



Platform, Services, and Utilities

Generated on: 2024-11-08 15:06:55 GMT+0000

SAP Commerce | 2205

PUBLIC

Original content: 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>.

JMX Monitoring in SAP Commerce

SAP Commerce implements Java Management Extensions (JMX) to allow the remote observation and management of your installation, allowing you to call methods within a running application for direct application monitoring.

Caution

Ensure that the JMX remote access functionality can only be accessed by authorized users with a secure (TLS) channel. Do not rely on default passwords or keys. For information about configuring JMX in a secure manner, see [Secure Configuration](#).

The Java Virtual Machine (JVM) comes with several MBeans already installed and activated that allow you to observe and update certain parameters in a JVM without custom implementation. In addition to these default MBeans, SAP Commerce provides the following:

- Custom MBeans providing you with the ability to observe and change additional VM parameters using JMX.
- Guidelines for you to write your own MBeans in custom extensions.
- Support for the Tanuki Java Service Wrapper.

With this suite of functionality, you can perform some of the following tasks:

- Display detailed information on your database environment. For example, details on the database, data sources, Java environment and login data.
- Display detailed information on connections. For example, maximum number of in-use connections, number of open JDBC connections, and connection state.
- Check the health of your production system.
- Analyze statistics concerning the usage of certain parts of your system.
- Analyze scheduled jobs.
- Display detailed information on the system cache.
- Remotely manage cache statistics.
- Monitor the usage of the ServiceLayer. Specify which methods to monitor, obtain information such as how often a method was called, and view statistics on how long that message takes to process.
- Stop and start the SAP Commerce MBeans at run time.

Enabling JMX Remote Access

Follow the steps to enable JMX remote access.

Context

JMX remote access functionality in SAP Commerce is disabled by default.

If you want to use the JMX remote access functionality, make sure that it can only be accessed by authorized users, via a secure (TLS) channel. Do not rely on default passwords or keys. For information about configuring JMX in a secure manner, see [Secure Configuration](#).

Procedure

1. Go to `<HYBRIS_CONFIG_DIR>` and edit your `local.properties` file.

2. In your properties file, edit the `tomcat.generaloptions` property and add the following options to it:

```
-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.ma
```

3. Save your properties file.

4. Run ant server.

Results

You have enabled the JMX remote access functionality.

SAP Commerce JMX MBeans Reference

An overview of the SAP Commerce MBeans available for monitoring and managing the system over JMX.

Get a list of available JMX MBeans by going to in the Administration Console.

DataSourceOverviewMBean

Displays information about the current master data source.

Attributes

Property	Description
DatabaseName	Shows the name of the database.
DatabaseURL	Shows the URL of the database.
DatabaseUser	Shows the database user.
DatabaseVersion	Shows the version of the database.
DriverVersion	Shows the driver version of the database.
ID	Shows the ID of the current data source.
JNDIName	Shows the Java Naming and Directory Interface name.
LoginTimeout	Shows the login timeout. A value of zero means that the timeout is the default system timeout.
MaxAllowedPhysicalOpen	Shows the maximum number of objects that can be allocated by the connection pool.
MaxInUse	Shows the maximum number of the in use connection.
MaxPhysicalOpen	Shows the maximum number of physical SQL connections.
MaxPreparedParameterCount	Shows the maximum allowed number of parameters within one prepared statement; -1 if no limit exists.
MillisWaitedForConnection	Shows the time in milliseconds how long it took to get a connection from the pool.
NumInUse	Shows the number of instances currently borrowed from the pool.

Property	Description
NumPhysicalOpen	Shows the number of currently open JDBC connections.
SchemaName	Shows the schema name of the database.
CanConnectToDataSource	Shows true if a connection to the pool is possible.
TablePrefix	Shows the table prefix for all tables.
ReadOnly	Shows true if the data source is in read-only mode.
Active	Shows true if the current data source is active.
AllDataSourceIDs	Shows a list with all available data source IDs.
Connections	Shows the total number of connections so far.

Operations

Operation Name	Description
resetStats	Resets the values for MillisWaitedForConnection, MaxInUse and Connections.

CronJobInfo

Displays information about **CronJobs**.

Attributes

Property	Description
RunningCronJobs	Provides a list with current running CronJobs (the CronJob code is displayed).

Operations

Operation Name	Description
abortRunningCronJobs	Aborts current running CronJobs.

MainCacheMBean

Shows statistics from the SAP Commerce Main Cache.

Attributes

Property Name	Description
CurrentCacheSize	Shows the current size of the internal cache in entries .
CurrentCacheSizeInPercent	Shows the current size of the internal cache (in percent, related to MaximumCacheSize).

Property Name	Description
EntitiesAddCount	Shows how many entities were added to the cache since creation or last clear.
EntitiesGetCount	Shows how many entities were requested from the cache since creation or last clear.
EntitiesMissCount	Shows how many entities were requested but wasn't in the cache since last clear.
EntitiesMissCountInPercent	Shows in percent how many entities were requested but weren't in the cache.
EntitiesRemoveCount	Shows how many entities were removed from the cache since creation or last clear.
MaxReachedCacheSize	Shows the maximum reached number of entries since creation of the cache. This is only reseted if clearCache() is called.
MaxReachedCacheSizeInPercent	Shows the maximum reached number of entries in percent since creation of the cache or last clear (related to MaximumCacheSize).
MaximumCacheSize	Shows the maximum cache size. This is the number of cache entries that are allowed until the cache mechanism removes the least recently used entries before adding new ones.

Operations

Operation Name	Description
clearCache()	Clears the internal cache and the cache statistics.
startCacheStatistics()	Starts collection of cache statistics.
stopCacheStatistics()	Stops collection of cache statistics.
showCacheStatistics(lowerBound, upperBound)	Shows the cache statistics which were collected by using startCacheStatistics. To display all statistics lowerBound must be 0 and upperBound must be 100.

FlexibleQueryCacheMBean

Shows cache statistics of the Flexible Query Cache.

Attributes

Property Name	Description
CurrentSize	Shows the current entities size of the flexible query.
CurrentSizeInPercent	Shows the current entities size in percent of the flexible query.
MaxSize	Shows the maximum entities size of the flexible query.

EntityRegionCacheMBean

Displays information about the default cache region for caching entity data (items).

Attributes

Property Name	Description
CacheFillRatio	Ratio between the current and the maximum available cache size, in 0 - 100 percent.
CurrentCacheSize	Size of elements currently in the cache.
EvictionCount	The number of times elements were evicted from the cache. This can happen explicitly (elements become invalid) as well as implicitly (cache is full and elements need to be removed to free up space).
FetchCount	The number of times element value needed to be computed due to not being in the cache. This should be quite close to MissCount (slightly larger in case of many parallel computations happening for the same element).
HitCount	The number of times the cache held the value for a queried element and therefore computation was not necessary.
HitRatio	Ratio between the number of times element values were already present and the overall number of times the cache was accessed, in 0 - 100 percent.
InvalidationCount	The number of times an element was removed from cache due to cache invalidation being received. Those are triggered by CRUP write operations throughout the whole cluster!
MaximumCacheSize	The maximum allowed number of elements in that cache. i Note The 'unlimited' cache region returns 0 here!
MissCount	The number of times element value had to be computed due to not being in the cache. Other than FetchCount, this figure doesn't count parallel computations by multiple callers where finally all callers reuse a single value.

QueryRegionCacheMBean

Displays information about the cache region responsible for caching all FlexibleSearch query results.

Attributes

Property Name	Description
CacheFillRatio	Ratio between the current and the maximum available cache size, in 0 - 100 percent.
CurrentCacheSize	Size of queries currently in the cache.
EvictionCount	The number of times queries were evicted from the cache. Queries are always evicted implicitly (cache is full and elements need to be

Property Name	Description
	removed to free up space).
FetchCount	The number of times a query result needed to be computed due to not being in the cache. This should be quite close to MissCount (slightly larger in case of many parallel computations happening for the same query).
HitCount	The number of times the cache held the result for a query and therefore computation was not necessary.
HitRatio	Ratio between the number of times element results were already present and the overall number of times the cache was accessed, in 0 - 100 percent.
InvalidationCount	The number of times a query was removed from cache due to cache invalidation being received.
MaximumCacheSize	The maximum allowed number of queries in that cache.
MissCount	The number of times a query result had to be computed due to not being in the cache. Other than FetchCount, this figure doesn't count parallel computations by multiple callers where finally all callers reuse a single value.

Custom JMX Beans

SAP Commerce provides tools to support adding custom JMX MBeans to fulfil your own application monitoring needs.

The **MBeanRegisterUtilities** is a SAP Commerce bean which deals with the registration or deregistration of MBeans which extend the **AbstractJMXMBean** class. Any class which does not extend this class is not recognized.

MBeanRegisterUtilities contains the methods `getRegisteredBeans()` and `getUnregisteredBeans()`, which give an overview of registered and unregistered beans. The return value of those methods is a map: `beanid > theBeanObject`.

You can add or remove MBeans in running environment using `registerBeans(beanmap)` and `unregisterBeans(beanmap)`. The result is adding or removing the beans in the JConsole in real time.

AbstractJMXMBean and ObjectName

AbstractJMXMBean is an abstract class which must be implemented by any custom MBean. Without it, the MBean is not recognized by the **MBeanRegisterUtilities** class.

An MBeanServer instance interacts with the MBeans registered with it via their **ObjectName**. The **ObjectName** must be unique within the system.

Introducing JMX Beans into SAP Commerce

Enable JMX monitoring, and activate MBeans with the SAP Commerce Administration Console.

Procedure

1. Create an interface with the signatures that you want to provide in JConsole.
2. Implement the interface, extending `AbstractJMXMBean`.
3. Add the annotation `@ManagedResource` for the class, and either `@ManagedOperation` or `@ManagedAttribute` for the class methods, depending what you want to provide.
4. Use the annotation attributes `description="your text"` for displaying more information in JConsole.
5. Register this class as a bean in your spring configuration.

```
package de.hybris.mbeans.*;

public interface MyJMXBean
{
    void clear()

    Integer getMaxAllowedSize();
}
```

6. Create your MBean class, extending `AbstractJMXMBean`.

```
package de.hybris.mbeans.impl.*;

@ManagedResource(description = "my first JMX Bean")
public class MyJMXBeanImpl extends AbstractJMXMBean
{
    @ManagedOperation(description = "Clears the internal cache.")
    public void clear() //this method returns nothing
    {
        new TenantAwareExecutor()
        {
            @Override
            protected Object doExecute()
            {
                Registry.getCurrentTenant().getCache().clear();
                return null; //therefore no need to return anything here
            }
        };
    }

    @ManagedAttribute(description = "Shows the upper limit for the cache. This is the number of
        + "cache entries that are allowed until the cache mechanism will remove the least recentl
        + "used entries before adding new ones.")
    public Integer getMaxAllowedSize()
    {
        return new TenantAwareExecutor<Integer>()
        {
            @Override
            protected Integer doExecute()
            {
                return Integer.valueOf(Registry.getCurrentTenant().getCache().getMaxAllowedSize());
            }
        }.getResult();
    }
}
```

A method is exposed as an operation only if it fulfills the following requirements:

- It is annotated with `@ManagedOperation`.
- It is not a JavaBean getter or setter.

7. Register the MBean in your `spring.xml` file.

```
<bean id="myjmxbean" class="de.hybris.mbeans.impl.MyJmxBeanImpl">
    <property name="jmxPath" value="cache=Main Cache"/>
    <property name="jmxDomain" value="hybris.de"/>
    <!-- if necessary add own properties here -->
</bean>

<!-- example for the class above -->
<bean id="myjmxbean" class="de.hybris.platform.jmx.mbeans.impl.MyJMXBeanImpl">
```

```
<property name="jmxDomain" value="example.com"/>
<property name="jmxPath" value="example = JMX Bean examples, ex1=My JMX Bean example"/>
</bean>
```

The `jmxPath` property is mandatory. With this `jmxPath` the unique `ObjectName` is created. You have to add at least one key/value pair, separated by '='. You can add other pairs as a comma-separated list'. For example, `key1=value1, key2=value2`.

`jmxDomain` is an optional attribute. The default domain is `hybris`, but you can use your own domain.

Activating MBeans in the Administration Console

You can activate or deactivate the available JMX monitoring MBeans in the SAP Commerce Administration Console.

Procedure

1. Start the Platform.
 2. Open the SAP Commerce Administration Console and go to **JMX MBeans**.
- You see an overview of the currently registered and not-yet registered MBeans.
3. Set each switch to the on position to register all MBeans.
 4. Open your preferred JMX monitoring tool and verify that the changes have taken effect.

Profiling ServiceLayer Usage Using AOP and JMX

You can specify which service layer methods are accessible via JMX by defining a template interface using the principles of AOP.

SAP Commerce uses a Spring-based aspect to define which packages, classes, or methods are profiled. The profiling logic is configured using a Spring AOP definition. You use an aspect for defining which packages, classes, or methods are profiled and which are not. The collected information is then accessible using JMX.

