used when creating a report.
<code>displayKey</code>	No	The localized value key to be used when creating a report. The value is used as a <code>displayName</code> for a localized name.

The data passed in the attributes is used by the resolver to gather the appropriate data you require in your report. The objects returned by following the relation are processed further. If there exists a report configuration provided for the type, it is consumed to create a full report.

## Resolving Relations

By default Platform uses the

`de.hybris.platform.audit.provider.internal.resolver.impl.ManyToManyReferencesResolver` class to resolve a relation attribute. An object of this class is available as a Spring bean with a `manyToManyReferencesResolver` name.

To define a custom resolver, extend

`de.hybris.platform.audit.provider.internal.resolver.ReferencesResolver` and register your implementation as a Spring bean. Next, pass the name of the bean in the `resolves-by` tag:

```

<relation-attributes>
    <relation-attribute relation="PrincipalGroupRelation" type="UserGroup" displayName="groups" >
        <resolves-by resolverBeanId="myCustomRelationAttributeResolver"/>
    </relation-attribute>
</relation-attributes>

```

## Example

In the `core-items.xml` file, there is a relation defined that connects `User` (extends `Principal`) and `UserGroup` (extends `PrincipalGroup`). The relation has the `PrincipalGroupRelation` code.

#### core-items.xml

```

<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="items.xsd">
    <!-- (...) -->

    <relations>
        <relation code="PrincipalGroupRelation" autocreate="true" generate="false" localized="false"
            deployment="de.hybris.platform.persistence.link.PrincipalGroupRelation">
            <sourceElement qualifier="members" type="Principal" collectiontype="set" cardinality="many"
                <modifiers read="true" write="true" search="true" optional="true"/>
            </sourceElement>
            <targetElement qualifier="groups" type="PrincipalGroup" collectiontype="set" cardinality="one"
                ordered="false">
                <modifiers read="true" write="true" search="true" optional="true"/>
            </targetElement>
        </relation>
        <!-- (...) -->
    </relations>

    <itemtypes>
        <itemtype code="Principal"
            extends="GenericItem"
            jaloClass="de.hybris.platform.jalo.security.Principal"
            autocreate="true"
            abstract="true"
            generate="true">
            <!-- (...) -->
        </itemtype>
        <itemtype code="PrincipalGroup"
            extends="Principal"
            jaloClass="de.hybris.platform.jalo.security.PrincipalGroup"
            abstract="true"
            autocreate="true"
            generate="true">
            <!-- (...) -->
        </itemtype>
        <itemtype code="User"
            extends="Principal"
            jaloClass="de.hybris.platform.jalo.user.User"
            autocreate="true"
            generate="true">
            <!-- (...) -->
        </itemtype>
        <itemtype code="UserGroup"
            extends="PrincipalGroup"
            jaloClass="de.hybris.platform.jalo.user.UserGroup"
            autocreate="true"
            generate="true">
            <!-- (...) -->
        </itemtype>
        <!-- (...) -->
    </itemtypes>
</items>
```

`PrincipalGroupRelation` is passed as a relation attribute value in a report configuration XML file. Additionally, the `UserGroup` type is set in the `type` attribute. With that, the resolver gathers the data for `UserGroups` that are in a relation with a given user. The resolver then processes the `UserGroup` configuration to gather the specific data you require - in this example, the localized name of the group.

#### user-groups-audit.xml

```

<audit-report-config name="UserReport">
  <given-root-type>User</given-root-type>
  <types>
    <type code="User" displayName="User">
      <atomic-attributes>
        <atomic-attribute qualifier="name" displayName="name"/>
        <atomic-attribute qualifier="uid" displayName="uid"/>
      </atomic-attributes>
      <relation-attributes>
        <relation-attribute relation="principalGroupRelation" displayName="groups" many="true"
          type="UserGroup"/>
      </relation-attributes>
    </type>
    <type code="UserGroup">
      <atomic-attributes>
        <atomic-attribute qualifier="locName" displayName="locName"/>
      </atomic-attributes>
    </type>
  </types>
</audit-report-config>

```

## Generating a Generic Report

After you create and extend a report configuration, you can generate the final report.

### Generating a Report

To generate a report using an existing configuration, find a unique code under which the configuration is stored in the database. Additionally, use `AuditViewService` that provides the following interface:

```

public interface AuditViewService
{
  Stream<ReportView> getViewOn(TypeAuditReportConfig config);
}

```

In your configuration, your root type is `User`. Find a specific instance of the `User` type you want to generate the report for. Use the service with a proper User PK and the name of the configuration (in your case it is `sampleConfig`).

```

// Assuming you have UserService injected by Spring
final UserModel user = userService.getUserForUID("sampleUser");

// Assuming you have AuditViewService injected by Spring
final Stream<ReportView> report = auditViewService.getViewOn(TypeAuditReportConfig.builder().withCon

```

## Interpreting a Report

Each next `ReportView` in a produced Stream contains a specific view on the User and its related items that are described in the configuration sorted by time of changes.

You can call these methods on a `ReportView` object:

- `String getChangingUser()` returns the name of the user who made the change
- `Date getTimestamp()` returns the Date of the change
- `Map<String, Object> getPayload()` returns the whole payload of the changes. This Map could be a Map of Maps depending on a configuration, and you could easily convert it into a proper Json representation if needed.

# Field Name Localization

Platform resolves field name localizations for data report views and allows you to customise the resolving algorithm.

Platform resolves localized field names for the `displayKey` property with values provided either in configuration (`local.properties` / `project.properties`) files or in language bundle files located in the `resources/localization` directory. Platform checks whether the `displayKey` definitions are provided in the configuration files first. If it cannot find them there, it checks whether they are available in the language bundle files.

The resolving algorithm uses the `AuditReportItemNameResolvable` and `AuditReportItemNameResolver` interfaces.

`AuditReportItemNameResolvable` is implemented by all audit report XML elements and handles passing necessary information about element definitions to `AuditReportItemNameResolver`.

## `AuditReportItemNameResolvable`

```
public interface AuditReportItemNameResolvable
{
    String getDisplayKey();
    String getDisplayName();
    String getDefaultName();
}
```

The `AuditReportItemNameResolver` interface has one method - `getName`. It takes as arguments a set of language ISO codes you choose for report generation, and a report XML element definition.

## `AuditReportItemNameResolver`

```
/**
 * Interface which provides methods to resolve name for audit report items for the given configuration
 * Item.
 */
public interface AuditReportItemNameResolver
{
    /**
     * Resolves AuditReportItemNameResolvable item's name
     *
     * @param langIsoCodes languages defined in audit configuration
     * @param item         Item which name will be resolved based on provided details and the report configuration
     * @return String representing name for the item
     * @see AuditReportItemNameResolvable
     */
    String getName(final Set<String> langIsoCodes, final AuditReportItemNameResolvable item);
}
```

## Example

### Configuration

This example report configuration is created for an `auditReportLocalization` extension:

#### `auditReportLocalization-testLocalization-audit.xml`

```
<audit-report-config name="testLocalization">
    <given-root-type>User</given-root-type>
    <types>
        <type code="User" displayKey="type.user.name">
            <atomic-attributes>
                <atomic-attribute qualifier="name" displayName="Name comes first" displayKey="my.local.name">
                    <value>${name}</value>
                </atomic-attribute>
            </atomic-attributes>
        </type>
    </types>
</audit-report-config>
```

```

<atomic-attribute qualifier="uid" displayKey="my.localized.user.uid"/>
</atomic-attributes>
<relation-attributes>
  <relation-attribute relation="PrincipalGroupRelation" type="UserGroup" many="true">
    <resolves-by resolverBeanId="manyToManyReferencesResolver"/>
  </relation-attribute>
</relation-attributes>
</type>
<type displayKey="my.localized.principal-group-relation" code="UserGroup">
  <atomic-attributes>
    <atomic-attribute qualifier="locName" displayKey="type.principalgroup.locname.name"/>
    <atomic-attribute qualifier="description" />
    <atomic-attribute qualifier="name" />
    <atomic-attribute qualifier="uid" />
  </atomic-attributes>
</type>
</types>
</audit-report-config>

```

The key value for the user ID (uid) field is provided in `project.properties` of the `auditReportLocalization` extension as `my.localized.user.uid=Uid` from Config.

These are example key values for the localized fields for the German language provided in `resources/localization` (appropriate localized values for English and Spanish are also provided and you can see some of them in the output below as **DE group name, ES group name**):

```

my.localized.principal-group-relation=DE group name
my.localized.user.name=DE name
my.localized.user.uid=DE uid

```

Platform resolves field names as follows:

- a field name is resolved based on a provided `displayName` attribute; in the example, it is `Name comes first`; in the example, both `displayName` and `displayKey` are defined (for `atomic-attribute qualifier=name` for `User`) but `displayName` takes precedence before `displayKey`
- when a `displayKey` is defined, Platform first checks config properties (`local.properties` / `project.properties`) for appropriate key values; if no values are provided there, Platform searches for them in language bundle files
- if neither of the above is defined, field names are resolved based on their default names:
  - a type name is resolved to the type code; for example, a field name for the `User` type is **User**
  - a virtual attribute name is resolved to the virtual attribute expression value; for example, for a `<virtual-attribute expression="owner">` expression, it is `owner`
  - an atomic attribute name is resolved to the atomic attribute qualifier value; for example, for `<atomic-attribute qualifier="description" />`, it is `description`
  - a reference attribute name is also resolved to the qualifier value
  - for a relation attribute, the default name is resolved based on the definition of the referenced type; for example, if a relation attribute is referenced by `type="Address"`, then the default name is resolved by the definition of the `Address` type, with the same order for resolving names applying for the type (that is: `displayName`, or `displayKey` (with values provided in the config properties)), then, if neither is defined, the name is resolved as the type code, which is **Address**

## Generic Audit

Generic audit tracks every persistence action, including creation, modification, and deletion for specified types. The audit is stored as a change log that allows you to see how an item changed over time.

Auditing is enabled globally by default through the `auditing.enabled=true` property. To disable it, add `auditing.enabled=false` to your `local.properties` file.

With auditing enabled globally, you can enable it separately for each type you wish to track. The property that allows you to do it uses the `audit.<typecode>.enabled=true` naming convention. For example, to enable auditing for the `User` type, use `audit.user.enabled=true`. There are multiple item types enabled for auditing by default. Navigate to the `<HYBRIS_BIN_DIR>/platform/bin/platform/project.properties` directory to see the list of such items.

Generic audit collects data that you can provide to your customers through personal data reports, for example for the purposes of GDPR compliance. For more information, see [Generating Personal Data Reports in Backoffice Framework](#).

### i Note

You cannot exclude subtypes of a specified type from an audit. If you enable auditing for `User`, all of the `User` subtypes are audited as well. The following is invalid:

```
audit.user.enabled=true
audit.employee.enabled=false
```

The reason is that `Employee` extends `User` and it is impossible to audit `Users` and at the same time skip `Employees`.

You can specify what you want to audit on a **finer** level of granularity than `type` - for more information, see [Change Log Filtering](#).

With many-to-many relations, enabling both sides of the relation isn't enough for the auditing feature to work automatically. Configure those relations separately in the same manner as you do for the source and target types.

## Audit Log Storage

SAP Commerce persists audit logs for types configured for auditing automatically. It uses a default storage implementation for that purpose.

### i Note

Auditing consumes database space. SAP Commerce audits multiple item types by default. If you make changes in those items, it may affect database performance over time.

SAP Commerce provides a JDBC-based implementation of audit storage by default. For each type, a new corresponding table is created during initialization or update. The table name follows the `<deploymentName><numericTypeCode>sn` naming convention. For example, an audit table for the `User` type would be called `users4sn`.

A table for a standard type has the following structure:

Field Name	Type	Description
ID	bigint	Record id
ITEMPK	bigint	Audited instance PK
ITEMTYPEPK	bigint	Audited type PK (ComposedType)
timestamp	datetime	Change timestamp
currenttimestamp	datetime	Audit record timestamp
changinguser	varchar	Author of the change
context	longtext	Additional context (JSON)

Field Name	Type	Description
payloadbefore	longtext	The whole business payload of audited type before a persistence operation (JSON)  Before INSERT, the payloadbefore is empty.
payloadafter	longtext	The whole business payload of audited type after a persistence operation (JSON).  After DELETE, the payloadafter is empty.
operationtype	bigint	0 - deletion, 1 - creation, 2 - modification

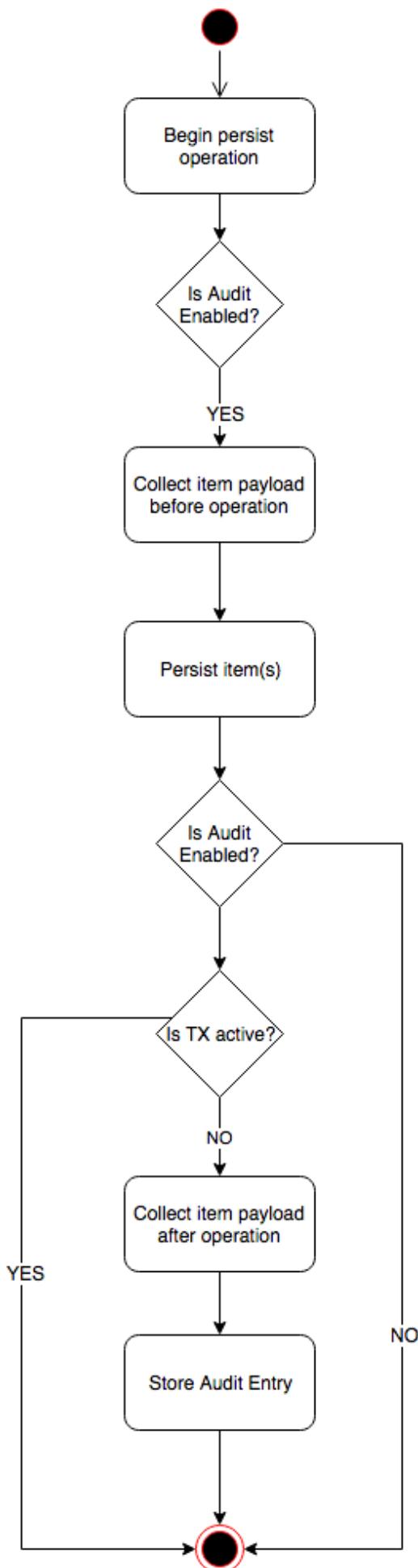
A table for many-to-many-relation types has the same structure, plus three additional columns:

Field Name	Type	Description
sourcePK	bigint	Source item PK
targetPK	bigint	Target item PK
languagePK	bigint	Language PK

### i Note

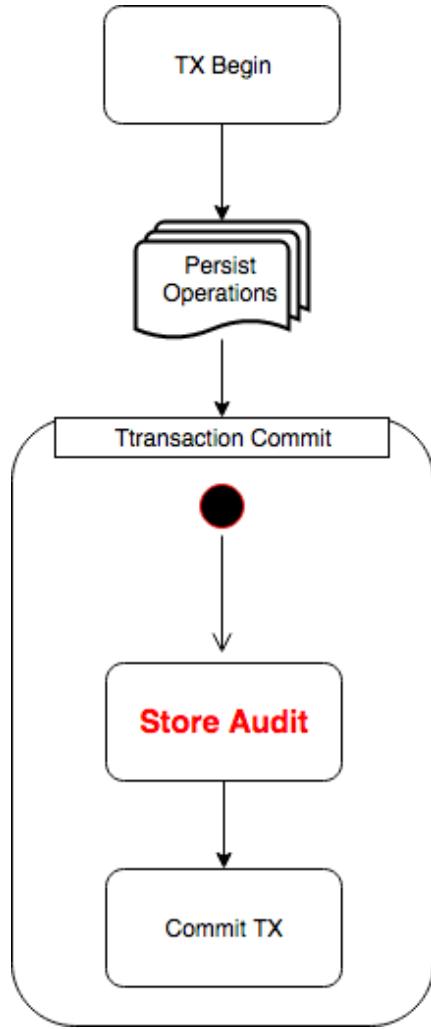
When auditing is enabled globally, audit tables are created up front, no matter whether a given type is configured for auditing or not. If it isn't configured, the table remains empty.

When you configure a type for auditing, each persistence operation to the type itself triggers the storing of an audit log. The framework does it automatically on each `ModelService#save` and `ModelService#remove` call. It doesn't require any further action or configuration. The following diagram shows the process for storing changes to a type:



An audit log entry is stored immediately after persisting item data itself when there's no active transaction.

In the following diagram, you can see how the same operation works when a transaction is in place:



The transaction is started and within it all operations such as creation, modification, and deletion are performed as in the first diagram. Note that an audit storing action is delayed to the point when the transaction is committed.

## Searching for Audit Log Entries

SAP Commerce provides a dedicated service that allows you to search for audit log entries for a particular type using specific search rules (even in a JSON audit payload).

An audit log is not backed by the SAP Commerce type system. As a result, it isn't possible to use `ModelService` or `FlexibleSearch` to obtain the entries.

The service is configured in Spring with the ID `readAuditGateway`, and has the following interface:

```

public interface ReadAuditGateway
{
    <T extends AuditRecord> Stream<T> search(AuditSearchQuery searchQuery);
}

```

The interface returns a Stream of `AuditRecord` objects that is an object representation of an audit entry. An `AuditSearchQuery` object that is passed to the search method is the object that holds all the necessary information to perform the search query on an audit table. It allows you to search in standard type-related audit tables as well as in link-related audit tables.

The default implementation of audit is backed by JDBC. It is possible to implement a different persistence method since `AuditSearchQuery` only keeps information about the type to be searched together with a collection of `SearchRule` objects that define which fields in the audit table or which attributes (values) in the audit table must be searched.

## Search Examples

The simplest query is a query for a type and a particular PK. To create such a query, first prepare an `AuditSearchQuery` object using some existing PK. In this example, we use `User` as a type:

### i Note

For convenience, `AuditSearchQuery` provides builders for building queries for standard types as well as for links.

```
// assuming User was found using UserService
final UserModel user = userService.getUserForUID("someUser");
// build the audit search query
final AuditSearchQuery query = AuditSearchQuery.forType("User").withPkSearchRules(user.getPk()).build();
// do search
final Stream<AuditRecord> records = readAuditGateway.search(query);
```

You can also search for more than one PK in one go:

```
// assuming User was found using UserService
final UserModel user1 = userService.getUserForUID("someUser-1");
final UserModel user2 = userService.getUserForUID("someUser-2");
final UserModel user3 = userService.getUserForUID("someUser-3");

// build the audit search query
final AuditSearchQuery query = AuditSearchQuery.forType("User").withPkSearchRules(user1.getPk(), user2.getPk(), user3.getPk());
// do search
final Stream<AuditRecord> records = readAuditGateway.search(query);
```

Although it is possible to create a query that reads the entire audit table into memory, it is not recommended as it may cause out-of-memory errors. Such a dangerous query can look like this:

```
final Stream<AuditRecord> records = readAuditGateway.search(AuditSearchQuery.forType("User").build())
```

It is also possible to create a query that does a search within a JSON payload of an audit entry:

```
// assuming User was found using UserService
final UserModel user = userService.getUserForUID("someUser");
// build the audit search query
final AuditSearchQuery query = AuditSearchQuery.forType("User")
    .withPkSearchRules(user.getPk())
    .withPayloadSearchRule("someFieldInPayload", "value")
    .withPayloadSearchRule("anotherFieldInPayload", "theOtherValue")
    .build();
// do search
final Stream<AuditRecord> records = readAuditGateway.search(query);
```

## Removing Audit Log Entries

Generic audit allows you to remove existing audit log entries.

This service is configured in Spring with the ID `writeAuditGateway`, and has the following interface:

```
public interface WriteAuditGateway
{
    void saveLinkAuditRecords(List<LinkAuditRecordCommand> cmdList);
    void saveTypeAuditRecords(List<TypeAuditRecordCommand> cmdList);
    int removeAuditRecordsForType(String type);
```

```
int removeAuditRecordsForType(String type, PK pk);
}
```

To remove audit log entries for a particular type and PK, use this code:

```
// assuming User was found using UserService
final UserModel user = userService.getUserForUID("someUser");

writeAuditGateway.removeAuditRecordsForType("User", user.getPk());
```

If a particular item is marked as locked against further modifications, it is not possible to remove its audit log entries.

For more information about locking and unlocking items, see [Item Locking Service](#).

## Audit Log and Generic Data Report

You can collect audit logs generated by generic audit and present such data in a form of report. It is possible thanks to the generic data report feature. You can do it in Backoffice or programmatically.

You can include audit log data for each type in the final generic data report by checking **Audit** in the **Create New Audit Report Data** wizard in Backoffice. For more information about the **Create New Audit Report Data** wizard, see [Generating Personal Data Reports in Backoffice Framework](#).

To do the same programmatically:

```
// assuming User was found using UserService
final UserModel user = userService.getUserForUID("someUser");

// option withFullReport() enables audit data in the report
final TypeAuditReportConfig config = TypeAuditReportConfig.builder()
    .withConfigName("PersonalDataReport")
    .withRootTypePk(user.getPk())
    .withFullReport()
    .build();

final Stream<AuditView> report = auditViewService.getViewOn(config);
```

For more information, see [Generic Data Report and Audit](#).

## Change Log Filtering

Generic audit allows you to configure auditing of data on a finer level of granularity than type. Narrowing down audited data to the data that you actually need prevents unnecessary database size growth.

When you use generic audit on **per-type** basis, you may decide to audit **Carts** and not to audit **Users**. As a result, all changes done to every cart are recorded and stored in the database. It may be a lot of data. Filtering allows you to define fine-grained rules, where the decision whether to store an audit change is based on the change content itself. For example, in the default implementation of the change log filtering, instead of auditing **all** carts, only saved instances of carts are audited. For more information about it, see [Conditional Session Cart Audit](#).

You can filter audit changes using these approaches:

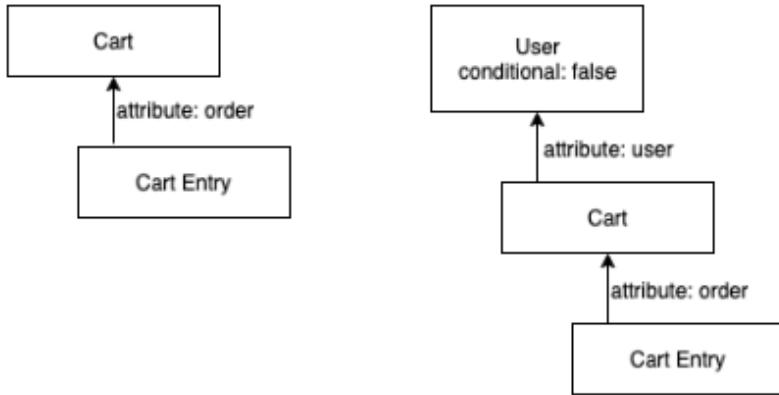
- with the **conditional audit**, you use an XML file to define **generic** filtering configurations that are filtering rules that specify what changes to audit
- with the **AuditChangeFilter** interface, you create **customized** filtering configurations programmatically

# Conditional Audit

With the conditional audit, you can configure what to filter without programming.

The conditional audit uses a filtering configuration, which is a filtering rule, consisting of conditional audit groups. An **audit group** consists of a hierarchy of logically related types and an auditing condition. The hierarchy forms a tree structure. Each node in the type hierarchy has an attribute that defines how to traverse from the node to the parent. An auditing condition decides whether a change in data is audited or not. A condition is always evaluated on a root type.

See the examples of Cart, and User conditional audit configuration structures:



In the Cart configuration example, **Cart** is the root type, and **CartEntry** is its child. This configuration applies to audit changes of both **Cart** and **CartEntry**. When a cart changes, that change is inspected against the configuration. A changed cart is of type **Cart** - it matches the type of the root in the configuration tree. Therefore, a condition set in that audit group can be evaluated directly on it. To inspect whether changes in **Cart Entry** can be audited, the audit group first needs to obtain the cart data. It uses the configured **order** attribute to find the appropriate cart, and evaluates the condition on it.

The User example extends the Cart example. **User** is the root type. It has the **conditional=false** attribute, which means that the audit group doesn't affect the audit changes made to a **User** instance itself. The **User** node is only added to the configuration because it an element of the condition used in the audit's group - specifically, the conditional expression is evaluated on it.

See this business scenario that uses the User configuration:

Assume that you want to audit some data of two specific customers (the **Customer** type). The customer ids are 001, and 002 respectively. You want to see how their carts (**Cart**) change, that is, what products (**CartEntries**) they add to their carts, in what quantity, and what products they remove from their carts before they eventually place their orders. You create an audit group that reflects the relation between those types - the group consists of the **Customer**, **Cart**, and **CartEntry** types, it also has a condition. The condition states that Platform should only persist data changes made to carts and cart entries that belong to customers whose ids are 001, and 002. Since you don't want to audit changes made to the customers themselves, that is changes to the attributes of the **Customer** type, the **Customer** type isn't affected by the configuration thanks to the **conditional=false** attribute.

This is the filtering configuration for this scenario:

```

<conditional-audit-config>
    <audit-group name="audit-carts-for-specific-users">
        <condition>get('uid') == '001' or get('uid') == '002'</condition>
        <type code="Customer" conditional="false">
            <type code="Cart" attribute="user">
                <type code="CartEntry" attribute="order"/>
            </type>
        </type>
    </audit-group>
</conditional-audit-config>

```

## Caution

The type traversal only supports one-to-many relations. Types can't be repeated in a configuration.

A condition is a SPEL expression. You can't rely on models for it. Instead, use the following method to access attributes:

```
public Object get(final String attribute)
```

An audit change is audited when the change states before or after a modification matches an audit group.

The following property enables full support for SpEL in audit condition evaluation. If `false`, only basic features such as calling methods and properties are supported. If `true`, full SpEL with static methods invocation and new object creation is supported:

```
audit.conditional.fullSpELSupport=false
```

The default value is `false`. If full support is enabled, make sure that your conditions are not vulnerable to code injection.

## XML Configuration

A conditional audit configuration starts with a single `conditional-audit-config` tag. Its children are `audit-group` elements that define conditional audit groups.