The aspect is wired using the configuration file called `core-profiling-spring.xml` as in the following example.

```
<!-- Profiler Aspect -->
<!--
    Sample 'pointcut' expressions:
    execution(* set*(*) )
        This pointcut matches a method-execution join point, if the method
        "set" and there is exactly one argument of any type
    within(com.company.*)
        This pointcut matches any join point in any type in the com.company
        The * is one form of the wildcards that can be used to match many
        execution(*      set*(*) ) && this(Point) && within(com.company.*)
        This pointcut matches a method-execution join point, if the method
        "set" and this is an instance of type Point in the com.company pac
        It can be referred to using the name "set()".
    Property:      limit
    Description:   If the execution time is lesser than the specified value, the metho
    Property:      aspectnodeName
    Description:   JMX node name (default: aspect class name)
    Property:      domain
    Description:   JMX domain name (default: JmxUtils.DEFAULT_JMX_DOMAIN)
    Property:      accuracy
    Description:   time measure accuracy (use 'ns' for nano seconds or 'ms' for milli
```

```

Property: assembler
Description: the assembler is used for gathering required MBeanInfo object for a

-->
<bean id="serviceLayerProfiler" class="de.hybris.platform.aop.ProfilingAspect">
    <property name="template" value="de.hybris.platform.jmx.mbeans.impl.ProfilerReportPOJO" />
    <property name="limit" value="0" />
    <property name="aspectNodeName" value="" />
    <property name="domain" value="" />
    <property name="accuracy" value="" />
</bean>

<aop:config proxy-target-class="true">
    <aop:aspect id="serviceLayerProfilerAspect" ref="defaultProfiler" >
        <!-- pattern which defines the methods for profiling -->
        <aop:pointcut id="profiledMethods" expression="execution(* de.hybris.platform.serv:>
        <!-- 'logExecutionTime' will be called for every match -->
        <aop:around pointcut-ref="profiledMethods" method="logExecutionTime" />
        <!-- exception profiling -->
        <aop:after-throwing pointcut-ref="profiledMethods" method="logException" />
    </aop:aspect>
</aop:config>

```

To use this feature, ensure that the AOP and Bean definitions in `core-profiling-spring.xml` are activated. By default, the configuration lines of these beans are disabled by comment tags.

Define Which Methods to Profile

The code snippet below shows how you define that all methods inside the ServiceLayer package should be profiled.

`core-profiling-spring.xml`

```
<aop:pointcut id="profiledMethods" expression="execution(* de.hybris.platform.servicelayer..*.*(..))
```

Define Which Methods are Available Over JMX and Fill Them with Live Data

To define which methods are accessible via JMX, use `ProfilingAspect` as the profiler. This aspect is responsible for logging the execution time and submitting the data to the configured template reference. The template instance implements `ProfilingReportBean`, and specifies which methods are accessible using JMX.

To use your own implementation of `ProfilingReportBean`, override the `Object logExecutionTime(final ProceedingJoinPoint pjp)` method of the `de.hybris.platform.aop.AbstractProfilingAspect`, as follows:

In the `core-profiling-spring.xml` file:

```
<bean id="defaultProfiler" class="de.hybris.platform.aop.ProfilingAspect">
    <property name="template" value="de.hybris.platform.jmx.mbeans.impl.ProfilerReportPOJO" />
    <property name="limit" value="0" />
</bean>
```

And for the `de.hybris.platform.jmx.mbeans.ProfilingReportBean`:

```
/**
 * This interface will be used for specifying all methods, v
 */
public interface ProfilingReportBean extends ProfilingReport {
    /**
     * returns the amount of failed method executions
     */
}
```

```

        * @return the amount of failed method executions
        */
        long getFailedExecutions();

        /**
         * returns max execution time of the profiled method
         * @return max. execution time
         */
        long getMaxExecutionTime();

        /**
         * returns min execution time of the profiled method
         * @return min. execution time
         */
        long getMinExecutionTime();

        /**
         * returns the total number of method calls of the profiled
         * @return total of methods calls so far
         */
        long getTotalCount();

        /**
         * returns the total execution time of the profiled method
         * @return total time
         */
        long getTotalTime();
    }
}

```

And then for the ProfilerReportPOJO implementation:

```

public class ProfilerReportPOJO implements ProfilingReportBean
{
    private long totalCount = 0;
    private long totalExecutionTime = 0;
    private long maxExecutionTime = 0;
    private long minExecutionTime = -1;
    private long failedExecutions = 0;

    /**
     * This method will be called by the {@link de.hybris.platform.jmx.mbeans.ProfilingReportBean#logExecutionTime}
     */
    @Override
    public void logExecutionTime(final long executionTime, final String exception)
    {
        totalCount++;

        totalExecutionTime += executionTime;
        if (executionTime > maxExecutionTime)
        {
            maxExecutionTime = executionTime;
        }
        else if (executionTime != 0 && (minExecutionTime == -1 || executionTime < minExecutionTime))
        {
            minExecutionTime = executionTime;
        }
    }

    @Override
    public void logException()
    {
        failedExecutions++;
    }

    /**
     * Method will be exported via JMX
     */
    * @see de.hybris.platform.jmx.mbeans.ProfilingReportBean#getFailedExecutions()
    */
    @Override
}

```

```

        public long getMaxExecutionTime()
        {
            return maxExecutionTime;
        }

        /**
         * Method will be exported via JMX
         *
         * @see de.hybris.platform.jmx.mbeans.ProfilingReportBean#getMaxExecutionTime()
         */
        @Override
        public long getMinExecutionTime()
        {
            return minExecutionTime;
        }

        /**
         * Method will be exported via JMX
         *
         * @see de.hybris.platform.jmx.mbeans.ProfilingReportBean#getMinExecutionTime()
         */
        @Override
        public long getTotalCount()
        {
            return totalCount;
        }

        /**
         * Method will be exported via JMX
         *
         * @see de.hybris.platform.jmx.mbeans.ProfilingReportBean#getTotalCount()
         */
        @Override
        public long getTotalTime()
        {
            return totalExecutionTime;
        }
    }
}

```

Listing the Profiled Methods

The default implementation of the profiler adds a new root node for each defined aspect to the domain node `hybris` and its tenant-related child node. Simply adjust the `domain` property as needed to define your own root node name.

For every aspect, a child-node with the name of the profiled class is added. For every one of these class nodes, there is a child-node with the name of the profiled methods.

Performance Tuning

You can exclude some performance issues early when creating the data model in a certain way, or during the early implementation phases. After the implementation phase, the SAP Commerce system infrastructure and environment including hardware, software, and configuration are typical aspects of performance planning and improvement.

Development Aspects

Some performance issues can be excluded early when creating the data model in a certain way or during the early implementation phases. This section addresses Consultants and Developers of SAP Commerce systems.

Data Model

Symptoms: Slow creation or modification of items - during synchronization or ImpEx runs, for example.

System performance adequate otherwise, for example during online shop operation.

Cause	Detection	Solution	See Also
Multiple database accesses during item creation	<ul style="list-style-type: none"> Create small ImpEx script to create two or three items of the ItemType you want to check. The items in the import must not exist in the Platform, otherwise you cannot test correctly: <ol style="list-style-type: none"> 1. Enable JDBC logging. 2. Clear JDBC log. 3. Import the small import file using <code>http://localhost:9001/monitor_database.jsp</code>. 4. Stop JDBC logging. 5. Open JDBC log in editor and search for an additional UPDATE statement after INSERT. 	Mark all attributes without custom logic as initial to avoid and additional update statement during item creation.	<ul style="list-style-type: none"> A C
Large number of ItemType/Relation instances in default tables	<ol style="list-style-type: none"> 1. Use your favorite database tool to determine the largest tables in your database (or use <code>http://localhost:9001/monitor_database_tablesizes.jsp</code> and select the Sort by Count button). 2. If you find GENERICITEMS or LINKS are in the TOP 5, determine which Itemtypes/Relations are missing a deployment definition. GENERICITEMS contains the ItemTypes with no individual deployment; LINKS contains the instances of RelationTypes with no individual deployment. 	Define deployments for your own ItemTypes/Relations and update/initialize the system afterwards.	

- [Specifying a Deployment for Platform Types](#)
- [Initializing and Updating SAP Commerce](#)

Slow statements when using relation getter/setter	Monitor your database using your favorite database tool or use JDBC logging including thread dump to log statement execution time. See also <i>Import of relation attributes</i> below.	<ul style="list-style-type: none"> Double check that you really do not need your relation to be sorted (ordered).
---------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------

- If not, mark source and target as unordered by adding `ordered="false"` to the `<sourceElement>` / `<targetElement>` tag.
- [Administration Console](#)
- [The Type System](#)

Missing index	Overall database performance is good, but individual statements have long execution time.
---------------	-------------------------------------------------------------------------------------------

1. Run JDBC logging.
2. Check for statements with an usually long execution time.
3. Check the WHERE part and the ORDER BY part of these statements whether the addressed tables and attributes have an index on them.
4. Create index by defining in the `items.xml` file.

- [items.xml : <index>](#) |

Import

Symptoms: Slow data import.

Cause	Detection	Solution	See Also
Unsorted items to be imported	All data related to a specific header can not be resolved at first pass.	Structure your header/data order, for example firstly import all items, afterwards all relations. For example a catalog version gets only imported at first pass if the related catalog is imported before. In addition, you can use multi-threading.	<ul style="list-style-type: none"> • ImpEx API, section <i>Import API</i>
Only one of multiple available CPU cores is used	Multi-threaded impex is not configured.	Use multi-threaded ImpEx import. Note that this requires structured import packages.	<ul style="list-style-type: none"> • ImpEx API, section <i>Import API</i>
Missing database indexes	Use JDBC logging to find long duration SELECT statements during import.	Make sure that all attributes marked as unique and all attributes used within item expressions are indexed. These attributes are used by SELECT statements.	<ul style="list-style-type: none"> • The Type System : RelationTypes • items.xml : <index>
Out of memory	The average amount of items imported per second gets lower and lower.	Your cache is configured too large. A good rule of thumb is to have a maximum of 40k cache entries per 500MB RAM.	<ul style="list-style-type: none"> • Configuring the Behavior of SAP Commerce
Import of relation attributes	Importing items with relation attributes has low performance.	If possible, try to import relations directly after the related items have been imported. The advantage is that the items are still kept in the SAP Commerce cache and therefore import is faster. Unfortunately, you cannot use this solution for directly importing the relation items, if you wish to frequently update the relations and your import file does not contain explicit deleting commands for those relations that needs to be deleted. This means, because your import files do not contain explicit deletes, this solution can only be used for initial imports.	<ul style="list-style-type: none"> • ImpEx API, section <i>Import API</i>

Synchronization

Symptoms: Slow catalog synchronization.

Cause	Detection	Solution
Due to dependencies between different synchronized ItemTypes, synchronization has to run in multiple passes to satisfy all dependencies.	Consult your application server log (console log) and check if multiple passes are performed to synchronize your catalog versions.	<p>Restructure your Root ItemTypes and adjust your attribute configuration to prevent unnecessary synchronization steps.</p> <p>For example, it is possible to import products without importing the categories, and afterwards import categories. It is also possible to import categories without importing products, and importing the products afterwards.</p> <p>However, if entry order matters, you need to import those items last which contain the other items' order. For example, if the order in which products are kept by a category matters, you need to import the products first and the categories afterwards.</p>

Avoiding Common Performance Bottlenecks

Lazy Loading

It can be judicious particularly in tight iteration loops to avoid loading complete models if all you require is some field/value within the model. See [Models](#).

Disabling Ordering for Ordered Relations

It is possible to disable the sorting of ordered relations which are defined by SAP Commerce . If your project does not rely on certain relations to be ordered, you can switch off ordering and therefore greatly improve import and synchronisation performance.

To disable ordering for the Category to Product relation, add the following property.

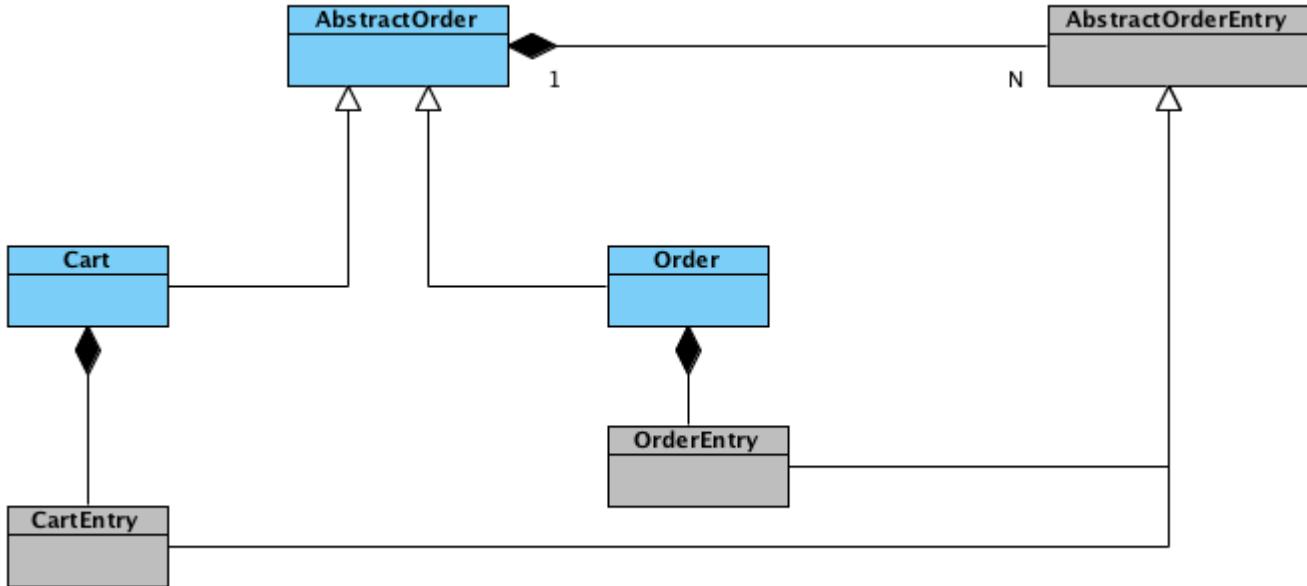
```
relation.CategoryProductRelation.source.ordered=false
relation.CategoryProductRelation.target.ordered=false
```

→ Tip

Ordering is disabled by default. One exception to this is **collectiontype** = "list", which is always ordered.