Element	Attributes	Description
<code>audit-group</code>	<code>name</code>	Conditional audit group definition. Identified by the <code>name</code> parameter.
<code>condition</code>		Defines a boolean expression that is executed on the root of a defined type hierarchy. If condition is <code>false</code> , a matching audit change isn't audited.
<code>type</code>		Type that forms an audit group tree.
	<code>code</code>	A typecode representing an item type, for example, the <code>Cart</code> type.
	<code>attribute</code>	Determines whether an attribute is used to traverse from a type to its parent, for example, <code>order</code> for <code>Cart Entry</code> .
	<code>conditional</code>	Determines whether a type should be affected by an audit group. If <code>false</code> , the audit group isn't triggered for the given type.
	<code>subtypes</code>	Determines whether subtypes of a type should be affected by the audit group. If <code>true</code> , the rule is also triggered for the subtypes.

Configuration file example:

```
<?xml version="1.0" encoding="UTF-8"?>
<conditional-audit-config>
  <audit-group name="session-cart">
    <condition>get('saveTime') != null</condition>
    <type code="Cart">
      <type code="CartEntry" attribute="order"/>
    </type>
  </audit-group>
</conditional-audit-config>
```

```
</audit-group>
</conditional-audit-config>
```

There can be more than one `audit-group` elements in a configuration. In such case, they're evaluated one by one.

You can use the `subtypes` attribute on a type tag to specify whether subtypes should be matched by this audit group.

```
<conditional-audit-config>
  <audit-group name="title-1">
    <condition>!get('code').endsWith('audit1')</condition>
    <type code="Title"/>
  </audit-group>

  <audit-group name="title-2">
    <condition>!get('code').endsWith('audit2')</condition>
    <type code="Title" subtypes="true"/>
  </audit-group>
</conditional-audit-config>
```

A Type can have the `conditional` attribute. If it's `false`, audit changes of that Type aren't affected by this audit's group.

```
<?xml version="1.0" encoding="UTF-8"?>

<conditional-audit-config>
  <audit-group name="order-entry-user">
    <condition>get('code').endsWith('_audit')</condition>
    <type code="User" conditional="false">
      <type code="Order" attribute="user">
        <type code="OrderEntry" attribute="order"/>
      </type>
    </type>
  </audit-group>
</conditional-audit-config>
```

All audit groups are enabled by default. You can disable them on runtime using a parameter:

```
audit.conditional.<audit-group name>.enabled=false
```

## Defining Conditional Audit Groups

Learn to define conditional audit groups for the generic audit.

### Context

The example group audits all `Title` items except those that have codes ending with the `no-audit` suffix.

The group is enabled by default. You can disable it on runtime by setting the property:

```
audit.conditional.title-with-suffix.enabled=false
```

### Procedure

1. Create a configuration file and put it into a resource folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<conditional-audit-config>
  <audit-group name="title-with-suffix">
    <condition>!get('code').endsWith('no-audit')</condition>
    <type code="Title"/>
  </audit-group>
</conditional-audit-config>
```

```
</audit-group>
</conditional-audit-config>
```

## 2. Register a

de.hybris.platform.persistence.audit.internal.conditional.ConditionalAuditChangeFilter Spring bean that references the configuration. Add the configuration to the auditChangeFilters list:

```
<bean name="titleWithSuffixConditionalAudit" class="de.hybris.platform.persistence.audit.intern
<constructor-arg name="conditionalAuditConfigXml" value="title-with-suffix-audit.xml"/>
<constructor-arg name="sldDataContainerProvider" ref="sldDataContainerProvider"/>
<constructor-arg name="commonI18NService" ref="commonI18NService"/>
<constructor-arg name="typeService" ref="typeService"/>
</bean>

<bean depends-on="auditChangeFilters" parent="listMergeDirective">
    <property name="add" ref="titleWithSuffixConditionalAudit" />
</bean>
```

## Conditional Session Cart Audit

SAP Commerce comes with a preconfigured generic audit group that allows you to ignore session carts and audit stored carts only.

A session cart is a transient piece of data - it may change a lot before it is eventually persisted as an order. Due to those multiple changes, auditing session carts has a big impact on performance and database storage. In most cases, you may not need to store all those data changes.

All changes made to a cart or a cart entry are ignored unless the cart is saved. A cart is identified as stored if its saveTime attribute is not null. This attribute is set when a storefront user chooses to save their cart, and is set to null when the user restores a previously saved cart. Cart entries are only audited if they belong to a stored cart.

This group is based on conditional audit feature and defined by conditionalSessionCartAudit bean. Conditional audit configuration file is stored in audit/conditional-session-cart-audit.xml in the Platform core extension.

To enable a session cart audit, remove conditionalSessionCartAudit from the auditChangeFilters bean, or change the audit.conditional.session-cart.enabled property value to false:

```
audit.conditional.session-cart.enabled=false
```

## AuditChangeFilter Interface Filtering

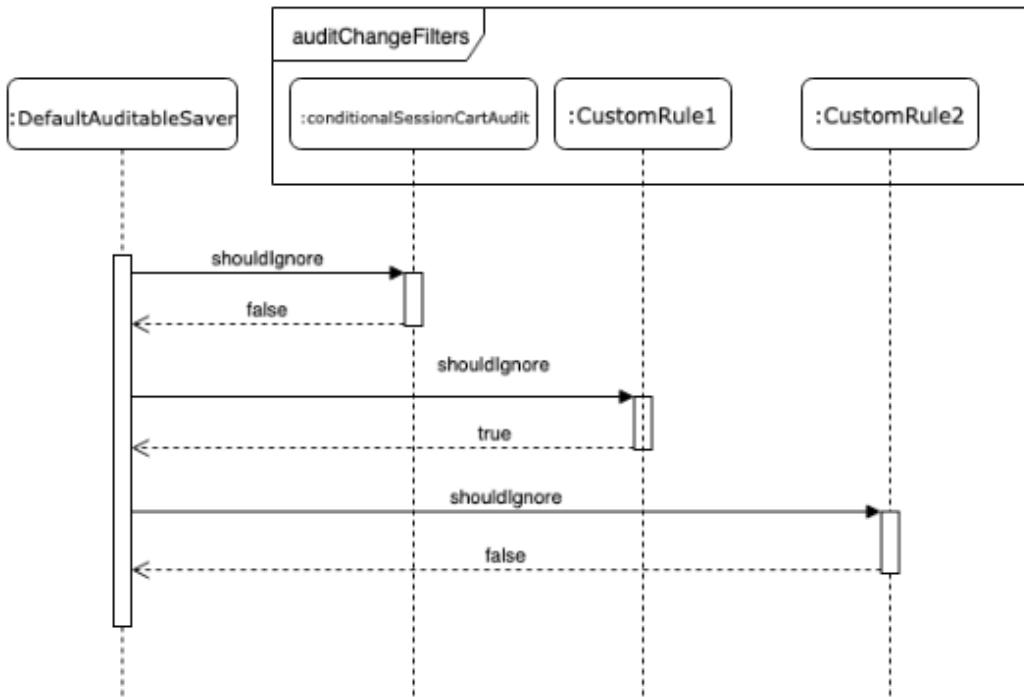
The interface AuditChangeFilter allows you to implement audit filtering on an api level.

The de.hybris.platform.persistence.audit.AuditChangeFilter interface provides the ignoreAudit method:



The ignoreAudit method returns true if AuditableChange shouldn't be stored in the database. The auditChangeFilters Spring bean (defined in servicelayer-spring.xml) is a list of audit change filters that are consulted every time a change is to be saved. AuditableChange isn't saved if any of the interface implementations in the list return true.

The diagram illustrates a scenario with three audit change filters. CustomRule1 filter returns true, so the change isn't audited.



Method parameter `AuditableChange` is a data container containing object state before and after audited operation. Use these values to determine if a given change should be saved. Since auditing is performed in low-level persistence layer, accessing models and using other high-level features, such as dynamic attributes, is not supported.

## Implementing the AuditChangeFilter Interface

Learn how to implement the `AuditChangeFilter` interface to create custom audit filtering rules.

### Context

The example assumes that you want to audit only those `Title` items that were created this year.

### Procedure

1. Create a `AuditTitlesCreatedThisYear` class that implements the `de.hybris.platform.persistence.audit.AuditChangeFilter` interface:

```

package de.hybris.platform.persistence.audit;

import de.hybris.platform.core.model.user.TitleModel;
import de.hybris.platform.directpersistence.cache.SLDDataContainer;

import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.Date;

public class AuditTitlesCreatedThisYear implements AuditChangeFilter
{
    @Override
    public boolean ignoreAudit(final AuditableChange change)
    {
        final SLDDataContainer dataContainer = getDataContainer(change);

        if (dataContainer == null || !TitleModel._TYPECODE.equals(dataContainer.getTypeCode()))
        {
            return false;
        }

        final LocalDateTime creationTime = getCreationTime(dataContainer);
    }
}
  
```

```

        return creationTime.getYear() < LocalDateTime.now().getYear();
    }

    private LocalDateTime getCreationTime(final SLDDataContainer dataContainer)
    {
        final Date creationDate = (Date) dataContainer.getAttributeValue("creationtime", null).ge
        return LocalDateTime.ofInstant(creationDate.toInstant(), ZoneId.systemDefault());
    }

    private SLDDataContainer getDataContainer(final AuditableChange change)
    {
        return change.getAfter() != null ? change.getAfter() : change.getBefore();
    }
}

```

The class checks whether `AuditableChange` is related to a Title item. It then uses `SLDDataContainer` to get the `creationtime` attribute. If the Title `creationtime` year is lesser than current year, true is returned. It means that audit should not be executed.

2. Register the class as a Spring bean and add it to `auditChangeFilters`:

```

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean class="de.hybris.platform.persistence.audit.AuditTitlesCreatedThisYear" name="auditTi
    <bean depends-on="auditChangeFilters" parent="listMergeDirective">
        <property name="add" ref="auditTitlesCreatedInThisYear" />
    </bean>
</beans>

```

`listMergeDirective` is used to add the `auditTitlesCreatedInThisYear` bean to `auditChangeFilters`.

## Generic Audit Properties

See the list of available generic audit properties with their default values.

Property	Default Value
<code>auditing.enabled</code>	<code>true</code>
<code>auditing.alltypes.enabled</code>	<code>true</code>
<code>auditing.blacklist</code>	<code>itemsynctimestamp,joblog,logfile,jobmedia,task,t</code>
<code>audit.&lt;typeCode&gt;.enabled</code>	<code>true</code> for <code>user, principalgrouprelation, abstractcontac</code> <code>paymentinfo, paymentmode, product, quote, quoteentry, r</code> <code>userpasswordchangeaudit, cxusertosegment, cxsegment</code> <code>csticket, comment, employee, csagentgroup, basesite, sa</code>
<code>audit.&lt;typeCode&gt;.blacklistedProperties</code>	Not applicable

Property	Default Value
audit.user.blacklistedProperties	encodedpassword
audit.userpasswordchangeaudit.blacklistedProperties	encodedpassword
audit.blacklistedProperties.obfuscationValue	****
audit.conditional.session-cart.enabled	true
audit.conditional.fullSpELSupport	false
audit.write.jdbc.batch.size	50

## Data Retention Framework

The Data Retention Framework enables you to retain instances of specified types until they are cleaned up. The framework uses configurable rules in which you specify the instances you are interested in, the cleanup logic you want to execute on them, and when to execute it.

The framework operation flow starts with collecting items (instances of types) according to defined rules. Platform retains those items for a defined period of time until they are removed or some other cleanup action is performed on them. The framework enables you to decide when to perform actions on a given item. You can decide whether it is a specific point in time or a finished period of time. When the point in time arrives, or a retention period for that item has finished, the framework mechanism triggers on that particular item specified actions.

### i Note

#### Retention of Audited Data

A retention job using the `basicRemoveCleanupAction` action only removes audited data if the **generic audit** feature is enabled.

For information about the generic audit feature, see [Generic Audit](#).

The framework provides a basic strategy for simple removal of items. For more sophisticated cases, the mechanism can use implementations you can provide in your business extensions.

The Data Retention Framework is based on a few concepts:

#### Retention Rules

You configure **retention rules** to indicate the items you are interested in. You specify a retention period, and state what cleanup logic you want to execute on the items when the retention period is over. For example, you may decide you want to remove orders that you have kept in your online store for at least 10 years.

#### Data Providers

A **data provider** collects and returns the items specified in your retention rules. Data providers return batch items in an iteration. A data provider is an instance of a class that implements the `RetentionItemsProvider` interface.

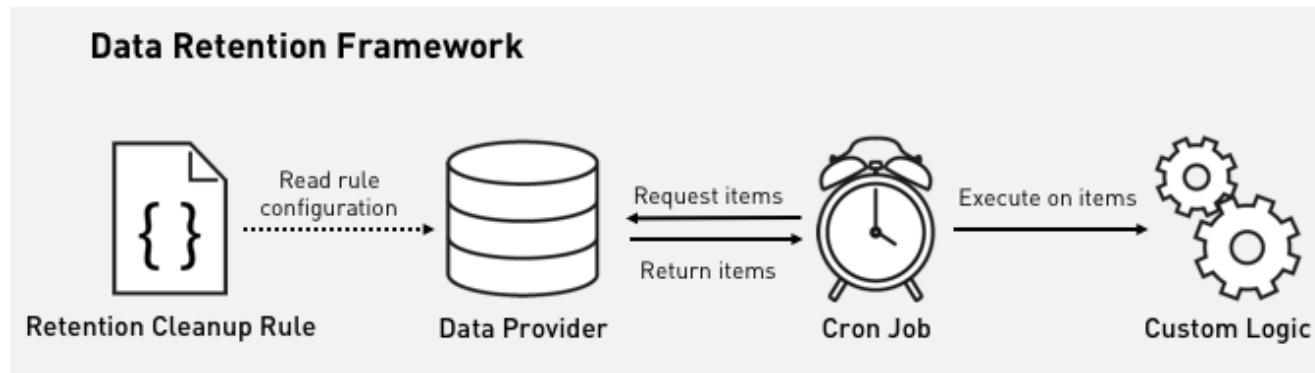
## Cron Job

A cron job (`AfterRetentionCleanupJobPerformable`) executes the retention logic. First, data providers fetch the data. Then the logic is executed on the data. You can execute the cron job with a Java API (`CronjobService.performCronJob(CronJobModel)`) or with a trigger. The execution is delegated to `AfterRetentionCleanupJobPerformable`.

### Business Logic

`RetentionCleanupAction` is an interface you should implement to define the custom logic you would like to perform on the collected items.

The diagram sums up the entire idea:



## Retention Rules

The Data Retention Framework allows you to configure retention rules.

To start working with the framework, create a `RetentionJobModel` model to use with either the `AfterRetentionCleanupRuleModel` or `FlexibleSearchRetentionRuleModel` rule configurations. To configure a rule, set your values for appropriate rule configuration parameters.

## AfterRetentionCleanupRuleModel Configuration

This example configuration enables you to get orders whose `itemtype` code starts with an `order` prefix and whose `expirationTime` is older than 10 minutes. These orders are then removed since the `actionReference` parameter is set to `basicRemoveCleanupAction`.