Boosting Performance by Redeclaring 1-n Relations

You can define a relation on an abstract level (between two abstract classes), and then re-declare this relation in concrete classes to point to concrete subclasses:



As you can see, there is a relation defined between **AbstractOrder** and **AbstractOrderEntry**. You can conveniently base your business code on these abstract classes, and reuse the code for concrete subclasses.

However, at runtime we deal with concrete relations:

- Order to OrderEntries
- Cart to CartEntries

The problem is that, when querying for entries of **Order**, **CartEntries** table are queried, too, even though we know that only **OrderEntries** should be taken into account. The same issue exists for **Cart**, namely that the **getEntries()** method would also look into the **OrderEntries** table, although we know that only **CartEntries** should be queried.

You can define a relation on an abstract level (between two abstract classes), and then re-declare this relation in concrete classes to point to concrete subclasses such that the definition of **Order** item contains only **OrderEntries** . For details, see [The Type System](#) .

This solution strikes a very good balance between flexibility and re-use on code level, without sacrificing performance at the ORM layer. It ensures:

- Flexibility on code-level: In java, you can use abstract base classes in your business logic, not relying on concrete implementations, so you can reuse the same logic for, say, **Order** and **Cart**
- Performance on persistence-level: When retrieving **Order** from the database, and then **OrderEntries** , SAP Commerce will query only the table containing **OrderEntries** . Other tables containing, for example, **CartEntries** etc. will not be included in the query.

Custom Ordering

We also introduced a custom property **ordering.attribute** defined for **AbstractOrder2AbstractOrderEntry** relation: .

By defining this property, it is possible to specify which attribute will be used to order the **many-side** items when retrieving from the database. In the example above, we defined the **many-side** as **ordered="false"** , and specified a custom ordering attribute.

SAP Commerce service layer takes care of putting **OrderEntries** in the correct order by setting the **entryNumber** , so there is no need for the ORM to add an additional ordering column, therefore the many side is **ordered="false"** .

However, entries retrieved from the database should be ordered according to the **entryNumber** computed upon save, and this can now be achieved using the **ordering.attribute** property.

Administration Aspects

The SAP Commerce system's infrastructure and environment including hardware, software, and configuration are typical administration aspects of performance planning and improvement. This section addresses Consultants and SAP Commerce Administrators.

Area/Cause	Detection	Solution
Machine		
Use of 32bit operating systems	Check system properties.	Use a 64bit system. We highly recommend to use a 64bit operating system to be able to address more than 4 GB of RAM.
Memory	Check system properties.	Increase physical machine RAM capacity to 8 GB or more. Hint: The following settings in bold should be adapted to the overall amount of memory in the system: Allow 2 GB for the operating system, so if you have 8 GB RAM installed, use -Xms6G -Xmx6G , if you have 16 GB RAM installed, use -Xms14G -Xmx14G .
Database		
MySQL Database is slow	System initialization takes very long (up to 30 min) when using MySQL.	Check settings for writing transactional logs in InnoDB.
Database configuration not optimal	<ul style="list-style-type: none"> • Database performance is not up to expectations. • Database proves to be bottleneck in system setup. 	Seek assistance through your support channels and benefit in building up expertise concerning databases.
Application Server		
Outdated version of the Java Virtual Machine.	Use java -version on your command line to check the version number.	Always use the latest version. We highly recommend the use of Java 6u12 or later.
Use of 32-bit Java	Use java -version on your command line to check whether you are using 64bit..	Times of 32-bit systems are over. We highly recommend using 64-bit architecture and a 64-bit Java VM to be able to use more than 2 GB memory for the java process.
Inadequate Java start parameters (including memory)	Various problems can arise by using wrong java parameters: <ul style="list-style-type: none"> • Long garbage collection pauses 	If using Sun Java 6 and using a 64bit system take the following settings as a starting point for your productive settings: -server -Xss256K -Xms6G

Area/Cause	Detection	Solution
	<ul style="list-style-type: none"> • High CPU load • OutOfMemoryExceptions 	<pre>-Xmx6G -XX:+UseConcMarkSweepGC -XX:NewRatio=1 -XX:PermSize=200M -XX:MaxPermSize=200M -XX:CMSInitiatingOccupancyFraction=85 -Xloggc:\${HYBRIS_LOG_DIR}/java_gc.log -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Dsun.rmi.dgc.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000</pre> <p>NOTE that these are just the memory/GC related settings you might/should need to add additional options.</p>

Configuration and Operation Aspects

Special operations and selected day-to-day tasks can require specific settings or procedures to avoid performance issues. This section addresses users of running systems typically using Backoffice for operation.

Cause	Detection	Solution	See Also
Generally	<p>How to identify performance issues:</p> <ol style="list-style-type: none"> 1. Check the CPU load of your application servers and database (DB) servers. 2. In case of high CPU load on DB Server use your database administration applications to determine: <ul style="list-style-type: none"> o Long running statements o Index usage o Number of statements per second 3. In case of high CPU load on application servers: <ul style="list-style-type: none"> o Get some thread dumps of the application server and analyze them. 	<p>Appserver : If the appserver tier seems to be the problem, you should use thread dumps / a profiling software to analyse, what is slowing down your system.</p> <p>Database : In order to find out, which queries might slow down your system, you should enable logging of slow statements on db server side.</p>	<ul style="list-style-type: none"> • Thread dump online Administration Console • MySQL - The Slow Query Log http://dev.mysql.com/doc/refman/5.1/en/slow-query-log.html ↗ :

Cause	Detection	Solution	See Also
	<ul style="list-style-type: none"> ○ Find out what is causing the load, for example high traffic on the shop, running import(exports, or product managers prepare the next summer catalog. 		
Database grows larger and larger	Lots of entries: <ul style="list-style-type: none"> • Cronjobs (joblogs table) • SavedValues (savedvalues table) • props table 	Clean up database.	<ul style="list-style-type: none"> • Item Attribute Modification History, section <i>Cleaning Up SavedValues</i>
Swapping	<ul style="list-style-type: none"> • Linux: Use top and vmstat. - Windows: Use the System Monitor to check the percentage of page file use. 	Add memory to the server or if not possible adjust your memory or cache settings to prevent your system from swapping.	<ul style="list-style-type: none"> • http://www.linuxcommand.org/man_pages/vmstat8.html ↗ : vmstat documentation • http://support.microsoft.com/kb/889654/en-us ↗

Tips on Performance Tuning

Several best practices for specific performance issues are provided in this topic. These are based on both internal testing and performance in real-world deployments.

Database

If the SAP Commerce cache is configured to hold a reasonable amount of entries there are no known issues regarding long-running queries. JDBC logs and profiler runs showed that most time was spent rendering the result page fetching their data from SAP Commerce cache. Exception from this is the creation of carts (see below) which can lead to problems.

- **Transaction awareness:**

If you are using MyISAM, note that MyISAM is not transaction aware. You should consider using InnoDB as Storage engine. But: This will lead to an estimated 20% loss of performance when doing inserts or update.

To use InnoDB, change your **local.properties**:

```
mysql.tabletype=InnoDB
```

Even if you lose some update performance we strongly recommend the use of InnoDB in productive environments to be transaction aware.

i Note

If you are using InnoDB there is a MASSIVE performance impact if using the default settings especially for `innodb_flush_log_at_trx_commit`. You have to set this to value 0 (write to log and flush every second). You can change this from the MySQL Administrator or directly inside your mysql configuration file `my.ini`.

Please be aware not to set `AUTOCOMMIT=0` easily as not everything sent to the DB is running within transactions.

- **Tuning database settings:**

- Make sure the query cache is turned **OFF**. This is important, because the SAP Commerce application has its own query cache which is faster, because it is inside the VM. If you have activated the mysql query cache in addition to the SAP Commerce cache, this will lead to more memory consumption and a performance overhead of approx 5%.

hasSessionCart vs. getSessionCart

Performance impact: **HIGH**

Make sure your system does not create a new cart when the user does not need it. This has massive impact to the system performance since for every session a new entry in the Carts table is created. In SAP Commerce, if using `cartService.getSessionCart()`, a new cart object is created implicitly if none is yet existing for this session. You should always use `cartService.hasSessionCart()` before.

You can simply test this behavior by executing

```
select count(*) from {Cart}
```

from the admin flexiblesearch test page before and after entering the store with a newly created session.

See an example of a correct use of `cartService.getSessionCart()` and `cartService.hasSessionCart()`:

```
if ((!cartService.hasSessionCart() ||
    cartService.getSessionCart().getEntries().isEmpty()) ...
```

Improved Session Handling

If you're experiencing problems with a large number of sessions, it is possible that JaloSessions are not destroyed when httpSessions expire. To fix this, you can use the property `improvedsessionhandling`.

When the property `improvedsessionhandling` is set to `true`:

1. JaloSession is not added to cache on creation. Extensions that use `JaloConnection#getSession(String)` should manually add the session to the cache after it is created using `@link JaloConnection#checkAndAddToSessionCache(JaloSession)`.
2. Method `@link JaloConnection#getSession(String)` will only return sessions which have been registered manually.
3. Http Session Listener is used to close the JaloSession when an HttpSession expires.

Use of Expensive Methods Like getAllProducts

Performance impact: can be **HIGH**

The `getAllProducts()` method is deprecated. Instead, use the FlexibleSearch query:

```
SELECT {PK} FROM {Product}
```

For more information, see [FlexibleSearch](#).

Use of Own Object Cache Frameworks

Performance impact: **LOW to MEDIUM**

In our experience the use of own object cache framework in addition to the SAP Commerce cache, except the caches that work on complete pages like Opensymphony's OSCache, can not be recommended. It turned out that it always only fixed symptoms and not the main causes of weak performance.

FlexibleSearch Query Results Minimum Time to Live

Performance impact: **LOW to HIGH**, depending on the number of item updates by runtime

SAP Commerce enables defining a minimum time to live (TTL) for results of FlexibleSearch queries. Specifying a TTL reduces load on the database.

SAP Commerce Cluster Performance Tweaks

Datagram Size (Microsoft Windows)

Microsoft Windows supports a fast I/O path which is utilized when sending "small" datagrams. The default setting for what is considered a small datagram is 1024 bytes; increasing this value to match your network MTU (normally 1500) can significantly improve network performance. To adjust this parameter:

1. Run Registry Editor (regedit)
2. Locate the following registry key `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\AFD\Parameters`
3. Add the following new DWORD value
 - o Name: `FastSendDatagramThreshold`
 - o Value: 1500 (decimal)
4. Reboot

Thread Scheduling (Microsoft Windows)

Windows (including NT, 2000 and XP) is optimized for desktop application usage. If you run two console ("DOS box") windows, the one that has the focus can use almost 100% of the CPU, even if other processes have high-priority threads in a running state. To correct this imbalance, you must configure the Windows thread scheduling to less-heavily favor foreground applications.

1. Open the Control Panel.
2. Open System.
3. Select the Advanced tab.
4. Under Performance select Settings.
5. Select the Advanced tab.
6. Under Processor scheduling, choose Background services.

Platform Filters

Platform Filters is a solution that provides a set of filters used for Web applications. These filters are represented by a filter chain which can be modified according to your needs. You may also implement your own filter and include it in a custom filter chain.

Every Web application requires Platform Filters to ensure that Platform works properly. All available filters are a part of Platform, but not all of them are necessary for every web application.

Catalog Version Filters are necessary for proper handling of catalog versions in a current session:

- **SimpleCatalogVersionActivationFilter**: It assures that configured names of catalog versions are set as session catalog versions. Mentioned configuration is done through Spring property `activeCatalogVersions`. For more information and an example of bean with `activeCatalogVersions` property, see section about configuring existing filters in [Filters for Web Applications](#) in Implementation.
- **DynamicCatalogVersionActivationFilter**: It takes care for activating the catalog versions at runtime. It allows to activate them also with proper URL parameter.

For more information, see [Catalogs and Catalog Versions](#).