```
code="Rule1"
retirementItemType="Order"
retirementDateAttribute="Order:expirationTime"
retentionTimeSeconds="600"
itemFilterExpression="{code} like 'order%'"
actionReference="basicRemoveCleanupAction"
```

The `AfterRetentionCleanupRuleModel` parameters include:

Parameter Name	Parameter Type	Required
code	String	Yes
retirementItemType	ComposedType	Yes
retirementDateAttribute	AttributeDescriptorModel	Yes/No*
retentionTimeSeconds	long	Yes/No*

Parameter Name	Parameter Type	Required
itemFilterExpression	String	No
retirementDateExpression	String	Yes/No*
actionReference	String	Yes

\* It is required to set `retirementDateAttribute` with `retentionTimeSeconds` or set `retirementDateExpression` in `AfterRetentionCleanupRuleModel`.

## FlexibleSearchRetentionRuleModel Configuration

This example configuration enables you to return items defined through the `searchQuery` parameter. These items are then removed because the `actionReference` parameter is set to `basicRemoveCleanupAction`.

```
code="Rule2"
actionReference="basicRemoveCleanupAction"
searchQuery="select {PK},{itemType} from {Order} where {code} like 'fsorder%' and {expirationTime} <
```

### ⚠ Caution

Ensure that the set `searchQuery` only returns a pk and an item type in proper order, for example "SELECT {PK}, {itemType} FROM..."

The `FlexibleSearchRetentionRuleModel` parameters include:

Parameter Name	Parameter Type	Required
code	String	Yes
actionReference	String	Yes
searchQuery	String	Yes
queryParameters	Map<String,String>	No
retentionTimeSeconds	long	No

### Calculated Query Parameters

Due to the fact that date calculations are database dependent, we added some calculated parameters to the `queryParameters` map. You can use those parameters for `searchQuery`. This example configuration enables you to get orders whose `expirationTime` is older than 10 minutes.

```
code="Rule3"
actionReference="basicRemoveCleanupAction"
retentionTimeSeconds="600"
searchQuery="select {PK},{itemtype} from {Order} where {expirationTime} < ?CALC_RETIREMENT_TIME"
```

The calculated parameters include:

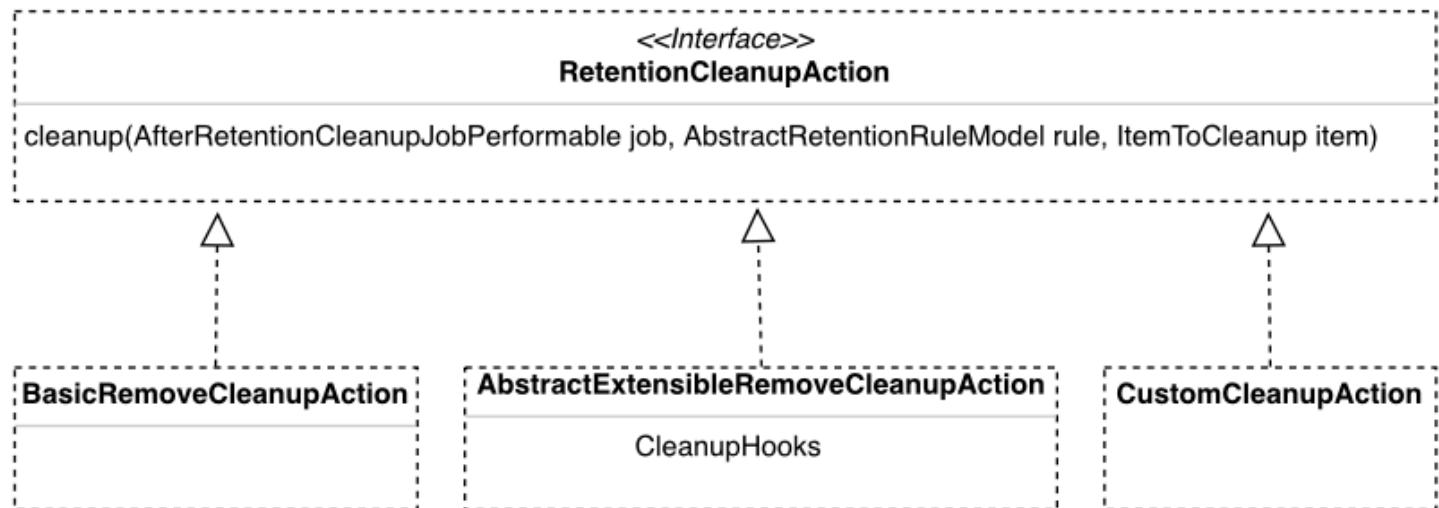
Key Name	Value Description	Type
JAVA_CURRENT_TIME	current_time from server	java.util.Date

Key Name	Value Description	Type
CALC_RETIREMENT_TIME	current_time, minus retentionTimeSeconds	java.util.Date
RETENTION_TIME_SECONDS	retentionTimeSeconds	long

## Implementing Cleanup Logic

The Data Retention Framework allows you to enhance the basic data removal action. You can also implement your own specialized logic for cleaning up data.

The Data Retention Framework offers the basic and extensible cleanup actions. Both of them implement the `RetentionCleanupAction` interface. You have to implement this interface to create your custom cleanup actions.



### Basic Removal Action

With the basic removal action, you remove items provided by a data provider (`BasicRemoveCleanupAction` has a Spring bean `id = "basicRemoveCleanupAction"`). The action also removes item-related audit records generated by the Generic Audit.

For more information on audit records, see [Generic Audit](#).

### Extensible Removal Action

The extensible removal action uses the abstract `AbstractExtensibleRemoveCleanupAction` class. Here you can enhance the basic removal process by plugging in custom hook logic. The extensible removal action executes the hook logic on related items and then removes the main item.

Anonymizing customer comments may be an example use case for the extensible removal action. In such a scenario your hook logic may remove identification information from comments, such as customer name. Then the main customer item is removed completely and what you end up with is their anonymized comments only.

Custom hooks must implement the `ItemCleanupHook` interface. They should also implement the `cleanupRelatedObjects` method that performs your custom logic on objects related to your main item.

During the cleaning process the `AbstractExtensibleRemoveCleanupAction` class iterates through all hooks and invokes the implemented `cleanupRelatedObjects` methods. The default implementation of

`DefaultExtensibleRemoveCleanupAction` removes the main item after all hooks are invoked. You can define a list of hooks in a Spring configuration by setting the `itemCleanupHooks` property.

Here is an example `DefaultExtensibleRemoveCleanupAction` configuration:

```
<alias name="defaultExtensibleRemoveCleanupAction" alias="extensibleRemoveCleanupAction"/>
<bean id="defaultExtensibleRemoveCleanupAction" class="de.hybris.platform.retention.impl.DefaultExtensibleRemoveCleanupAction">
    <property name="itemCleanupHooks">
        <list/>
    </property>
</bean>
```

In addition, there are two predefined lists of hooks for orders and customers:

```
<util:list id="orderCleanupHooks" value-type="de.hybris.platform.retention.hook.ItemCleanupHook"/>
<util:list id="customerCleanupHooks" value-type="de.hybris.platform.retention.hook.ItemCleanupHook"/>
```

You can extend those lists and add your hooks in a similar way as in this example:

```
<bean id="customCleanupHookMergeDirective" depends-on="customerCleanupHooks" parent="listMergeDirective">
    <property name="add" ref="customCleanupHook" />
</bean>
```

And use it in your extensible action:

```
<bean id="customRemoveCleanupAction" class="foo.bar.CustomExtensibleRemoveCleanupAction" parent="abstractExtensibleRemoveCleanupAction">
    <property name="itemCleanupHooks" ref="customerCleanupHooks"/>
</bean>
```

## Custom Cleanup Action

If you want to introduce your custom cleanup logic, provide your own implementation of the `RetentionCleanupAction` interface, set up a bean id on a Spring XML, and set up the name of this bean on `actionReference`.

## Defining Cronjob Behavior

One of the required parameters of `RetentionJobModel` is `batchSize`. It determines how many items are returned by a data provider in one package and processed by `RetentionCleanupAction`.

The whole package of items is processed in a separate transaction scope. If any exception is thrown from `RetentionCleanupAction`, the transaction is rolled back for the entire package.

## Example Rule Configurations

See the examples of rule configurations in data retention framework. The examples include `AfterRetentionCleanupRule` and `FlexibleSearchRetentionRule` configurations.

### AfterRetentionCleanupRule

#### i Note

Set `impex.legacy.scripting` to `false`. In SAP Commerce Administration Console, check the `Enable code execution` box in the `Settings` section of the `Impex Import` tab.

### Example of AfterRetentionCleanupRule

```

# CUSTOMER
INSERT_UPDATE Customer;uid[unique=true];customerID;name;
;user1;user1;name;

# RETENTION RULE
insert_update AfterRetentionCleanupRule;code[unique=true];retirementItemType(code);itemFilterExpressi
    ; rule; Order; {code} like 'order%'; basicRemoveCleanupAction; 600; Order:expirationTime

# JOB
INSERT_UPDATE RetentionJob;code[unique=true];retentionRule(code);batchSize
; retentionJob; rule; 100

# CRON JOB
INSERT_UPDATE CronJob;code[unique=true];job(code);sessionLanguage(isoCode)[default=en]
; retentionCronJob; retentionJob;

# ORDERS
INSERT_UPDATE Order;code[unique=true];user(uid);date[dateformat=dd.MM.yyyy HH:mm];expirationTime[date
"#%groovy%
import java.util.Date
import groovy.time.TimeCategory

String lines = ''
Date now = new java.util.Date()
String format = "dd.MM.yyyy HH:mm:ss"
String creationDate = now.format(format, TimeZone.getTimeZone('CET'))

for(i = 0; i < 50; i++) {
    use( TimeCategory ) {
        after30Mins = now - (i * 30).seconds
    }
    String expirationTime = after30Mins.format(format, TimeZone.getTimeZone('CET'))
    lines += 'order' + i + ';' + user1 + ';' + creationDate + ';' + expirationTime + ';EUR\n'
}
impex.info('importing Orders: +' + lines)
reader = new CSVReader(lines)
reader.setMaxBufferLines(100000)
impex.includeExternalData(reader)
';

# select {code}, {expirationTime} from {Order} where {code} like 'order%' order by {expirationTime}

```

### Execute the cron job using the scripting console (groovy)

```

import de.hybris.platform.core.Registry
import de.hybris.platform.servicelayer.cronjob.CronJobService
CronJobService cronJobService = Registry.getApplicationContext().getBean("cronJobService")
cronJobService.performCronJob(cronJobService.getCronJob("retentionCronJob"))

```

## FlexibleSearchRetentionRule

### i Note

The query is DB-dependent (date-time manipulation). In this example, it was tested using the MySQL DB.

### Example of FlexibleSearchRetentionRule

```

# FS RULE
insert_update FlexibleSearchRetentionRule;code[unique=true];searchQuery;actionReference;
; fsrule; select {PK}, {itemtype} from {Order} where {code} like 'fsorder%' and {expirationTime} < (c

# FS JOB
INSERT_UPDATE RetentionJob;code[unique=true];retentionRule(code);batchSize
; fsretentionJob; fsrule; 100

```

```

# FS CRON JOB
INSERT_UPDATE CronJob;code[unique=true];job(code);sessionLanguage(isoCode)[default=en]
; fsretentionCronJob; fsretentionJob;

# ORDERS
INSERT_UPDATE Order;code[unique=true];user(uid);date[dateformat=dd.MM.yyyy HH:mm];expirationTime[date
"%#groovy%
import java.util.Date
import groovy.time.TimeCategory

String lines = ''
Date now = new java.util.Date()
String format = "dd.MM.yyyy HH:mm:ss"
String creationDate = now.format(format, TimeZone.getTimeZone('CET'))
for(i = 0; i < 10; i++) {
    use( TimeCategory ) {
        after = now - i.days
    }
    String expirationTime = after.format(format, TimeZone.getTimeZone('CET'))
    lines += 'fsorder' + i + ';' + user1; + creationDate + ';' + expirationTime + ';EUR\n'
}
impex.info('importing Orders: +\n' + lines)
reader = new CSVReader(lines)
reader.setMaxBufferLines(100000)
impex.includeExternalData(reader)
";

```

#### Execute the cron job using the scripting console (groovy)

```

import de.hybris.platform.core.Registry
import de.hybris.platform.servicelayer.cronjob.CronJobService
CronJobService cronJobService = Registry.getApplicationContext().getBean("cronJobService")
cronJobService.performCronJob(cronJobService.getCronJob("fsretentionCronJob"))

```

## Examples for Testing

### AfterRetentionCleanupRule

#### i Note

Set `impex.legacy.scripting` to `false`. In SAP Commerce Administration Console, check the **Enable code execution** box in the **Settings** section of the **Impex Import** tab.

#### Example of AfterRetentionCleanupRule

```

# CUSTOMER
INSERT_UPDATE Customer;uid[unique=true];customerID;name;
;user1;user1;name;

# RETENTION RULE
insert_update AfterRetentionCleanupRule;code[unique=true];retirementItemType(code);itemFilterExpressi
    ; rule; Order; {code} like 'order%'; basicRemoveCleanupAction; 600; Order:expirationTime

# JOB
INSERT_UPDATE RetentionJob;code[unique=true];retentionRule(code);batchSize
; retentionJob; rule; 100

# CRON JOB
INSERT_UPDATE CronJob;code[unique=true];job(code);sessionLanguage(isoCode)[default=en]
; retentionCronJob; retentionJob;

# ORDERS
INSERT_UPDATE Order;code[unique=true];user(uid);date[dateformat=dd.MM.yyyy HH:mm];expirationTime[date
"%#groovy%
import java.util.Date
import groovy.time.TimeCategory

String lines = ''

```

```

Date now = new java.util.Date()
String format = "dd.MM.yyyy HH:mm:ss"
String creationDate = now.format(format, TimeZone.getTimeZone('CET'))

for(i = 0; i < 50; i++) {
    use( TimeCategory ) {
        after30Mins = now - (i * 30).seconds
    }
    String expirationTime = after30Mins.format(format, TimeZone.getTimeZone('CET'))
    lines += 'order' + i + ';' + user1; + creationDate + ';' + expirationTime + ';' + EUR\n'
}
impex.info('importing Orders: +\n' + lines)
reader = new CSVReader(lines)
reader.setMaxBufferLines(100000)
impex.includeExternalData(reader)
";

```

# select {code}, {expirationTime} from {Order} where {code} like 'order%' order by {expirationTime}

#### Execute the cron job using the scripting console (groovy)

```

import de.hybris.platform.core.Registry
import de.hybris.platform.servicelayer.cronjob.CronJobService
CronJobService cronJobService = Registry.getApplicationContext().getBean("cronJobService")
cronJobService.performCronJob(cronJobService.getCronJob("retentionCronJob"))

```

#### FlexibleSearchRetentionRule

##### i Note

The query is DB-dependent (date-time manipulation). In this example, it was tested using the MySQL DB.

```

# FS RULE
insert_update FlexibleSearchRetentionRule;code[unique=true];searchQuery;actionReference;
; fsrule; select {PK}, {itemtype} from {Order} where {code} like 'fsorder%' and {expirationTime} < ((

# FS JOB
INSERT_UPDATE RetentionJob;code[unique=true];retentionRule(code);batchSize
; fsretentionJob; fsrule; 100

# FS CRON JOB
INSERT_UPDATE CronJob;code[unique=true];job(code);sessionLanguage(isoCode)[default=en]
; fsretentionCronJob; fsretentionJob;

# ORDERS
INSERT_UPDATE Order;code[unique=true];user(uid);date[dateformat=dd.MM.yyyy HH:mm];expirationTime[date
"%#groovy%"
import java.util.Date
import groovy.time.TimeCategory

String lines = ''
Date now = new java.util.Date()
String format = "dd.MM.yyyy HH:mm:ss"
String creationDate = now.format(format, TimeZone.getTimeZone('CET'))
for(i = 0; i < 10; i++) {
    use( TimeCategory ) {
        after = now - i.days
    }
    String expirationTime = after.format(format, TimeZone.getTimeZone('CET'))
    lines += 'fsorder' + i + ';' + user1; + creationDate + ';' + expirationTime + ';' + EUR\n'
}
impex.info('importing Orders: +\n' + lines)
reader = new CSVReader(lines)
reader.setMaxBufferLines(100000)
impex.includeExternalData(reader)
";

```

#### Execute the cron job using the scripting console (groovy)

```

import de.hybris.platform.core.Registry
import de.hybris.platform.servicelayer.cronjob.CronJobService
CronJobService cronJobService = Registry.getApplicationContext().getBean("cronJobService")
cronJobService.performCronJob(cronJobService.getCronJob("fsretentionCronJob"))

```

## Examples for Maintenance

You can use the data retention framework for basic maintenance tasks, such as cleaning database tables from old diagnostic records. You can define a particular retention rule, job, cron job, and trigger to keep your data clean.

### AfterRetentionCleanupRule

This rule strictly sticks to a declared type. Set a Date-based property (retirement date attribute) that should be compared when records are selected according to the retention time (counted in seconds). In the following example, the rule filters out and deletes all FINISHED CronJobHistory entries that ended more than 3600 seconds ago. The retention cron job runs every 30 minutes. It uses `basicRemoveCleanupAction` for removing items.

```

# RETENTION RULE
insert_update AfterRetentionCleanupRule;code[unique=true]; retirementItemType(code); itemFilterExpression
; cronJobHistoryCleanupRule; CronJobHistory; {status} IN ( {{ SELECT {pk} FROM {CronJobStatus} where
# JOB
INSERT_UPDATE RetentionJob; code[unique=true]; retentionRule(code); batchSize
; cronJobHistoryRetentionJob; cronJobHistoryCleanupRule; 100
# CRON JOB
INSERT_UPDATE CronJob;code[unique=true]; job(code); sessionLanguage(isoCode)[default=en]
; cronJobHistoryRetentionCronJob; cronJobHistoryRetentionJob;
# Trigger run every 30 minutes
INSERT_UPDATE Trigger; cronJob(code)[unique = true]; second; minute; hour; day; month; year; relative
; cronJobHistoryRetentionCronJob; 0; 30; -1; -1; -1; true; true; -1

```

The filter expression is optional, however it can be useful when you want to filter out specific records.

### FlexibleSearchRetentionRule

This rule is based on a flexible search query. You can flexible-select rows. The query returns a list of PKs from one type. You can use a predefined `current_date` value which returns a current date value during execution.

In the example, the defined rule removes all orders named as `internal` whose expiration date is older than one day.

### Example of AfterRetentionCleanupRule

```

# FS RULE
insert_update FlexibleSearchRetentionRule;code[unique=true];searchQuery;actionReference;
; fsInternalOrderRule; select {PK}, {itemtype} from {Order} where {name} like 'internal%' and {expira
# FS JOB
INSERT_UPDATE RetentionJob;code[unique=true];retentionRule(code);batchSize
; fsInternalOrderRetentionJob; fsInternalOrderRule; 100
# FS CRON JOB
INSERT_UPDATE CronJob;code[unique=true];job(code);sessionLanguage(isoCode)[default=en]
; fsInternalOrderRetentionCronJob; fsInternalOrderRetentionJob;
# TRIGGER
INSERT_UPDATE Trigger; cronJob(code)[unique=true]; cronExpression
; fsInternalOrderRetentionCronJob; 0 */4 * ? * *

```

## Item Locking Service

The Item Locking Service enables you to lock items against modification or removal.

You can use the service to lock any item that is an instance of `ItemModel`. For example, you may need to lock some specific instance of the `User` type - and when it is a customer - their orders, or comments they left about the products they bought.

To be able to lock or unlock items, you must be the admin or belong to `itemLockingGroup`.

This code sample shows how to lock an item:

### → Remember

You can only lock items that are persisted in the database - they must have a non-null PK.

```
UserModel user = modelService.create(UserModel.class);
user.setUid(UUID.randomUUID().toString());
modelService.save(user);

// now we lock the user
itemLockingService.lock(user);
```

For the `AfterRetentionCleanupRuleModel` rule, all locked items are skipped when the rule is being executed. For the `FlexibleRetentionRuleModel` rule, if you now try to remove, for example, the locked `User` item, you get `ModelRemovalException`. Similarly, when you try to edit it, you get `ItemLockedForProcessingException`. If you want to skip locked items to avoid getting exceptions when using `FlexibleRetentionRuleModel`, add a suitable part to the `searchQuery` parameter yourself, for example: "... and (it.sealed=0 or it.sealed is NULL)...".

This code sample shows how to check whether an item is locked:

```
itemLockingService.isLocked(user);

// or directly from model without using the Item Locking Service
user.isSealed();
```

This code sample shows how to unlock an item:

```
itemLockingService.unlock(user);
```

## Enabling Users to Lock Items

By default, only the admin can lock or unlock items. To enable others to perform those functions, add them to `itemLockingGroup`.

This code sample shows how to enable users other than the admin to lock and unlock items:

```
Set<PrincipalGroupModel> oldGroups;
Set<PrincipalGroupModel> groupsWithItemLockingGroup;
final PrincipalGroupModel group = modelService.create(UserGroupModel.class);
group.setUid("itemLockingGroup");
oldGroups = userService.getCurrentUser().getGroups();
groupsWithItemLockingGroup = new HashSet<>(oldGroups);
groupsWithItemLockingGroup.add(group);
userService.getCurrentUser().setGroups(groupsWithItemLockingGroup);

UserModel user = modelService.create(UserModel.class);
user.setUid(UUID.randomUUID().toString());
modelService.save(user);