Filters with different use:

- **SessionFilter**: It takes care for session handling to detect and/or create the SAP Commerce session based on a HTTP session.
- **ProfileFilter**: It measures the time it took to process the complete request.
- **Log4jFilter**: It makes sure to store the remote IP address before the request gets processed and removes it afterwards.
- **RedirectWhenSystemIsNotInitializedFilter**: It takes care for redirection of the current request in case the system is recognized as not initialized.
- **DataSourceSwitchingFilter**: It takes care for switching the data sources for given tenant.
- **SecureMediaFilter**: It handles securing access to medias. For more details read [Secure Media Access](#)
- **WebAppMediaFilter**: It handles both secure and non-secure types of media. For more details, see [WebAppMediaFilter](#)

Filters for Web Applications

Every Web application requires Platform Filters to ensure that Platform works properly. All available filters are a part of Platform, but not all of them are necessary for every web application.

Using Filters

The `PlatformFilterChain` is triggered on every HTTP request. The default Spring bean of the `PlatformFilterChain` is registered in the Spring application context in the `filter-spring.xml` file. It instantiates filters that are necessary for every Web application.

The `core-filter-spring.xml` is defined as follows:

`core-filter-spring.xml`

```
<!-- deprecated -->
<bean id="dynamicTenantActivationFilter" class="de.hybris.platform.servicelayer.web.DynamicTenantActivationFilter">
<!-- deprecated -->
<bean id="tenantActivationFilter" class="de.hybris.platform.servicelayer.web.TenantActivationFilter">
```

```

<constructor-arg value="${tenantId}"/>
</bean>

<bean id="sessionFilter" class="de.hybris.platform.servicelayer.web.SessionFilter" >
  <property name="sessionService" ref="sessionService"/>
</bean>

<bean id="log4jFilter" class="de.hybris.platform.servicelayer.web.Log4JFilter"/>

<bean id="defaultPlatformFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterCha:
  <constructor-arg>
    <list>
      <ref bean="log4jFilter"/>
      <ref bean="sessionFilter"/>
    </list>
  </constructor-arg>
</bean>

```

However, every Web application can instantiate other filters in their own Spring application context file. In other words, any filter can be easily activated or deactivated in the Spring application context for every single Web application.

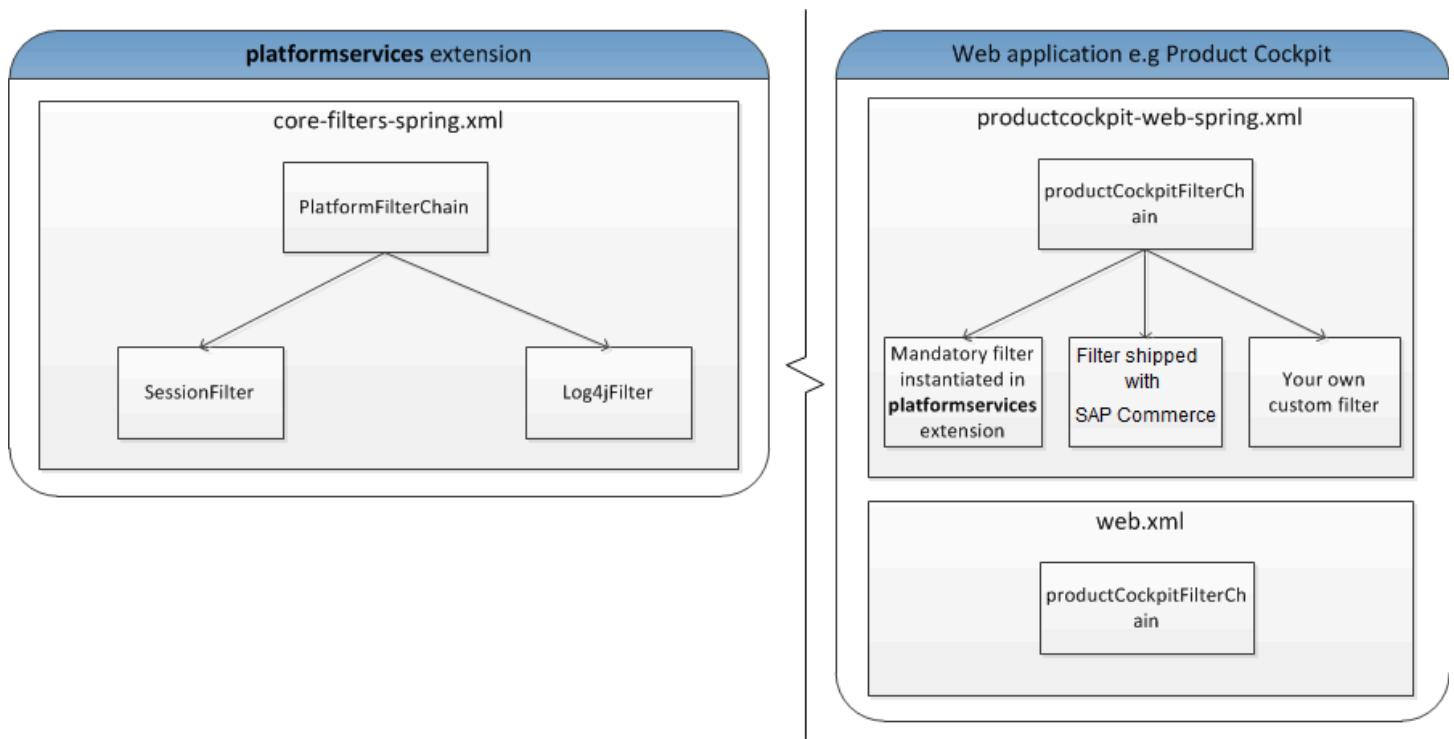


Figure: This diagram shows the default Platform filter chain registered in **platformservices** extension and a custom Platform filter chain registered in any Web application.

Configuring Existing Filters

Every extension that has a **web** extension module needs to have its own definition of a custom filter chain based on the **platformFilterChain** Spring bean. Typically it is a mix of the filters provided by the global Spring application context and Spring application context of this extension. In almost every case only two files need to be modified. It is the **web.xml** and the **<extensionname>-web-spring.xml** of the extension. If you register a custom filter chain or a custom instance of existing or new filter in the Spring application context of specific extension, then for better readability it is recommended to add **<extensionname>** prefix at the beginning of each instance.

To configure your own filter chain definition for your custom **mystore** extension, follow below instruction:

1. Create the filter chain in a Spring application context of your extension.

mystore-web-spring.xml

```

<bean id="mystoreFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterChain"
    <constructor-arg>
        <list>
            <ref bean="mystoreProfileFilter"/>
            <ref bean="log4jFilter"/>
            <ref bean="mystoreRedirectFilter"/>
            <ref bean="sessionFilter"/>
            <ref bean="mystoreDataSourceSwitchingFilter"/>
            <ref bean="mystoreCatalogVersionActivationFilter"/>
        </list>
    </constructor-arg>
</bean>

<bean id="mystoreCatalogVersionActivationFilter"
    class="de.hybris.platform.servicelayer.web.SimpleCatalogVersionActivationFilter"    >
    <property name="catalogVersionService" ref="catalogVersionService"/>
    <property name="activeCatalogVersions" value="hwcatalog:Online,SampleClassification:1.0"/>
</bean>

<bean id="mystoreRedirectFilter"
    class="de.hybris.platform.servicelayer.web.RedirectWhenSystemIsNotInitializedFilter">
    <constructor-arg>
        <value><!-- nothing - redirect to default maintenance page --></value>
    </constructor-arg>
    <constructor-arg>
        <list>
            <value>login</value>
            <value>static</value>
        </list>
    </constructor-arg>
</bean>

<bean id="mystoreDataSourceSwitchingFilter" class="de.hybris.platform.servicelayer.web.DataSourceSwitchingFilter"/>
</bean>

<bean id="mystoreProfileFilter" class="de.hybris.platform.servicelayer.web.ProfileFilter"/>
</bean>

```

2. Activate the filter chain in the web.xml of the extensions.

If you already have the `HybrisInitFilter` or `RootRequestFilter` filter already in place, then just replace it.

`web.xml`

```

<filter>
    <filter-name>mystoreFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

```

3. Set the mapping correctly in the web.xml file.

`web.xml`

```

<filter-mapping>
    <filter-name>mystoreFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```

i Note

Async Support in Tomcat 7

If you use Servlet 3.0, you need to add `<async-supported>true</async-supported>` to every filter and servlet that you use together with the ZK framework. Otherwise you might get errors like

Request processing failed; nested exception is `java.lang.IllegalStateException: Async support must be enabled on a servlet and for all filters involved in async request processing. This is done in Java code using the Servlet API or by adding "<async-supported>true</async-supported>" to servlet and filter declarations in web.xml`

Creating a New Filter

You can create your own filter and add it to the filter chain of your Web application. Make sure that when creating a custom filter you need to implement the `doFilter()` method.

The following example presents how to do it properly:

1. Create new class.

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

    @Override
    public void doFilter(final ServletRequest request, final ServletResponse response, final FilterChain filterChain)
        throws IOException, ServletException
    {
        LOG.info("Pre Request actions...");
        filterChain.doFilter(request, response);
        LOG.info("Post request actions...");
    }
}
```

It is important to call the `filterChain.doFilter()` method, as without it the filter chain is going to break. A good practice in some cases could be wrapping the `filterChain.doFilter()` call in `try` block and put some required actions also in `finally` block:

- Everything you put before the `filterChain.doFilter()` method is executed before the request is finished.
- Everything after that method is executed after the request is finished. It is useful for deactivating tenants, session, and other.

2. Now register the bean for above filter and inject it in the filter chain in the Spring application context of your extension:

`myWebApp-web-spring.xml`

```
<bean id="myWebAppFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterChain"
    <constructor-arg>
        <list>
            <ref bean="log4jFilter"/>
            <ref bean="sessionFilter"/>
            <ref bean="myWebAppMyFilter"/>
        </list>
    </constructor-arg>
</bean>

<bean id="myWebAppMyFilter" class="de.hybris.platform.myWebApp.MyFilter">
</bean>
```

Polyglot Persistence

The polyglot persistence provides a way to store specified data types in an alternative storage. It supports all CRUD operations as well as searching for objects. Learn how to configure this feature and how to implement your own alternative storage.

Polyglot persistence supports the **create**, **read**, **update**, and **delete** operations as well as searching for objects. The **create**, **update**, and **delete** operations are transparent and available through service layer. The **read** operation offers two options. The first option, which is transparent to the user, is reading an object through service layer. The second option is realized by the **polyglot persistence querying language**. The polyglot language is added to Flexible Search because searching for data in the main relational database is done by queries similar to SQL queries. The polyglot language is designed for document-based storage that cannot support the SQL query language.

For more information about the language, see [Polyglot Persistence Query Language](#).

You can use polyglot persistence for both, existing, and new item types. The Platform module provides a generic way to handle the final storage. It means that you must deliver your alternative storage externally and it can't be a part of the Platform module.

Benefits

Polyglot persistence may be helpful in these situations:

- you want to relieve the load of your database before it hits performance limit; you may improve performance of the whole system by storing some data types in an alternative storage
- you want to provide a non-SQL storage for some data
- you want to optimize the data structure around one root element; the **documentcart** extension is an example implementation of storing cart elements with its all related types (such as **cartEntry** or **productInfo**) as one composed structure (a document).

Limitations

Polyglot persistence has limitations that you must consider before using it.

Polyglot persistence doesn't support these functionalities:

- data migration between the existing database and the final storage
- impex import (it requires unsupported FlexibleSearch features such as unions and joins).
- y2ysync framework
- data report and audit
- SAP Commerce Administration Console doesn't accept inline parameters for polyglot queries (but Admin APIs fully support querying)
- search restrictions

The polyglot querying language capabilities are limited to minimum. The supported operations include:

- querying for model or item instances
- filtering (the **WHERE** clause) based on attributes (localized and unlocalized); supported operators include: **=**, **<>**, **<**, **<=**, **>**, **>=**, **IS NULL**, **IS NOT NULL**, parenthesis, Boolean operators: **AND**, **OR**
- sorting instances (the **ORDER BY** clause) based on attribute values

The operations that aren't supported are:

- JOINS
- SUBQUERIES
- GROUP BY
- CASE
- Functions

- Operators: LIKE, IN

When you switch an existing data type into a polyglot type, you must change all existing flexible search queries related to that type into polyglot. SAP Commerce comes with many built-in types and the logic around it. This logic uses FlexibleSearch queries heavily. If you want to switch a particular type into polyglot, you must rewrite all FlexibleSearch queries related to that type into polyglot versions. This point is also valid if you already use Platform and have added your own types and FlexibleSearch queries.

Related Information

[Polyglot Persistence Query Language](#)

[ydocumentcart Extension Template](#)

Type Configuration

To use a specific type in polyglot persistence, you must configure this type so that polyglot persistence supports it.

A polyglot persistence item type configuration is based on properties files, `project.properties` or `local.properties`.