itemLockingService.lock(user);
```

The admin user is a member of `itemLockingGroup` by default.

## Audit Records of Generic Audit

The Generic Audit feature can track changes in items and it shows those changes as audit records. After you lock items that you track using the Generic Audit, you won't be able to remove their audit records either.

For more information on audit records, see [Generic Audit](#).

## Data Validation Framework

SAP Commerce provides a data validation framework based on the JSR 303 bean validation specification. This allows you to define data validation constraints in an easy and intuitive way.

Key features of the Data Validation framework include the following:

- It is based upon the Hibernate Validator that is the reference implementation of the JSR 303 Bean Validation.
- It includes the `ValidationService` that loads the validation engine with constraints, and invokes its logic.
- It has a `ValidationInterceptor` that hooks the `ValidationService` into SAP Commerce Model life cycles.
- Validation constraints can be created and manipulated in the SAP Commerce Administration Cockpit, and loaded into the validation engine at runtime, and are persisted in the persistence layer.
- The logic is triggered both implicitly by the `ValidationInterceptor` and explicitly by direct calls to the `ValidationService`.
- Validation violations are caught and displayed in an intuitive fashion in cockpits.
- There are multiple levels of violation severity: **Info**, **Warning**, and **Error**.

## Introduction

The purpose of the Data Validation is:

- To offer a framework for defining data validation constraints in easy and intuitive way.
- To validate data before it is saved.
- To notify about any validation violations should they occur.

The validation logic may be triggered:

- Implicitly by the `ValidationInterceptor` that hooks into calls to a Model's save method.
- Explicitly by manually calling `validate` method of the `ValidationService` and passing in a SAP Commerce Model or POJO to be validated.

When some validation violations are found, they are presented to the caller for a resolution. The validation framework does not extend to performing client-side validation, rather the validation is performed on the server side only.

There are three main areas to consider when describing the Data Validation:

- The `ValidationService`: Validation constraint types are defined and data is validated.
- The Administration Cockpit: A front end for managing instantiated validation constraints types.
- Cockpit integration: Providing users with validation feedback in cockpits.

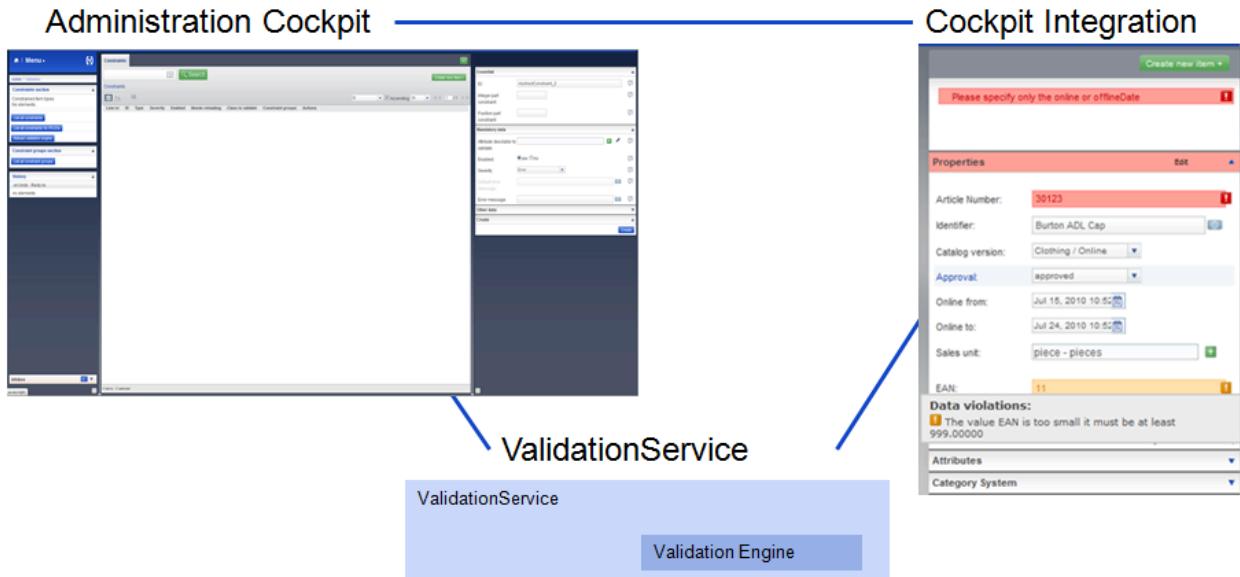


Figure: Different areas related to the Data Validation.

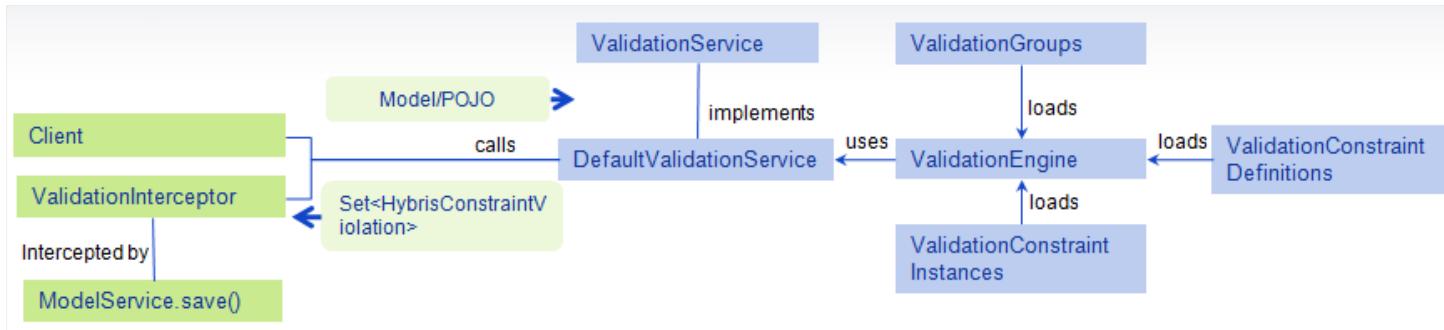
## Validation Architecture

See the architecture of the data validation framework.

The main features include:

- Validation is performed by the `DefaultValidationService` that implements `ValidationService`.
- This service uses a JSR 303-compliant `Validation engine`: Hibernate Validator that performs the validation.
- The validation engine expects:
  - `ValidationConstraint` definitions and instances that are loaded into the validation engine.
  - `Constraint groups` for validating multiple constraints in one invocation.
  - `constraint instances` in the form of annotated java classes, or encoded in an XML file.
- The service can be used both explicitly and implicitly.
  - In both cases, Models or POJOs are passed to the service to be validated and a possibly empty set of `HybrisConstraintViolations` is returned.
  - The service is called implicitly via the `ValidationInterceptor` that intercepts calls made by `ModelService.save()`.
  - The service can be called directly via a normal java call to the service's methods.
- `ValidationService` contains two groups of methods:
  - Validation methods used to validate items, properties, or values.
  - Control methods used to alter a behavior of the validation engine.

See the illustration of the Data Validation architecture.



## Validation Constraints

Validation constraints are regular SAP Commerce Types and are stored in a database like all other Types.

As a consequence they may be created in the Administration Cockpit. When the validation framework starts or is reloaded with the `reloadValidationEngine` method, the following steps are automatically performed:

1. All constraint types are retrieved from the database.
2. They are transformed from SAP Commerce Models to an XML representation expected by the validation engine.
3. They are loaded into the validation engine.

We can modify the SAP Commerce constraints, and again call `reloadValidationEngine` method to reload the validation engine. It allows us to change the validation constraints at runtime. The validation engine loads also any constraints that are annotated directly in Java code, for example:

```
public class User
{
    @NotNull
    private String first_name;
}
```

## Constraint Definition

The Data Validation framework provides implementation of all constraint types defined in the JSR 303 specification. This includes:

- **PatternConstraint**: To validate values using regular expressions.
- **PastConstraint** and **FutureConstraint**: To validate values in comparison with the current date.
- **MinConstraint** and **MaxConstraint**: To validate integer values in comparison with a set value.
- **DecimalMinConstraint** and **DecimalMaxConstraint**: To validate values in comparison with a set decimal value.
- **SizeConstraint**: To validate sizes of strings and collections.
- **AssertTrueConstraint** and **AssertFalseConstraint**: To ensure that a value is true or false.
- **IsNullConstraint** and **IsNotNullConstraint**: To ensure that a value is null or not null.
- **NotEmpty** and **NotBlank**: To ensure that a value is not empty or blank.
- **XorNotNull Annotation**: to ensure that of two given values, one and only one value is set.
- **Dynamic Annotation**: similar to PatternConstraint but evaluates BeanShell code at runtime.

To see how these constraints are represented in SAP Commerce, and how to create constraint types, see the section below.

## Creating Constraints Using Annotation

Constraints can be created by adding JSR303 Annotations to any POJO you may have written.

### SamplePOJO.java

```
class SamplePOJO
{
    @NotNull
    private String value1;

    @Size(min = 3, max = 10)
    private String value2;

    public void setValue1(final String value){
        this.value1 = value;
    }

    public String getValue1(){
        return value1;
    }

    public void setValue2(final String value){
        this.value2 = value;
    }

    public String getValue2(){
        return value2;
    }
}
```

@NotNull annotation for SamplePOJO.value1 has the same effect as creating `IsNotNullConstraint` in the SAP Commerce Administration Cockpit. The similar situation is with the @Size annotation that corresponds to `SizeConstraint`. Annotations may be also used for Models, but it makes no sense, because Models are generated automatically and then annotations are removed. It is possible to create constraints for the same value both using annotations and the Administration Cockpit, for example, `IsNotNullConstraint` for value2. However, constraints created by using annotations are not displayed in the SAP Commerce Administration Cockpit. Nevertheless, they are used in the Data Validation process.

Let us consider the following example. There is a POJO with `@Max(value = 3)` annotation and a `MaxConstraint` with value set to 4 for the same POJO. The instance of this POJO gets the value set to 5. The data validation reports two violations:

- Max violation: **Value is 5 but must be 3.**
- Max violation: **Value is 5 but must be 4.**

## Constraint Type System

The `AbstractConstraint` is the base type for all constraints and it contains methods and fields common for all constraint types. The `AbstractConstraint` contains an `active` attribute that can be used to exclude constraints from the validation process without a need to delete them and a `needReload` attribute, which displays whether the current constraint has already been loaded or not.

Constraints may have different severity levels:

- **Error**
- **Warning**
- **Info**

They can be used to distinguish between various levels of constraint violations that may occur during the validation process. They also have an influence on the validation process. For example, a user is notified about some errors and warnings during the validation and is able to save an item only when errors are corrected.

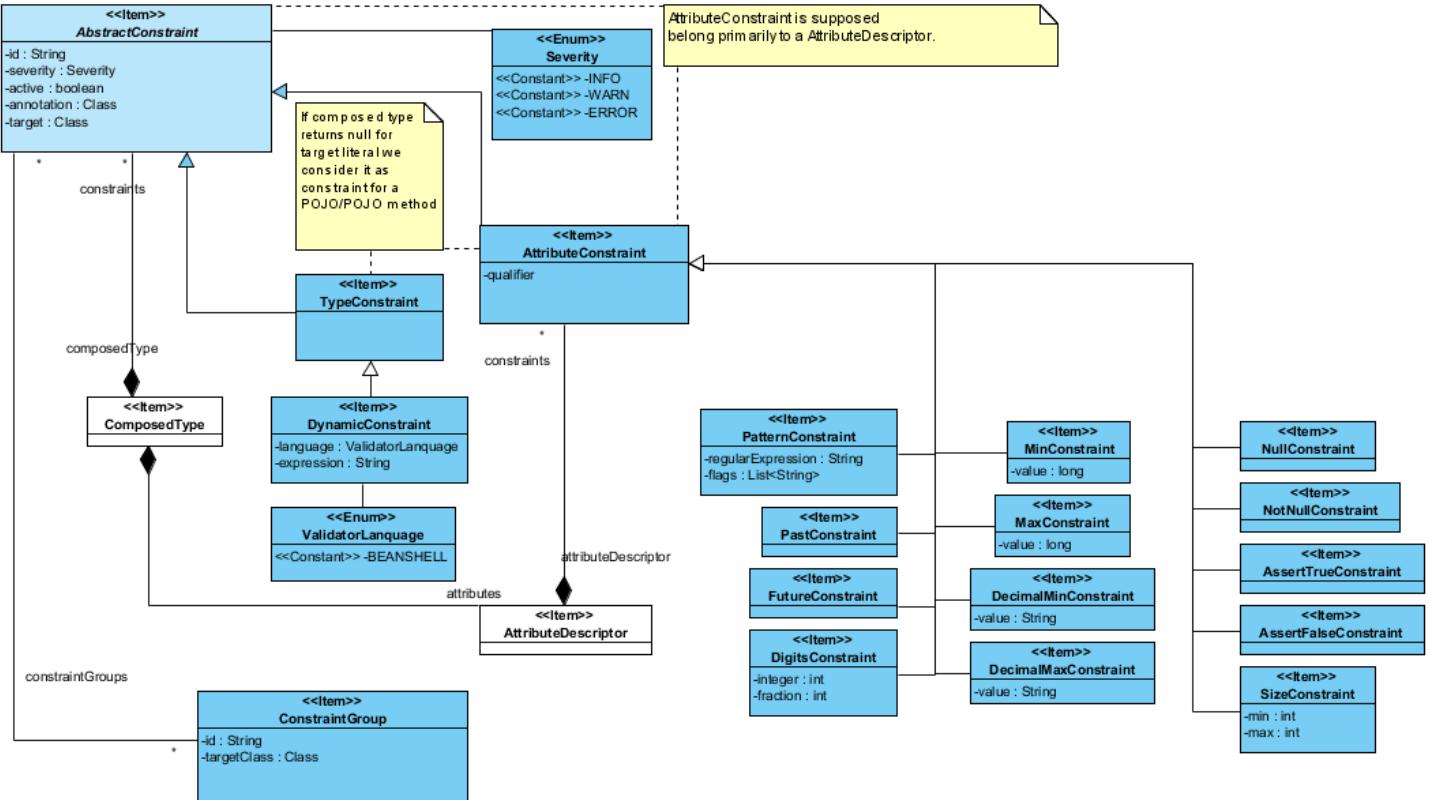


Figure: Hierarchy of constraint types in the Data Model Validation

Every constraint type has a default error message that can be localized in the resources / `<extension_name>/ValidationMessages.properties` file in your extension.

It is common for all instances of a given constraint type. Besides, the default error message, every instance of the constraint type has also the `message` property. You may change its value to localize the message of the instance.

Constraints in the Data Model Validation are divided into two main groups:

- **Attribute** related constraints.
- **Type-related** constraints.

This division reflects differences between various constraint types and allows to maintain order in the code.

The **AttributeConstraint** is a base type for all attribute-related constraints. It extends the **AbstractConstraint** definition with the `qualifier` field and a reference to the **AttributeDescriptor**. The **AttributeConstraint** contains information about a constrained attribute. It is used when you want to validate only one field in a type.

There is also a possibility to validate POJO objects. In this case, the **AttributeDescriptor** and the **ComposedType** in a constraint instance are not used and the `target` attribute is set to a POJO class.

The **TypeConstraint** is a base type for all type-related constraints. It extends the **AbstractConstraint** and has a reference to the **ComposedType**. The **TypeConstraint** contains information about a constrained type. It is used when you want to validate more than one field in a type. For example, the **XorNotNullConstraint**, which extends the **TypeConstraint**, checks whether one of two fields: `code` or `name` is not null.

The **DynamicConstraint** allows to use the BeanShell script language to define a constraint for a type or an attribute. The main advantage of the **DynamicConstraint** is an ability to change validation code without a need of recompilation. For details, see the **Creating Dynamic Constraints** section.

# Constraint Groups

Find out more about constraints groups.

JSR303 Validators have the notion of constraint groups. Unless otherwise specified, all constraints belong to the default constraint group, which the validator uses by default when performing data validation. However, you're able to create, modify, and delete new constraint groups, and assign constraints to them. You may then perform data validation using a specific constraint group. This allows you to group validation rules to be run in different contexts. Constraint groups contain one or more constraints. The default constraint group can't be modified, renamed, or removed. It contains all constraints that aren't assigned to any other constraint group, and is used by default for data validation. However, you can call the validate method and pass in not only the Model or POJO, but also another constraint group. To activate one or more groups, call the `validationService.setActiveConstraintGroups(list of groups)`. Constraint groups can be defined using the SAP Commerce Administration Cockpit. For more information, see [Creating and Modifying Validation Constraints](#).

## Validation Errors

When validation errors occur during the **implicit** validation, a `ModelSavingException` is thrown, to conform with the current `ModelService` behavior.

However, the `ModelSavingException` contains a special `ValidationViolationException` as a cause of error, which can then be extracted and acted upon by the client. The `ValidationViolationException` gives information about constraint violations that occur during the validation process. It contains a message describing problems and a list of `HybrisConstraintViolation` objects, which gives a detailed information about violations.

When validation errors occur during the **explicit** validation, only a list of `HybrisConstraintViolation` is returned by the `ValidationService`.

The `HybrisConstraintViolation` is a wrapper class around `ConstraintViolation` object from the JSR 303 specification. It contains convenient methods that can be used to get information about validation violations. For more details, please refer to the API documentation.

## ValidationService

The `ValidationService` allows you to use the validation engine through a set of validation and control methods.

- Validation methods are used to validate items, properties, or values.
- Control methods are used to alter a behavior of the validation engine.

## Control Methods

- `setActiveConstraintGroups` method allows choosing which constraint group is used for the implicit validation process. The `save` method from the `ModelService` requires only an item as a parameter. When you wish to change a default behavior and implicitly validate all saved items using specific constraint group, call the `setActiveGroups` method from the `ValidationService` with a desired group as the parameter. These settings persist for the current session only. An example:

```
Collection<ConstraintGroupModel> groups;
// add desired constraint groups to groups object
validationService.setActiveConstraintGroups(groups);
```

- `reloadValidationEngine` method informs the validation engine that constraints have changed and reload is needed. Reloading the engine is necessary only when some constraints are added, removed, or edited. It is triggered in the

Administration Cockpit. You can also call this explicitly when performing an explicit validation:

```
validationService.reloadValidationEngine();
```

## Validation Methods

- `validate` method allows you to explicitly validate a Model or POJO.
- `validateProperty` method validates only one property from a Model or a POJO.
- `validateValue` method checks if a given value is valid for the specified property from a Model or POJO. This value, not the property value set in a Model, is checked for validity. To validate an existing Model property, use the `validateProperty` method.

All validation methods allow you to validate a Model or POJO class. A collection of constraint groups can be passed as an additional parameter. When groups are omitted, all active constraints from the default group are considered during the validation process.

All methods from this group return a collection of `HybrisConstraintViolation` objects. When the returned list is empty, the validation succeeded.

## Implicit Validation with the Validation Interceptor

The `ValidationInterceptor` is used to hook into the Model saving process where a model is first validated before it can be saved.

The `ValidationInterceptor` is called just before the Model is saved to the database. Next, it passes the Model to the `ValidationService`, where appropriate validation rules are applied. If the Model is valid, it is saved to the database, otherwise the `ValidationViolationException` is thrown. All Models are implicitly validated before saving to the database. By default all saved items are validated using the `defaultGroup` constraint group. It means that all active constraints that do not belong to any group are in the default group and are used for the validation. When you wish to change a default behavior and implicitly validate all saved items using a specific constraint group, call the `setActiveGroups` method with a desired group as the parameter. If you wish to validate a constraint from another group in addition to constraints that are not assigned to any group, call the `setActiveGroups` method with both groups (default group and the additional group) as parameters. These settings hold for the current session only.

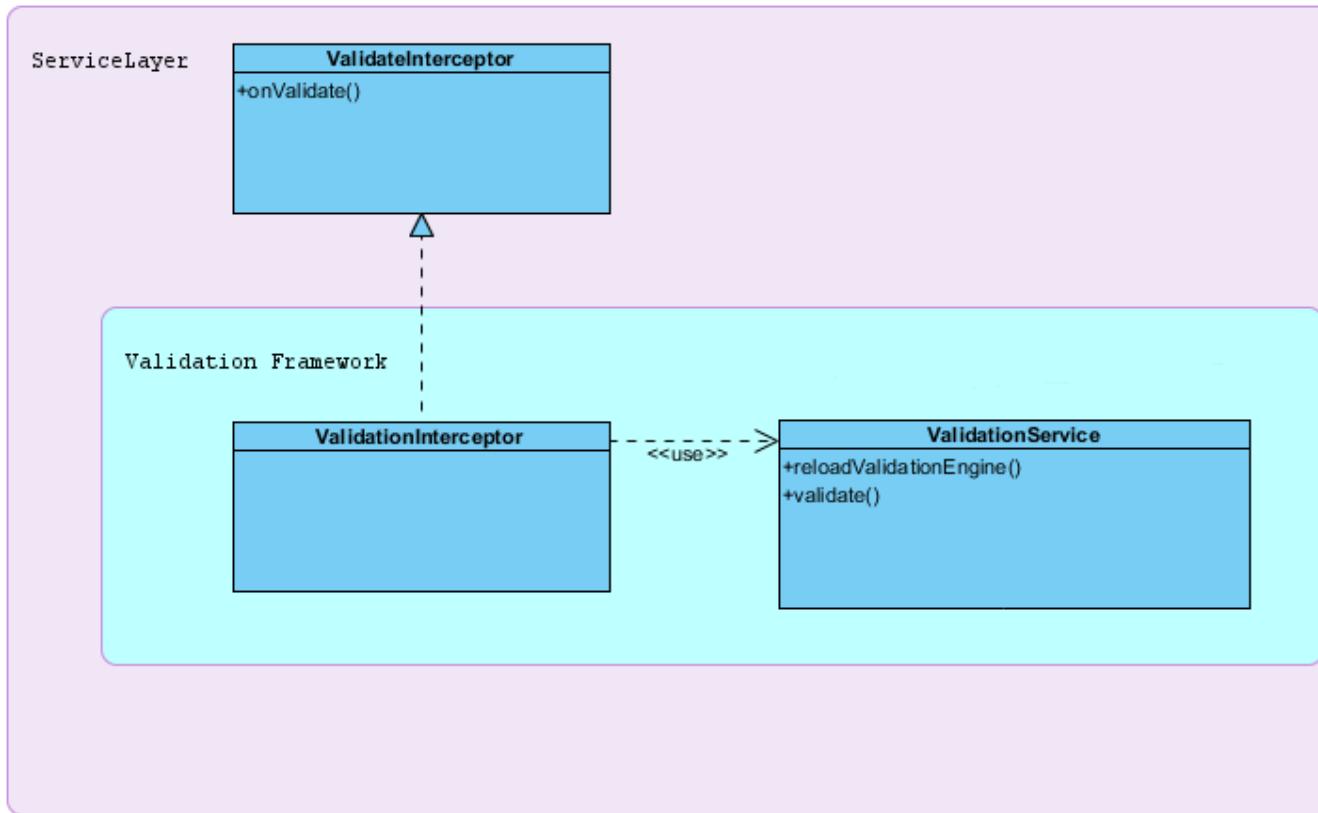


Figure: A diagram presenting how the validation hooks into the ServiceLayer.

## Explicit Validation with Direct Calls

The main usage of the **ValidationService** is for intercepting and validating models that are being saved. However, it is also possible to call the **ValidationService** directly.

All validation methods allow one to validate a Model or POJO class.

Following POJOs are used for a demonstration:

`SamplePOJO.java`

```

class SamplePOJO
{
    private String value1;
    private String value2;

    public void setValue1(final String value){
        this.value1 = value;
    }

    public String getValue1(){
        return value1;
    }

    public void setValue2(final String value){
        this.value2 = value;
    }

    public String getValue2(){
        return value2;
    }
}

```

# Examples of Validation Methods

See examples of validation methods.

- Validating a Model with the default constraint group:

```
ProductModel model = new ProductModel();
Set<HybrisConstraintViolation> constraintViolations =
validationService.validate(model);
```

- Validating a POJO with the default constraint group:

```
SamplePOJO pojo = new SamplePOJO();
Set<HybrisConstraintViolation> constraintViolations =
validationService.validate(pojo);
```

- Validating a Model with custom constraint groups:

```
ProductModel model = new ProductModel();
Collection<ConstraintGroupModel> groups;
// add desired constraints groups to groups object and
// pass it to the ValidationService
Set<HybrisConstraintViolation> constraintViolations =
validationService.validate(model, groups);
```

- Validating a POJO with custom constraint groups:

```
SamplePOJO pojo = new SamplePOJO();
Collection<ConstraintGroupModel> groups;
// add desired constraints groups to groups object and pass it to
// the ValidationService
Set<HybrisConstraintViolation> constraintViolations =
validationService.validate(pojo, groups);
```

- Validating one property from a Model with a default constraint group:

```
ProductModel model = new ProductModel();
String propertyName = "code";
Set<HybrisConstraintViolation> constraintViolations =
validationService.validateProperty(model, propertyName);
```

- Validating one property from a POJO with a default constraint group:

```
SamplePOJO pojo = new SamplePOJO();
String propertyName = "value1";
Set<HybrisConstraintViolation> constraintViolations =
validationService.validateProperty(pojo, propertyName);
```

- Validating one property from a Model with custom constraint groups:

```
ProductModel model = new ProductModel();
String propertyName = "code";
final Collection<ConstraintGroupModel> groups;
// add desired constraints groups to groups object and
// pass it to the ValidationService
Set<HybrisConstraintViolation> constraintViolations =
validationService.validateProperty(model, propertyName, groups);
```

- Validating one property from a POJO with custom constraint groups:

```
SamplePOJO pojo = new SamplePOJO();
String propertyName = "value1";
final Collection<ConstraintGroupModel> groups;
```

```
// add desired constraints groups to groups object and
// pass it to the ValidationService
Set<HybrisConstraintViolation> constraintViolations =
validationService.validateProperty(pojo, propertyName, groups);
```

- Validating one value from a Model with a default constraint group:

```
ProductModel model = new ProductModel();
String propertyName = "code";
String value = "1234";
Set<HybrisConstraintViolation> constraintViolations =
validationService.validateValue(model, propertyName, value);
```

- Validating one value from a POJO with a default constraint group:

```
SamplePOJO pojo = new SamplePOJO();
String propertyName = "value1";
String value = "1234";
Set<HybrisConstraintViolation> constraintViolations =
validationService.validateValue(pojo, propertyName, value);
```

- Validating one value from a Model with a custom constraint groups:

```
ProductModel model = new ProductModel();
String propertyName = "code";
String value = "1234";
final Collection<ConstraintGroupModel> groups;
// add desired constraints groups to groups object and
// pass it to the ValidationService
Set<HybrisConstraintViolation> constraintViolations =
validationService.validateValue(model, propertyName, value, groups);
```

- Validating one value from a POJO with custom constraint groups:

```
SamplePOJO pojo = new SamplePOJO();
String propertyName = "value1";
String value = "1234";
final Collection<ConstraintGroupModel> groups;
// add desired constraints groups to groups object and
// pass it to the ValidationService
Set<HybrisConstraintViolation> constraintViolations =
validationService.validateValue(pojo, propertyName, value, groups);
```

## Validating Data for Multiple Languages

The Platform enables you to validate data in multiple languages before saving it. You can follow the guidelines to create constraints of any available type.

The following example assumes that you create an `IsNotNull` constraint for the `name` attribute of the `Title` type, for the Spanish and English languages. In order to save a new title, fill in the `name` field for both English and Spanish.

## Creating a Constraint

You can easily create a new validation constraint and apply it to multiple languages using the Backoffice Administration Cockpit.

### Procedure

1. Log in to Backoffice.

2. Go to **System > Validation > Constraints**.
3. Click the small arrow next to the **+** icon to open the options menu.
4. Click the arrow next to **Attribute Constraint**, and choose **IsNotNull constraint**.
5. In the window that appears, provide an identifier for your constraint in the **ID** field.
6. Click the **...** icon next to the **Enclosing Type** field, and search for and select the type with identifier **Title**.
7. Click the **...** icon next to the **Attribute descriptor to validate** field, and select name.
8. Click the **...** icon next to the **Validation languages** field, and click the check marks next to English and Spanish in the window that appears to select them.
9. **Optional:** Click next to add additional attributes for your constraint.
10. Click **Done** to save your constraint.

## Creating an Item

After you have created your constraint, create an item.

### Procedure

1. Go to **User > Titles**.
2. Click the **+** icon to add a new item.
3. Complete the **Token** field.
4. Click **Done**.

### Results

If you try to save your item and you haven't provided both English and Spanish names for it, you get an error telling you that these names are required. This message means that validation works and the constraint has been imposed on the title.

## Customizing

Although the validation framework provides all constraints from the JSR 303 specification, sometimes other constraint types may be needed. This chapter presents how to create a custom constraint type.

## Creating Constraint Types

See the examples of how to create constraint types, including parameterized constraints.

### **i Note**

#### **Simple Constraint Types**

This section describes how to create simple constraint types. If you are interested in creating the **DynamicConstraint**, refer to the **Creating Dynamic Constraints** topic.

New constraint types can be defined in the `<extension_name>-items.xml` file in your extension. You may decide which basic classes the new constraint types should extend:

- **AttributeConstraint:** If you wish to add constraints to a specified attribute.

- **TypeConstraint:** If you wish to add constraints to a type (a group of attributes).

Custom constraint type creation consists of the following steps:

1. Define a constraint type in the `<extension_name>-items.xml` file in your extension.
2. Create an annotation class that links the constraint type with a constraint validator.
3. Create the constraint validator class with a custom validation logic.
4. Add a constraint violation message template to the `resources/<extension_name>/ValidationMessages.properties` file.

### i Note

#### Package Name

In the examples from the next section, replace `<YourPackage>` with a package name that you use in your extension.

## Example of Simple Constraint

Learn how to create a simple `NotEmptyConstraint`, which not only checks if a string is null, but also whether a validated string is not empty.

### Constraint Type Definition

First it is necessary to create the constraint type definition in the `<extension_name>-items.xml` file. The definition consists of:

- `code` property: The constraint name
- `extends` property: A base type name
- `jaloClass` property: The name of the Jalo item class (gets generated at first build).
- `annotation` attribute with `defaultValue` property set to the name of the annotation class. Annotation is a link between the constraint type and a constraint validator that is used to validate data. To get more information about how to create an annotation, see the **Annotation** section. The constraint validator contains code that is executed when a constrained field or type is being validated.

#### items.xml