Here is an example:

```
polyglot.repository.config.mypolyglot.beanName=myPolyglotRepository
polyglot.repository.config.mypolyglot.typeCodes=E,G[x=E]
```

Where:

- `polyglot.repository.config` is a prefix for a polyglot configuration; you can't change this prefix
- `mypolyglot` is a name of the instance; polyglot supports many repositories at once
- `myPolyglotRepository` is a name of the bean implementing the `de.hybris.platform.persistence.polyglot.ItemStateRepository` interface
- E, and G are type names from the type system; these names must match the `code` attributes from your `*-items.xml` file.
- `[. . .]` is where conditions are defined
- in `[x=E]`, x is a qualifier attribute from the relation defined in `*-items.xml` that points to a related item; the name must match the `qualifier` attribute from your `*-items.xml` file
- in `[x=E]`, E is a target type name (a `typeCode` of a target item); it allows the system to narrow down possible target item types; the name must match the `code` attribute from your `*-items.xml` file.

The prerequisites for item types are:

- a type must have its own representation in an `*-items.xml` file
- fully supported types must have their own deployment

You can configure item types as:

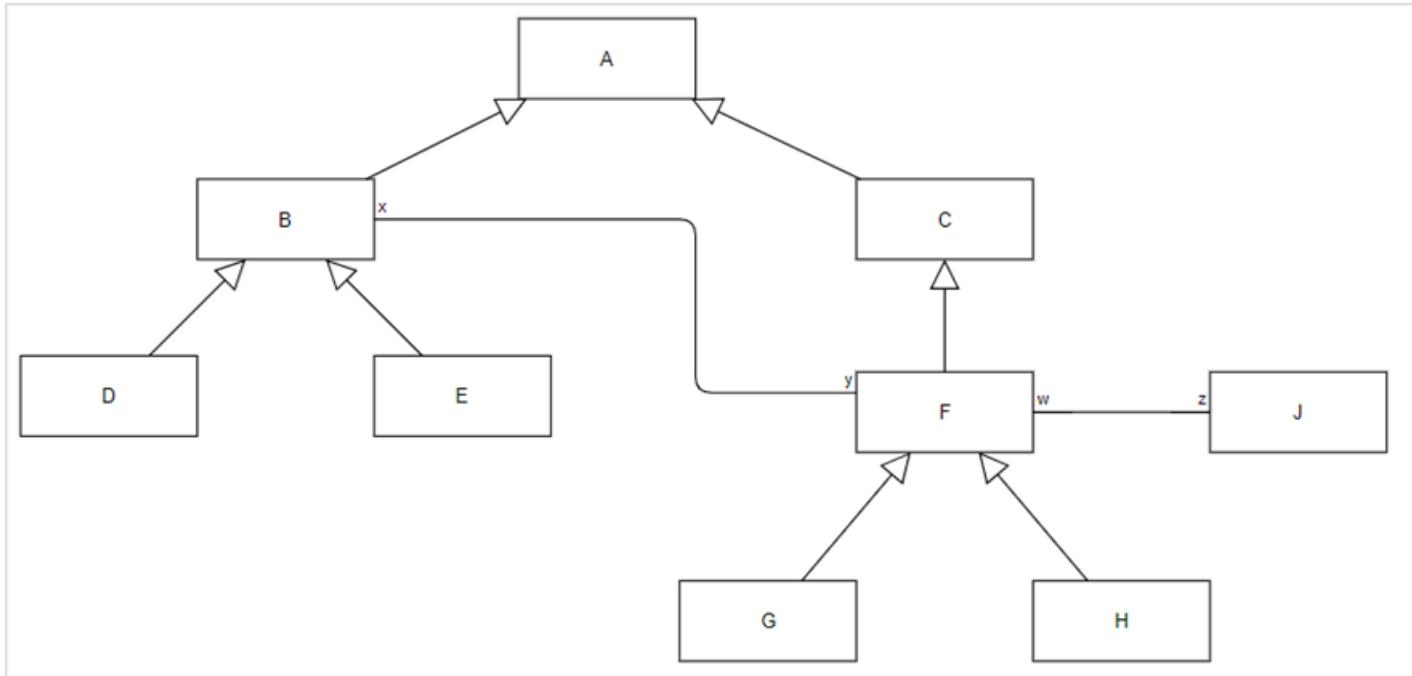
- FULL: All items of this type are stored in a particular polyglot storage. To configure a type as fully supported, add it to the configuration without any conditions.
- PARTIAL: Only items that meet given criteria are stored in a particular polyglot storage. This kind of configuration provides a way to store 1 item type in different polyglot storages and in the main database. To configure a type as partially supported, add it to a configuration with conditions in brackets []:
 - `G[x]` means that G item is stored in polyglot only when its x related item is stored in polyglot.
 - `G[x=E]` means that G item is stored in polyglot only when its x related item is stored in polyglot and x related item is of E type.

Configure at least 1 type as fully supported. All partial types must somehow relate to the item type that is supported fully. All polyglot child items are configured as polyglot too, with the same conditions as their parents. All parent items are treated by polyglot as partially supported.

Examples

All the examples below use the type hierarchy from the diagram.

A, B, C, D, E, F, G, H, J are item type codes, and x, y, w, z are relation qualifiers.



Example 1

```

polyglot.repository.config.mypolyglot.beanName=myPolyglotRepository
polyglot.repository.config.mypolyglot.typeCodes=D,E
  
```

Types D and E are configured as fully supported with myPolyglotRepository.

Types B and A are configured as partially supported with myPolyglotRepository. It is because an instance of the A type may be of the D type as well.

Example 2

```

polyglot.repository.config.mypolyglot.beanName=myPolyglotRepository
polyglot.repository.config.mypolyglot.typeCodes=B
  
```

Type B is fully supported with myPolyglotRepository - types D and E are configured as polyglot automatically.

Type A is configured automatically as partially supported with myPolyglotRepository.

Example 3

```

polyglot.repository.config.mypolyglot1.beanName=myPolyglotRepository1
polyglot.repository.config.mypolyglot1.typeCodes=D
polyglot.repository.config.mypolyglot2.beanName=myPolyglotRepository2
polyglot.repository.config.mypolyglot2.typeCodes=E
  
```

Type D is fully supported with myPolyglotRepository1.

Type E is fully supported with myPolyglotRepository2.

Type A and B are partially supported with myPolyglotRepository1 and myPolyglotRepository2.

Example 4

```
polyglot.repository.config.mypolyglot.beanName=myPolyglotRepository
polyglot.repository.config.mypolyglot.typeCodes=E,F[x]
```

Type E is fully supported with myPolyglotRepository.

Types F, G, and H are partially supported by myPolyglotRepository. The condition is the x relation. It means that item types F, G, and H are stored in myPolyglotRepository only when its related E item is stored in myPolyglotRepository.

The A, B, and C types are partially supported by myPolyglotRepository.

To see it more clearly, follow the usage example:

```
D d = new D();
G g = new G();
g.x = d;
saveAll(); //g is stored in the main database (not polyglot) because its related item doesn't support x relation

E e = new E();
G g = new G();
g.x = e;
saveAll(); //g is stored in polyglot because its related item is stored in polyglot too
```

Example 5

```
polyglot.repository.config.mypolyglot.beanName=myPolyglotRepository
polyglot.repository.config.mypolyglot.typeCodes=E,G[x=E]
```

Type E is fully supported with myPolyglotRepository.

Type G is supported partially with myPolyglotRepository. The condition is the x relation. The G item is stored in myPolyglotRepository only when its x related item is stored in myPolyglotRepository and the x related item is of the E type.

Example 6

```
polyglot.repository.config.mypolyglot.beanName=myPolyglotRepository
polyglot.repository.config.mypolyglot.typeCodes=E,F[x],J[w]
```

Type E is fully supported with myPolyglotRepository.

Type F is supported partially with myPolyglotRepository. The condition is the x relation. The F item is stored in myPolyglotRepository only when its x related item is stored in myPolyglotRepository and the x related item is of the E type.

Type J is supported partially with myPolyglotRepository. In the diagram, you can see that there is a chain relation to the fully supported E type. The J item is stored in myPolyglotRepository when its w related item is stored in myPolyglotRepository (and the x related item is of the E type).

One-To-Many Part-Of Relations

It's important to implement one-to-many part-of relations carefully so that they can be handled properly by business logic. For example, in the `AbstractOrder2TestItemA` relation:

```
<relation code="AbstractOrder2TestItemA" localized="false">
<sourceElement qualifier="abstractOrder" type="AbstractOrder" cardinality="one">
</sourceElement>
<targetElement qualifier="testItemA" type="TestItemA" cardinality="many">
<modifiers partof="true"/>
</targetElement>
</relation>
```

With the following storage configuration:

```
polyglot.repository.config.ydocumentcart.typeCodes=Cart, TestItemA[abstractOrder]
```

Set your relations using many-side setters. For `AbstractOrder2TestItemA`, use the following setter:

```
testItemA.setAbstractOrder(cart);
```

In one-to-many relations, the system stores references on the “many” side. As a result for `AbstractOrder2TestItemA`, `TestItemA` has reference to `Cart`. In JALO or ServiceLayer, however, one-to-many relations can be set through either one-side or many-side setters.

The system saves items in a number of phases. As a result, when `Cart` is saved and gets its own PK, `TestItemA` is also saved, but its reference to `Cart` is null. The 2nd phase updates `TestItemA` with reference to `Cart` through an update statement in a relational database. In Jalo or ServiceLayer, the process is transparent.

However, when using polyglot persistence, it's important to ensure that a reference to `Cart` is available at the first step of the saving process when it's being decided if a given item can be stored in polyglot persistence. An instance of `TestItemA` can only be saved and stored when it has a reference to `Cart`. Using one-side setters ensures that an item of this type references `Cart` from the beginning of the saving process and is saved in the polyglot storage.

When using the `AbstractOrder2TestItemA` part-of relation, define it as required by using the `optional="false"` attribute to ensure that it works correctly in polyglot persistence:

```
<relation code="AbstractOrder2TestItemA" localized="false">
<sourceElement qualifier="abstractOrder" type="AbstractOrder" cardinality="one">
<modifiers optional="false" />
</sourceElement>
<targetElement qualifier="testItemA" type="TestItemA" cardinality="many">
<modifiers partof="true"/>
</targetElement>
</relation>
```

As a result, you need to call `testItemA.setAbstractOrder(cart)` explicitly to ensure correct behavior.

Caution

As items cannot be moved between persistencies, set all our references properly so that it is clear when a given item can be stored.

Creating Items Without Parent Attribute

The following property prevents the system from creating items without a parent attribute that are configured to be stored in the polyglot persistence:

```
polyglot.allow.creation.without.parent=false
```

The default value of the property is **false**.

Moving Entities Between Documents Securely

Moving polyglot entities from one document to another in repositories that don't support such operations by changing their parent attribute could result in data corruption. The following property allows you to define what happens when changing the parent of polyglot entities isn't supported:

```
polyglot.allow.changing.entity.parent=false
```

The property can have the following values:

- **true** - the system shows an error in the logs, but the operation is carried out
- **false** - the system throws an exception

The default value of the property is **true**.

Unit of Work

With the unit of work of the polyglot persistence feature, you can cache all modifications performed on a single item and flush them when a main operation ends. The unit of work also manages the item version that is used for optimistic locking.

Enabling Unit of Work

The feature is optional and disabled. To enable it, decorate your `ItemStateRepository` spring bean with `UnitOfWorkAwareItemStateRepository`, and replace the `beanName` in your `project/local.properties` file. Here is an example configuration:

`*-spring.xml`

```
<bean id="unitOfWorkAwareItemStateRepository" class="de.hybris.platform.persistence.polyglot.uow.UnitOfWorkAwareItemStateRepository">
    <constructor-arg ref="myItemStateRepository"/> <!-- Your ItemStateRepository -->
    <constructor-arg ref="unitOfWorkProvider" /> <!-- Unit of work provider, delivered with the Platform -->
</bean>
```

`project/local.properties`

```
polyglot.repository.config.yourPolyglotName.beanName=unitOfWorkAwareItemStateRepository
polyglot.repository.config.yourPolyglotName.typecodes=TypeA,TypeB,TypeC //types configured as polymorphic
```

Details

Unit of work caching is based on a single thread. Model service triggers an operation when saving an item. Each operation creates its own Scope and that scope lives until all modifications on a given item end. As an example, consider having an X type with three attributes (A, B, and C), and executing this code:

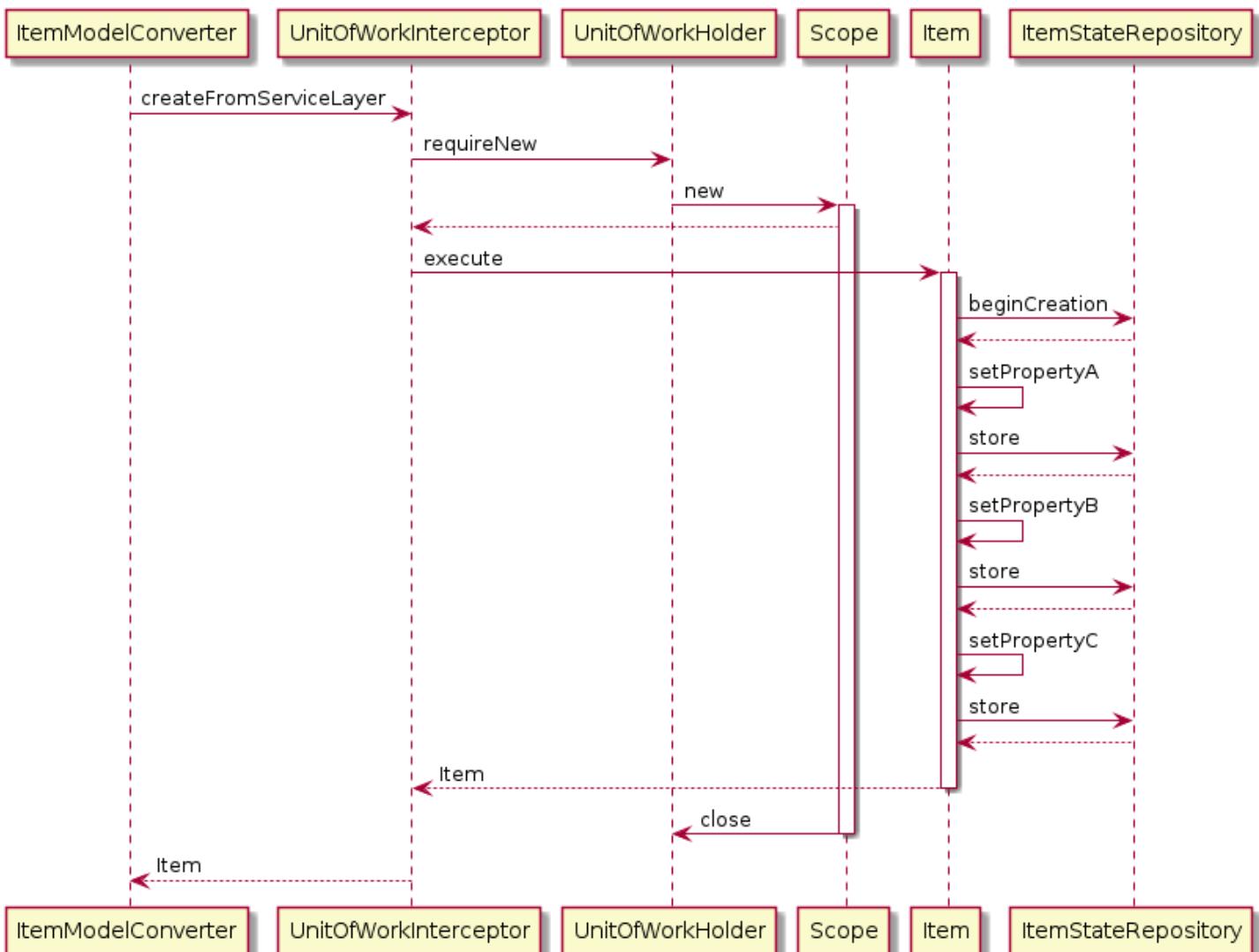
```
X x = modelService.create(X.class);
x.setA("aa");
x.setB("bb");
x.setC("cc");
modelService.save(x); //modelService.save triggers a main operation
```

As a result:

- with unit of work disabled, your repository gets hit four times (by methods `beginCreation` once, and `store` three times), and the final item version is set to 4
- with unit of work enabled, your repository gets hit only once, and the final item version is set to 1

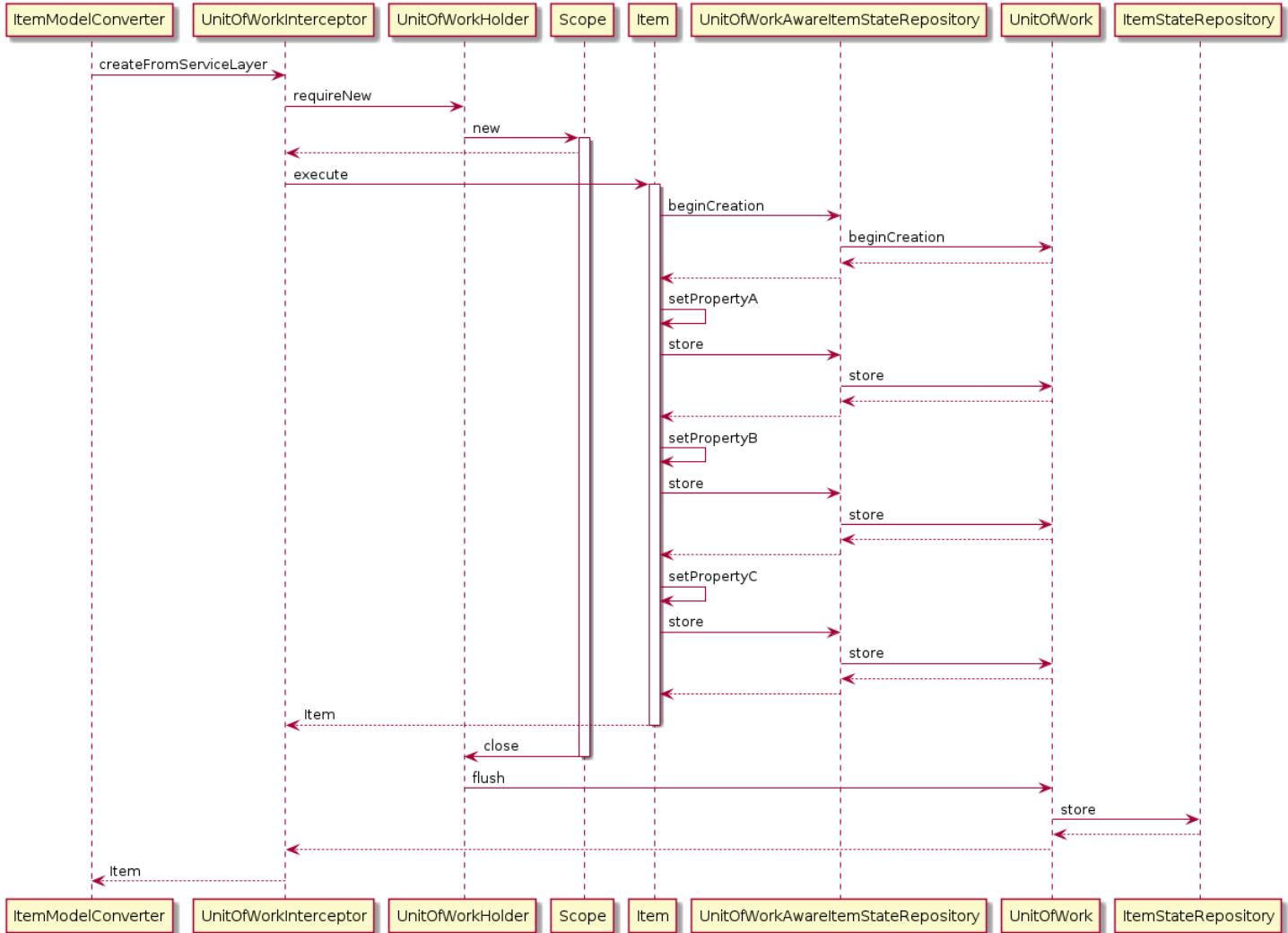
Unit of Work Disabled

The diagram is simplified to improve readability.



Unit of Work Enabled

The diagram is simplified to improve readability.



ItemStateRepository API

Polyglot persistence exposes an API to allow implementation of repositories.

The repositories you implement provide storage for types that you want to be supported by polyglot persistence. To allow a given repository to support a given type, provide a type configuration that binds the type with the repository. For more information about a type configuration, see [Type Configuration](#).

To provide a repository, implement the `ItemStateRepository` interface. Since `ItemStateRepository` is an interface, you choose the type of physical storage your data is stored in. It can be, for example, a relational database, a NoSQL database, or an in-memory data grid.

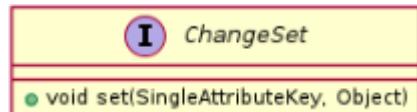
An example implementation is available in the `documentcart` extension (see [ydocumentcart Extension Template](#)). To learn how to implement an in-memory repository, see [Implementing ItemStateRepository](#).

The `ItemStateRepository` interface looks as follows:



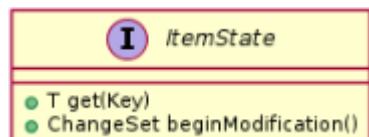
- The `beginCreation()` method creates the initial state (as an implementation of `ChangeState`) based on the item being created.
- The `store()` method saves or updates an item in a repository. It does so by using `ChangeState`, which describes either the initial state of an item or the change that should be applied to an item state.
- The `get()` method retrieves the item state instance matching a provided `Identity` from the repository.
- The `find()` method performs search based on provided `Criteria`.
- The `isSupported()` method is a generic mechanism that allows you to check whether a given implementation of the `ItemStateRepository` repository supports a given feature.
- The `remove()` method removes the item state from the repository.

The `ChangeSet` interface looks as follows:



`ChangeSet` is a contract for an implementation to be able to set a given value for a given key.

The `ItemState` interface looks as follows:



`ItemState` is a contract for an implementation to be able to get() a value representing a given key.

The instance of an implementation of an item state is later stored, updated, or retrieved by the implementation of the `ItemStateRepository` interface.

Implementing ItemStateRepository

Implement `ItemStateRepository` to provide a repository for desired item types.

Context

Use `ItemStateRepository` to provide a repository, for example, for the `Title` item type.

i Note

For simplicity, this example implementation doesn't cover, among other things, optimistic locking checks, exceptions handling, or transactions.

To provide a configuration for the repository to support the `Title` type, see [Type Configuration](#).

Procedure

1. Provide a name and an underlaying storage for your `ItemStateRepository` implementation.

This implementation uses an in-memory map store data in:

```

public class InMemoryTitleRepository implements ItemStateRepository
{
    private final ConcurrentHashMap<Identity, SimpleState> inMemory = new ConcurrentHashMap<>

    ...
}

```

2. Provide an implementation for `ItemState` and its methods.

`ItemState` is a wrapper around the map that stores key-attribute pairs. With the `beginModification()` method, it provides a modification change set used by the polyglot framework to provide a change to be stored. With the `applyChanges()` method, you can apply the change set when storing changes (see the implementation of the update part of the `store()` method).

```

package de.hybris.platform.persistence.polyglot;

import java.util.HashMap;
import java.util.Map;

public class SimpleState implements ItemState{

    private final Map<SingleAttributeKey, Object> values;

    public SimpleState(final Map<SingleAttributeKey, Object> values)
    {
        this.values = values;
    }

    @Override
    public <T> T get(Key key) {
        return (T) values.get(key);
    }

    @Override
    public ChangeSet beginModification()
    {
        return new InMemoryTitleRepository.ModificationChangeSet(this);
    }

    SimpleState applyChanges(final InMemoryTitleRepository.ModificationChangeSet modification
    {
        final HashMap newValues = new HashMap<>(values);
        newValues.putAll(modification.valuesToModify);
        newValues.put(WellKnownKey.VERSION.key, modification.baselineVersion + 1);

        return new SimpleState(newValues);
    }
}

```

3. Implement the `beginCreation()` method.

The `beginCreation()` method provides a new instance of the implementation of `ChangeSet` that can hold attribute values required for creation of an item.

```

public class InMemoryTitleRepository implements ItemStateRepository
{
    private final ConcurrentHashMap<Identity, SimpleState> inMemory = new ConcurrentHashMap<>

    @Override
    public ChangeSet beginCreation(final Identity id)
    {
        return new CreationChangeSet(id);
    }

    static class CreationChangeSet implements ChangeSet
    {
        final Identity id;
        final Map<SingleAttributeKey, Object> initialValues = new HashMap<>();

```

```

public CreationChangeSet(final Identity id)
{
    this.id = id;
    initialValues.put(WellKnownKey.ITEM_PK.key, id);
    initialValues.put(WellKnownKey.VERSION.key, Long.valueOf(0));
}

@Override
public void set(final SingleAttributeKey key, final Object value)
{
    initialValues.put(key, value);
}

...
}

```

4. Implement the `store()` method so you can persist something in your storage.

You either put your `CreationChangeSet` change set in memory (creation of an item) or you find a corresponding item for a given `ModificationChangeSet` change set and apply changes on it so that you can store the updated item.

```

public class InMemoryTitleRepository implements ItemStateRepository
{
    ...

    @Override
    public void store(final ChangeSet changeSet)
    {

        if (changeSet instanceof CreationChangeSet)
        {
            final CreationChangeSet creation = (CreationChangeSet) changeSet;
            inMemory.putIfAbsent(creation.id, new InMemoryImmutableState(creation.initialValue));
            return;
        }

        if (changeSet instanceof ModificationChangeSet)
        {
            final ModificationChangeSet modification = (ModificationChangeSet) changeSet;
            final SimpleState currentState = inMemory.get(modification.id);
            final SimpleState newState = currentState.applyChanges(modification);
            inMemory.replace(modification.id, newState);
            return;
        }
    }

    static class ModificationChangeSet implements ChangeSet
    {
        final Identity id;
        final long baselineVersion;
        final Map<SingleAttributeKey, Object> valuesToModify = new HashMap<>();

        public ModificationChangeSet(final ItemState itemState)
        {
            id = itemState.get(WellKnownKey.ITEM_PK.key);
            baselineVersion = itemState.get(WellKnownKey.VERSION.key);
        }

        @Override
        public void set(final SingleAttributeKey key, final Object value)
        {
            valuesToModify.put(key, value);
        }
    }

    ...
}

```

5. Implement the get() and remove() methods.

These methods get/remove proper item state instance from the internal storage.

```
public class InMemoryTitleRepository implements ItemStateRepository
{
    ...
    @Override
    public ItemState get(final Identity id)
    {
        return inMemory.get(id);
    }
    @Override
    public void remove(final ItemState state)
    {
        final Identity id = state.get(WellKnownKey.ITEM_PK.key);
        inMemory.remove(id);
    }
    ...
}
```

6. Implement the find() method.

Using available helper classes, the method builds a predicate based on provided criteria and uses the predicate to filter the map. Having a map filtered as a stream, it uses another helper class to build a find result required by the polyglot framework.

```
public class InMemoryTitleRepository implements ItemStateRepository
{
    ...
    private final Function<Criteria, MatchingPredicateBuilder> predicateBuilderProvider = Mat
    @Override
    public FindResult find(final Criteria criteria)
    {
        final Predicate<ItemState> matchingPredicate = buildMatchingPredicate(criteria);
        final Comparator<ItemState> cmp = ItemStateComparatorCreator.getItemStateComparator(c
        final Stream<ItemState> stream = inMemory.values().stream().map(ItemState.class::cast
            .sorted(cmp));

        return StandardFindResult.buildFromStream(stream).withCriteria(criteria).build();
    }

    private Predicate<ItemState> buildMatchingPredicate(final Criteria criteria)
    {
        final MatchingPredicateBuilder builder = predicateBuilderProvider.apply(criteria);
        return builder.getPredicate();
    }
}
```

Results

You have implemented `ItemStateRepository` and provided a custom storage for your data.