```
<itemtype code="NotEmptyConstraint" autocreate="true" generate="true"
extends="AttributeConstraint"
jaloClass="<YourPackage>.NotEmptyConstraint"
<description>Custom constraint which checks for empty strings
(Apache commons implementation)</description>
<attributes>
    <attribute qualifier="annotation" type="java.lang.Class"
    redeclare="true">
        <modifiers write="false" initial="true" optional="false"/>
        <defaultValue>
            <code><YourPackage>.NotEmpty.class
        </defaultValue>
    </attribute>
</attributes>
</itemtype>
```

### Annotation

The next step is to create an annotation class to link the created constraint type with the constraint validator. The annotation class presented below may be used as a template for creating annotations. An annotation type is defined using the `@interface` keyword. All attributes of an annotation type are declared in a method-like manner. The JSR 303 specification demands, that any constraint annotation defines:

- The `message` attribute: Returns the default key for creating error messages if the constraint is violated.
- The `groups` attribute: Allows the specification of validation groups, to which this constraint belongs. The default value is an empty array of type `Class<?>`.
- The `payload` attribute: Can be used to assign custom payload objects to a constraint. This attribute is not used by the API itself.

Moreover, there are some rules that should be obeyed:

- The value of `@Target` annotation should be set to `FIELD` when creating the `AttributeConstraint`.
- The value of `@Target` annotation should be set to `TYPE` when creating the `TypeConstraint`.
- The default value of `message()` method should be set to key from the `resources/ <extension_name>/ValidationMessages.properties` file of your extension.

### NotEmpty.java

```
@Target({ FIELD })
@Retention(RUNTIME)
@Constraint(validatedBy = NotEmptyValidator.class)
@Documented
public @interface NotEmpty
{
    String message() default "{<YourPackage>.NotEmpty.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

### Constraint Validator

The constraint validator contains the validation logic. Its `isValid` method inherited from the `ConstraintValidator` interface from the JSR 303 specification is invoked whenever a Model containing constrained value is validated. Constraint validator definition consists of the annotation class, in our case `NotEmpty`, and a class of attribute being validated, in that case `String`. The same class is used in `isValid` method.

### NotEmptyValidator.java

```
public class NotEmptyValidator implements ConstraintValidator<NotEmpty, String>
{
    @Override
    public void initialize(final NotEmpty constraintAnnotation)
    {
        // empty
    }

    @Override
    public boolean isValid(final String value,
    final ConstraintValidatorContext context)
    {
        return !StringUtils.isEmpty(value);
    }
}
```

### Error Message

Insert an error message to the `resources/ <extension_name>/ValidationMessages.properties` file of your extension. It is displayed by default for every instance of a constraint type. There is also a possibility to override this message, but it should be done for each instance separately. The common practice is to use a fully qualified class name with `.message` suffix as a default value.

## ValidationMessages.properties

```
<YourPackage>.NotEmpty.message='{type}.{field}' can not be empty.
```

# Example of Parametrized Constraint

Learn how to add a parametrized `StringLengthConstraint` that checks if a given string length is in specified range.

The procedure is the same as for creating simple constraints but this time the new constraint type brings own attributes. Furthermore, a configuration for managing these attributes should be added for the Administration Cockpit additionally.

## Constraint Type Definition

Parametrized constraint definition extends the definition of a simple constraint type from the previous example with additional attributes: `min` and `max`.

### items.xml

```
<itemtype code="StringLengthConstraint" autocreate="true" generate="true"
extends="AttributeConstraint"
jaloclass=".StringLengthConstraint">
<description>Custom constraint which checks if string's length is
in specified range.</description>
<attributes>
    <attribute type="java.lang.Integer" qualifier="min">
        <description>Underflow value</description>
        <modifiers read="true" write="true" search="true" optional="false"
initial="true" />
        <persistence type="property" />
    </attribute>
    <attribute type="java.lang.Integer" qualifier="max">
        <description>Overflow value</description>
        <modifiers read="true" write="true" search="true" optional="false"
initial="true" />
        <persistence type="property" />
    </attribute>
    <attribute qualifier="annotation" type="java.lang.Class" redeclare="true">
        <modifiers write="false" initial="true" optional="false"/>
        <defaultvalue>
            <YourAnnotationsPackage>.StringLength.class
        </defaultvalue>
    </attribute>
</attributes>
</itemtype>
```

## Annotation

Annotation definition from the previous example is extended with methods for additional parameters. These additional parameters are then used by the `ConstraintValidator` described below. Note that in annotation definition only primitive types like `String`, `Class`, annotations, enumerations, or 1-dimensional arrays are allowed. You can also configure here default values.

### StringLength.java

```
@Target({ FIELD })
@Retention(RUNTIME)
@Constraint(validatedBy = StringLengthValidator.class)
@Documented
```

```
public @interface StringLength
{
    String message() default "{<YourPackage>.StringLength.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
    int min() default 0;
    int max() default Integer.MAX_VALUE;
}
```

## Constraint Validator

Constraint validator definition from the section **Example of Simple Constraint** is extended with:

- Additional parameters
- `initialize` method to gather parameters from the `constraintAnnotation`
- Optional `validateParameters` method

### StringLengthValidator.java

```
public class StringLengthValidator implements ConstraintValidator<StringLength, String>
{
    private int min;
    private int max;

    @Override
    public void initialize(final StringLength constraintAnnotation)
    {
        min = constraintAnnotation.min();
        max = constraintAnnotation.max();
        validateParameters();
    }

    @Override
    public boolean isValid(final String value, final ConstraintValidatorContext constraintValidatorContext)
    {
        if (value == null) // According to the JSR 303 null values are valid
        {
            return true;
        }
        final int length = value.length();
        return length >= min && length <= max;
    }

    private void validateParameters()
    {
        if (min < 0)
        {
            throw new IllegalArgumentException("The min parameter cannot be negative.");
        }
        if (max < 0)
        {
            throw new IllegalArgumentException("The max parameter cannot be negative.");
        }
        if (max < min)
        {
            throw new IllegalArgumentException("The length cannot be negative.");
        }
    }
}
```

## Error Message

In error messages, you can address constraint parameters using their names.

## ValidationMessages.properties

```
<YourPackage>.StringLength.message='{type}.{field}' length must be between
{min} and {max}.
```

## Configuration in the Administration Cockpit

An additional XML file is needed for Administration Cockpit to properly display new constraint types. The configuration file should be located in the /resources/admincockpit/config directory:

### Editor\_StringLengthConstraint\_Admin.xml

```
<editor>
    <group qualifier="AttributeConstraint" visible="true" initially-opened="true">
        <label lang="de">Obligatorische Daten</label>
        <label lang="en">Mandatory data</label>
        <property qualifier="AbstractConstraint.id" />
        <property qualifier="AttributeConstraint.descriptor" />
        <property qualifier="AbstractConstraint.active" />
        <property qualifier="AbstractConstraint.severity" />
        <property qualifier="StringLengthConstraint.min" />
        <property qualifier="StringLengthConstraint.max" />
        <property qualifier="AbstractConstraint.defaultMessage" />
        <property qualifier="AbstractConstraint.message" />
    </group>
    <group qualifier="admin" visible="true" initially-opened="false">
        <label lang="de">Andere Daten</label>
        <label lang="en">Other data</label>
        <property qualifier="AbstractConstraint.type" />
        <property qualifier="AbstractConstraint.target" editor="class" />
        <property qualifier="AttributeConstraint.qualifier" />
        <property qualifier="AbstractConstraint.constraintGroups" >
            <parameter>
                <name>allowCreate</name>
                <value>true</value>
            </parameter>
            <parameter>
                <name>defaultEmptyLabelKey</name>
                <value>ea.constrain_group_default_label</value>
            </parameter>
        </property>
        <property qualifier="AbstractConstraint.annotation" editor="class"/>
        <property qualifier="Item.pk" />
        <property qualifier="Item.creationTime" />
        <property qualifier="Item.modifiedtime" />
    </group>
</editor>
```

## Creating Constraint Groups

Constraint groups may be created using Java code.

Each constraint group should have a unique ID. Thus, each instance of the `ConstraintGroup` should use a unique interface as a marker. The interface is automatically generated during runtime basing on the unique group ID. However, it is possible to provide the interface name that should be used. It can be an existing interface class or a nonexisting class, because it can be generated. There exists a default constraint group, which can't be modified, removed, or assigned to constraints. Each constraint that is not assigned to any other group belongs to the default group.

To get constraints that are in the default group, use `validationService.getDefaultconstraintGroup()` method, as in the example.

```
AssertTrueConstraintModel constraint = modelService.create(AssertTrueConstraintModel.class)
constraint.setId("trueconstraint");
//set attribute descriptor but no group here!
```

```

modelService.save(constraint);

constraint.getConstraintGroups(); //returns an empty set
validationService.getDefaultconstraintGroup().getConstraints(); //return a collection with the Assert

constraint.setConstraintGroups(SelfDefinedGroup);
validationService.getDefaultconstraintGroup().getConstraints(); //now it returns an empty collection!

```

The following code snippet presents how to create constraint groups in Java code.

```

Set<AbstractConstraintModel> constraints;
// add desired constraints to set;
ConstraintGroupModel group1 = new ConstraintGroupModel();
group1.setId("group1");
group1.setConstraints(constraints);
modelService.save(group1);
ConstraintGroupModel group2 = new ConstraintGroupModel();
group2.setId("group2");
group2.setInterfaceName("non.existing.Interface"); // it is important to use
// different interface marker for each group
group2.setConstraints(constraints);
modelService.save(group2);

```

For details see the **Constraint Groups** section above.

## Creating Dynamic Constraints

The main advantage of the **DynamicConstraint** is that validation business logic can be written in Java dynamically without a need for recompilation. This is possible thanks to the BeanShell interpreter that can execute Java source code at runtime.

To create a **DynamicConstraint**, you may use the SAP Commerce Administration Cockpit.

For more information, see [Administration Cockpit Tutorial - Chapter \(2\)](#): section **Dynamic Constraints**.

The script body for the **DynamicConstraint** attached to the **Product** checks if the **Code** is equal to the attribute **Name** for the current context language:

```
return getCode().equals(getName());
```

An instance, which gets validated is implicitly in context. It means that in our example **getName()** and **getCode()** methods are executed directly on the validated **Product** instance. A result of dynamic constraint validation depends on values of attributes of instance being validated.

Hints for writing BeanShell snippet:

1. General regulation to write **DynamicConstraint** script, is that the expression must return object, which somehow can be evaluated as a boolean **true** or **false** value (logic or textual).
2. Implicitly script is performed when the validation object is passed as the **this** object in Java meaning. All methods from the constrained Model can be invoked as **get(Property)**, **is(Property)**.
3. You can also access implicit context object **ctx**, which is an instance of the **org.springframework.context.ApplicationContext**. It gives the possibility to access Spring-based beans, services.
4. Be aware of runtime exceptions, for example **NullPointerException** when invoking method of the null object.

Another example of BeanShell expression uses the ServiceLayer's **I18Nservice**. It compares the **isocode** of **LanguageModel** with the **Locale**'s language of the current session:

```
java.util.Locale loc =  
ctx.getBean(de.hybris.platform.servicelayer.i18n.I18NService.class).getCurrentLocale();  
return !loc.getLanguage().equals(getIsocode());
```

The code snippet below shows how to create `DynamicConstraint` using Java:

```
DynamicConstraintModel constraint = new DynamicConstraintModel();  
constraint.setActive(true);  
constraint.setLanguage(ValidatorLanguage.BEANSHELL);  
constraint.setExpression("return false;");
```