Example

```
package de.hybris.platform.persistence.polyglot;

import java.util.Comparator;
import java.util.HashMap;
import java.util.Map;
import java.util.Objects;
import java.util.concurrent.ConcurrentHashMap;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Stream;
```

```

public class InMemoryTitleRepository implements ItemStateRepository
{
    private final ConcurrentHashMap<Identity, SimpleState> inMemory = new ConcurrentHashMap<>();
    private final Function<Criteria, MatchingPredicateBuilder> predicateBuilderProvider = MatchingF

    @Override
    public ChangeSet beginCreation(final Identity id)
    {
        return new CreationChangeSet(id);
    }

    @Override
    public void store(final ChangeSet changeSet)
    {
        Objects.requireNonNull(changeSet);
        if (changeSet instanceof CreationChangeSet)
        {
            final CreationChangeSet creation = (CreationChangeSet) changeSet;
            inMemory.putIfAbsent(creation.id, new SimpleState(creation.initialValues));
            return;
        }

        if (changeSet instanceof ModificationChangeSet)
        {
            final ModificationChangeSet modification = (ModificationChangeSet) changeSet;
            final SimpleState currentState = inMemory.get(modification.id);
            final SimpleState newState = currentState.applyChanges(modification);
            inMemory.replace(modification.id, newState);
            return;
        }
    }

    @Override
    public ItemState get(final Identity id)
    {
        return inMemory.get(id);
    }

    @Override
    public void remove(final ItemState state)
    {
        final Identity id = state.get(WellKnownKey.ITEM_PK.key);
        inMemory.remove(id);
    }

    @Override
    public FindResult find(final Criteria criteria)
    {
        final Predicate<ItemState> matchingPredicate = buildMatchingPredicate(criteria);
        final Comparator<ItemState> cmp = ItemStateComparatorCreator.getItemStateComparator(criteria);
        final Stream<ItemState> stream = inMemory.values().stream().map(ItemState.class::cast).filter(matchingPredicate);
        .sorted(cmp);

        return StandardFindResult.buildFromStream(stream).withCriteria(criteria).build();
    }

    private Predicate<ItemState> buildMatchingPredicate(final Criteria criteria)
    {
        final MatchingPredicateBuilder builder = predicateBuilderProvider.apply(criteria);
        return builder.getPredicate();
    }

    static class CreationChangeSet implements ChangeSet
    {
        final Identity id;
        final Map<SingleAttributeKey, Object> initialValues = new HashMap<>();

        public CreationChangeSet(final Identity id)
        {
            this.id = id;
            initialValues.put(WellKnownKey.ITEM_PK.key, id);
            initialValues.put(WellKnownKey.VERSION.key, Long.valueOf(0));
        }
    }
}

```

```

@Override
public void set(final SingleAttributeKey key, final Object value)
{
    initialValues.put(key, value);
}

static class ModificationChangeSet implements ChangeSet
{
    final Identity id;
    final long baselineVersion;
    final Map<SingleAttributeKey, Object> valuesToModify = new HashMap<>();

    public ModificationChangeSet(final ItemState itemState)
    {
        id = itemState.get(WellKnownKey.ITEM_PK.key);
        baselineVersion = itemState.get(WellKnownKey.VERSION.key);
    }

    @Override
    public void set(final SingleAttributeKey key, final Object value)
    {
        valuesToModify.put(key, value);
    }
}
}

```

Product and Data Modeling

SAP Commerce allows you to match your specific business needs through product and data modeling. Good understanding of the type system allows you to reflect in your application your business model. The bean and enum generation feature also serves that purpose.

The topics include:

[Generating Beans and Enums](#)

In SAP Commerce, you can generate custom Java Beans and Enums. In your extension, you can provide a configuration file where you specify the class name, attributes, and a possible superclass of your Java Bean or Enum.

[Product Modeling](#)

There are two ways of modeling your product strategy in SAP Commerce: use the type system or the classification functionality.

[The Type System](#)

A type is a template for objects. Types define product data that objects may carry and specify relations between objects, and also make product data persistent by categorizing the data and relating it to database fields. Every object stored in Platform is a type instance.

Generating Beans and Enums

In SAP Commerce, you can generate custom Java Beans and Enums. In your extension, you can provide a configuration file where you specify the class name, attributes, and a possible superclass of your Java Bean or Enum.

Definitions of these Java Beans or Enums are merged across extensions. For example, if two extensions specify a bean and enum with the same, fully classified class, the generated class includes the attributes from these two extensions. This allows you to extend existing beans and enums, but you don't have to subclass them or replace references to the former bean and enum.

Overview

To generate a Java bean or Enum, you must create a file named `<extensionname> -beans.xml` for your extension. Your beans.xml file must contract the bean.xsd schema compatibility, and it must be located in the **resources** folder of the related extension. The build process picks up this file and generates Java Beans from its content.

Every beans.xml file must fulfill the contract of the correct XML file. The file can contain the following:

- Bean class definition:
 - Every bean class definition can provide the name of the other class that it extends.
 - Every bean class definition can provide a custom template used to render this Java Bean.
 - Every bean class definition can contain any number of property definitions. A property definition is a base to generate getters and setters for properties.
 - Every bean class definition can contain a description.
 - Every property definition must contain a property name and a property type.
- Enum class definition:
 - Every property definition can contain a description.
 - Every enum class definition must contain at least one enum value.
 - Every enum class definition can contain a description.

Such a single bean and enum definition provides the information to create a Java Bean or Enum class that is serializable. It also has a no-argument constructor, and enables access to properties using the getter and setter methods. For enum only, the possible enum values must be provided, and enum is implicitly serializable. All of the types declared above don't require a dependency to any item from the type system.

The same bean and enum definition can be kept in different extensions, meaning different beans.xml files. The expected result is to have one bean and enum definition with a merged list of all attributes. During the merging phase, there is a logical validation performed that checks if the bean and enum definitions fulfill some constraints.

i Note

[Find Out More](#)

beans.xsd File

The build process puts a beans.xsd schema file into the resources folder of the related extension if it finds that there is already a beans.xml file, but no XSD schema file yet.

Examples of beans.xml Files

Empty beans.xml file:

The following is an example of an empty beans.xml file:

```
<!-- [y] hybris Platform Copyright (c) 2000-2012 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:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="beans.xsd">
</beans>
```

Example of a beans.xml file.

The following is an example of `beans.xml` file. Assume that you have two extensions; `custom` and `test`, where `custom` depends on the `test`. This is how the `test-beans.xml` file looks for the `test` extension:

```
<!-- [y] hybris Platform Copyright (c) 2000-2012 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:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="beans.xsd">

    <enum class="de.hybris.test.data.enums.EngineType">
        <description>Engine types.</description>
        <value>diesel</value>
        <value>gasoline</value>
        <value>electric</value>
    </enum>

    <bean class="de.hybris.test.data.beans.Engine" >
        <description>Class describing engine.</description>
        <property name="cylinders" type="java.lang.Integer" >
            <description>Number of cylinders.</description>
        </property>
        <property name="ccms" type="java.lang.Float" />
    </bean>

    <bean class="de.hybris.test.data.beans.CarData" >
        <description>Class describing car.</description>
        <property name="id" type="java.lang.Integer" />
        <property name="type" type="de.hybris.test.data.enums.EngineType" >
            <description>Type of engine used in this car.</description>
        </property>
    </bean>

</beans>
```

This is how the `custom-beans.xml` file looks for the `custom` extension:

```
<!-- [y] hybris Platform Copyright (c) 2000-2012 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:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="beans..>

    <enum class="de.hybris.test.data.enums.EngineType">
        <value>hydrogen</value>
    </enum>

    <bean class="de.hybris.test.data.beans.Engine" >
        <property name="turbine" type="boolean" />
    </bean>

    <bean class="de.hybris.test.data.beans.SportCarData"
          extends="de.hybris.test.data.beans.CarData" template="resource/sport-template.vm" >
        <property name="customExhaust" type="boolean" />
    </bean>

</beans>
```

The above definition produces:

- **EngineType**: Enum class with four values; DIESEL, GASOLINE, ELECTRIC, and HYDROGEN. Three of them are defined in the `test` extension and one is defined in the `custom` extension.
- **CarData**: Java Bean with two ID properties; type of int type and EngineType (enum), respectively.

- **Engine**: Java Bean that contains two properties defined in the `test` extension, and a property defined in custom extension.
- **SportCarData**: Java Bean that extends, or is inherited from, the `CarData` type and provides an additional property called `customExhaust`, of boolean type.

Additionally, a custom velocity template called `sport-template.vm` is defined. The template is located in the `resource` folder of the extension.

Starting Generation

The Java Bean generation feature is attached into the build process and is invoked right after the model generation. It is optimized to be called only if any of the available `beans.xml` files have changed since the last build or if a new `beans.xml` has been created. In other words, you can invoke Java Beans generation manually by calling `ant all` in the command window.

Generated java files are located in `<{HYBRIS_BIN_DIR}>/platform/bootstrap/gensrc` directory.

Generated classes are located in `<{HYBRIS_BIN_DIR}>/platform/bootstrap/modelclasses` directory.

Generation Template

The template is a velocity script. This mechanism enables you to control what is added to the source code of the generated Java Bean sources.

Global Templates

There are two default global templates to render beans and enums. These templates are used if no custom template exists for a specific bean or enum in the `beans.xml` file, or if a given template does not exist. The default templates are:

- For beans: `global-beantemplate.vm`
- For enums: `global-enumtemplate.vm`

You can find them in the `platform/bootstrap/resources/pojo` folder.

Creating a Custom Template

If you had a scenario where you wanted a getter of every bean's property to log the result before return, you would do the following:

1. Create a template by copying `global-beantemplate.vm` (see above) and save it in the `resources` folder of the extension where the `beans.xml` file is stored.

Make your changes to the file, such as add log statements:

```

...
#foreach($i in $imports)
import $i;
#end
// NEW: adding import for log4j
import org.apache.log4j.Logger;

...
{
//NEW: adding LOG member
static protected Logger LOG = Logger.getLogger(${shortClassName}.class);

...
    public $v.type get${StringUtil.capitalize($v.name)}() //create a getter
{

```

```

        LOG.info($v.name); //logger call
        return $v.name;
    }
...

```

2. To refer a given template to be used for generation for a certain bean or enum, add a `template` attribute in the bean or enum definition in the `beans.xml` file.

In the example below, the `car-logged-template.vm` file is used to render a `CarData` Java Bean:

```

<bean class="de.hybris.test.data.beans.CarData" template="resources/car-logged-template.vm"
      <property type="boolean" name="customExhaust" />
      <property name="segment" type="de.hybris.test.data.enums.Segment" />
</bean>

```

i Note

If the specified bean or enum is redefined in other extensions, the last provided custom template is used to render it. If no customized templates exists, the global template is used.

The rule for overwriting the custom template does not apply in case of inheritance. Every specific subtype can have its own template for render.

The generated Java class of `CarData` is shown below:

```

package de.hybris.test.data.beans;

import de.hybris.test.data.enums.EnginType;
import de.hybris.test.data.enums.Segment;
import org.apache.log4j.Logger; //here static import reference

public class CarData implements java.io.Serializable
{
    static protected Logger LOG = Logger.getLogger(CarData.class); //here logger with appropriate

        //for each property
    /** <i>Generated property</i> for <code>CarData.customExhaust</code> property defined at extension
     * private boolean customExhaust; //create a instance attribute
     * //for each property
    /** <i>Generated property</i> for <code>CarData.id</code> property defined at extension <code
     * private Integer id; //create a instance attribute
     * //for each property
    /** <i>Generated property</i> for <code>CarData.segment</code> property defined at extension
     * private Segment segment; //create a instance attribute
     * //for each property
    /** <i>Generated property</i> for <code>CarData.type</code> property defined at extension <code
     * private EnginType type; //create a instance attribute

        public CarData()
    {
        // default constructor
    }

        //for each property

    public boolean getCustomExhaust() //create a getter
    {
        LOG.info(customExhaust); //logger call
        return customExhaust;
    }

        //for each property

    public Integer getId() //create a getter
    {
        LOG.info(id); //logger call
        return id;
    }

        //for each property

```

```

public Segment getSegment() //create a getter
{
    LOG.info(segment); //logger call
    return segment;
}
//for each property

public EngineType getType() //create a getter
{
    LOG.info(type); //logger call
    return type;
}
}

```

Custom Attributes That You Can Use in Template

To make it usable, the velocity context holds a valuable object that is used during rendering final java source class. Depending on the object type being rendered, a different set of objects is available in the context.

Type of Object Being Rendered	Identifier in Velocity Context	Object Type	Object Description
bean/enum	packageName	String	The package name extracted from the bean/enum full class qualifier.
bean/enum	shortClassName	String	The class name extracted from the bean/enum full class qualifier. It can contain generic information, if defined.
bean/enum	StringUtils	org.apache.commons.lang.StringUtils	The Utils class can be used for various string operations like empty checking, stripping, trimming, and joining. For more details, read http://commons.apache.org/lang/api-2.5/...
bean/enum	constructorName	String	The class name extracted from the bean or enum full class qualifier, and is stripped of any generic information.
bean	superclassShortName	String	The short identifier of the class name at the <code>extends</code> identifier in the beans definition file. It can be empty if the bean has no <code>extends</code> attribute defined.
bean	memberVariables	List of MemberVariables instances	Represents an abstraction for properties of the bean. Every MemberVariable instance contains information about: <ul style="list-style-type: none"> • type: Property type name. Short type name of the property. • name: Property name. Literal identifier passed from bean definition.

Type of Object Being Rendered	Identifier in Velocity Context	Object Type	Object Description
			<ul style="list-style-type: none"> comment: A comment string containing information about which extension the property was defined in. overridden: A dynamic flag calculated while merging different beans.xml files for different extensions. It is set to true if the property is overriding some previous declaration in the type hierarchy.
bean	imports	List of Strings	Contains a list of strings containing all the full class identifiers found for all properties. Extends attributes for the given bean definition.
enum	values	List of Strings	List of strings representing enumeration literals as enumeration values from the enum definition in beans.xml file.

Generation Validation

There are two types of validation that is performed while Java Beans are generated:

- Validations that break the generation of Java beans
- Validations that warn about potential problems

Validations that Break the Generation of Java Beans

The validations in this section break the generation of Java beans.

- Empty property name definition

Bean definition in extensionname-beans.xml:

```
<bean class="de.hybris.test.data.class.CarData" ><!-- here is class not allowed keyword -->
    <property name="" type="java.lang.Integer" />
</bean>
```

Console output:

```
[generatebeans] 16:51:08,746 [main] ERROR BeanGenerator - Found empty property name for type
Problem found during processing the file F:\4_6_trunk_test\hybris\bin\custom\training\resourc
```

- Any class identifier of a property that inherits the super class or enum class cannot contain a java keyword:

Bean definition in extensionname-beans.xml:

```
<bean class="de.hybris.test.data.class.CarData" ><!-- here is class not allowed keyword -->
    <property name="id" type="java.lang.Integer" />
```

```
<property name="type" type="de.hybris.test.data.enums.EngineType" />
</bean>
```

Console output:

```
[generatebeans] 16:18:16,886 [main] ERROR BeanGenerator - Class name de.hybris.test.data.clas
```

- A property name must be unique in the whole type hierarchy for a Java Bean:

Bean definition:

```
<!-- bean definition in training extension -->
<bean class="de.hybris.test.data.beans.CarData" >
    <property name="id" type="java.lang.Integer" />
    <property name="type" type="de.hybris.test.data.enums.EngineType" />
</bean>

<!-- bean definition in custom extension -->
<bean class="de.hybris.test.data.beans.CarData" >
    <property name="id" type="java.lang.Integer" />
    <property name="type" type="de.hybris.test.data.enums.EngineType" />
</bean>
<!-- bean definition in one extension -->
```

Console output:

```
[generatebeans] 16:26:04,725 [main] ERROR BeanGenerator - Duplicate definition of attribute ( Problem found during processing the file F:\4_6_trunk_test\hybris\bin\custom\custom\resources
```

- The class name with package must not clash for the bean or enum definition:

Bean definition:

```
<!-- bean definition in training extension -->
<bean class="de.hybris.test.data.beans.CarData" >
    <property name="id" type="java.lang.Integer" />
    <property name="type" type="de.hybris.test.data.enums.EngineType" />
</bean>

<!-- bean definition in custom extension -->
<enum class="de.hybris.test.data.beans.CarData" >
    <value>radio</value>
    <value>gps</value>
</enum>
```

Console output:

```
[generatebeans] [generatebeans] 16:30:57,069 [main] ERROR BeanGenerator - Given 'enum' definition with the same class name, declared at <training>. Problem found during processing F:\4_6_trunk_test\hybris\bin\custom\custom\resources\custom-beans.xml for extension <custom>
```

Validations that Warn About Potential Problems

The validations in this section warn about potential problems.

- Reference order warnings:

i Note

When referencing the custom bean or enum declared in different beans .xml files, you have to be fully aware of the extension build order. In other words, in the extension **training**, you should not reference the bean or enum declared in the extension **custom** if the extension **training** does not require the extension **custom**, directly or indirectly.

Assume that the extension `custom` requires the extension `template`. You can have a system with the extension `template` without the extension `custom`, but you cannot have a system with the extension `custom` without the extension `template`.

Bean definition:

```
<!-- bean definition in training extension -->
<bean class="de.hybris.test.data.beans.CarData" extends="de.hybris.test.data.beans.Vehicle" >
    <property name="id" type="java.lang.Integer" />
    <property name="type" type="de.hybris.test.data.enums.EngineType" />
</bean>
<!-- bean definition in custom extension -->
<bean class="de.hybris.test.data.beans.Vehicle" >
    <property name="power" type="java.lang.Integer" />
</bean>
```

Console output:

```
[generatebeans] 16:42:02,644 [main] WARN ExtensionOrderValidateListener - Databean definitio
extends the type (de.hybris.test.data.beans.Vehicle) which is not declared yet, probably exte
```

Common Mistakes that Stop Generation

The following list provides common mistakes the results in stopping the generation:

- XML syntax errors. However, the generator provides as much information as possible about syntax issues:

Console output:

```
Invalid content was found starting with element 'property'. One of '{bean, enum}' is expected
```

- The template path should refer to the main directory of the extension. You should avoid using absolute paths.
- The bean file name should be in the following format: `<extensionname>-beans.xml`.

Template-Related Validations

You can define a template for a specific bean or enum definition. For more details, see the [Generation Template](#) section.

- Create a custom template for a specific bean or enum definition, if redeclaring the same bean or enum you would override the customized one.

Bean definition:

```
<!-- bean definition in training extension -->
<bean class="de.hybris.test.data.beans.CarData" template="resources/regular-template.vm" >
    <property name="id" type="java.lang.Integer" />
    <property name="type" type="de.hybris.test.data.enums.EngineType" />
</bean>

<!-- bean definition in custom extension -->
<bean class="de.hybris.test.data.beans.CarData" template="resources/regular-template.vm" >
    <property name="id" type="java.lang.Integer" />
    <property name="type" type="de.hybris.test.data.enums.EngineType" />
</bean>
```

Console output:

```
[generatebeans] 17:07:26,365 [main] INFO BeanTemplateListener - Overwriting custom template
late.vm for bean (de.hybris.test.data.beans.CarData) defined at <training> with a new templat
```

- Create a custom template, if a custom template for the target bean or enum does not already exist:

Console output:

```
[generatebeans] 17:18:32,975 [main] WARN BeanGenerator - Given template 'not-exists-template <training> is not accessible - falling back to default one 'global-beantemplate.vm'
```

Improvements in Beans Generating Mechanism

Beans generating mechanism allows you to define annotations for generated classes. Annotations can be generated on the following levels:

- Class
- Member variables
- Getters and setters

To use this functionality, add the annotations as follows:

- In regular Java source files: Include the annotations between the `<annotations> </annotations>` tags.
- For class level annotations: Place the `<annotations>` tag as a child of the `<bean>` tag.
- For member variables, getters, and setters: Place the `<annotations>` tag as a child of the `<property>` tag that describes the member variable you want to annotate. The `<scope>` attribute of the `<annotations>` tag can take one of the following values: `<member>`, `<getter>`, `<setter>`, `<all>`.

You can also mark some member variables as essential for generated `hashCode()` and `equals()` methods. To use them, add the `<equals>` attribute with a value set to `<true>` to the `<property>` tag that the `hashCode()` and `equals()` methods use. If you want a generated `equals()` method to use its parent's `equals()`, add the `<superEquals>` attribute and set its value to `<true>` inside the `<beans>` tag. The default values of the `<equals>` and `<superEquals>` attributes are set to `<false>`, so this improvement doesn't affect the existing bean definitions.

Swagger Documentation Annotations

You can add documentation to a DTO by using the standard `<description>` element in `beans.xml`, and additional `<hints>` element.

On a class level:

```
<description>DTO description</description>
<hints>
    <hint name="wsRelated"/>
    <hint name="alias">sample</hint>
</hints>
```

The `wsRelated` hint is required if you want Swagger documentation annotation to be generated.

The `alias` hint allows you to override a DTO name (default value is taken from a class name). In most cases, it is needed because the marshaller from `webservicescommons` removes the `WsDTO` suffix from a class name (for example, `AddressWsDTO -> address, AddressListWsDTO -> addressList`)

Property level:

```
<description>Property description</description>
<hints>
    <hint name="alias">newName</hint>
    <hint name="required">true</hint>
```

```
<hint name="allowedValues">true, false</hint>
<hint name="example">true</hint>
</hints>
```

The required hint allows you to mark a parameter as required.

The allowedValues hint allows you to define a list or range of values that can be used for this property.

i Note

In case of Swagger, there are three ways to describe the allowed values:

1. To set a list of values, provide a comma-separated list. For example: first, second, third.
2. To set a range of values, start the value with "range", and surrounding by square brackets, include the minimum and maximum values, or round brackets for exclusive minimum and maximum values. For example: range[1, 5], range(1, 5), range[1, 5).
3. To set a minimum/maximum value, use the same format for range but use "infinity" or "-infinity" as the second value. For example, range[1, infinity] means that the minimum allowed value of this parameter is 1.

The example hint allows you to define an example value of a parameter, for example a date in specific format for a date object `<hint name="example">yyyy-MM-ddTHH:mm:ssZ</hint>`.

Support for the @Deprecated Annotation Attributes

The beans.xml and items.xml files can have the optional since and forRemoval attributes of the @Deprecated annotation. You can use these attributes for classes, methods, or enums. The generated java classes have the `@Deprecated(since="xxxx", forRemoval="true")` annotation.

The value of the `java.lang.Deprecated#forRemoval` attribute must always be set to true by the code generator.

For more information, see <https://docs.oracle.com/javase/9/docs/api/java/lang/Deprecated.html> .

Generating and Customizing Events

Platform extensions can signal business state changes by publishing events. To create new event classes, use bean generation, which provides higher transparency and allows for customization.

Declaring a New Event

Platform events must inherit from `AbstractEvent` to be able to publish events via `EventService.publishEvent(event)`. To enable bean generation to do that, add the additional `type="event"` property to the standard `<bean>` tag.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="beans.>
    <bean class="de.hybris.platform.servicelayer.event.events.AfterSessionCreationEvent" type='
</beans>
```

With that, the generated class should be ready to be used as an event. Here is what the code should look like:

```
/*
 * -----
```

```
* --- WARNING: THIS FILE IS GENERATED AND WILL BE OVERWRITTEN!
...
*/
package de.hybris.platform.servicelayer.event.events;

import java.io.Serializable;
import de.hybris.platform.servicelayer.event.events.AbstractEvent;

public class AfterSessionCreationEvent extends AbstractEvent {

    public AfterSessionCreationEvent()
    {
        super();
    }

    public AfterSessionCreationEvent(final Serializable source)
    {
        super(source);
    }
}
```

i Note

As you can see, there are always two constructors: one without an argument and one with a `Serializable` object. We need to provide one with a `Serializable` object, because `AbstractEvent` allows passing a source object at creation time.

Declaring Attributes for Events

Many events need to store some payload information to be consumed by listeners. While you could use the `Serializable` source member inherited by `AbstractEvent`, it's recommended to declare event attributes explicitly as bean properties. The generated class provides getters and setters for them.

This is an example from `core-beans.xml`:

```
<bean class="de.hybris.platform.servicelayer.event.events.AfterSessionAttributeChangeEvent" type="event">
    <property name="attributeName" type="String"/>
    <property name="value" type="Object"/>
</bean>
```

Customization

During a project implementation, it might be necessary to add to an event's payload without actually subclassing it and replacing all places where the event is published.

There are two features that hybris provides for that: extensibility of the event POJO without subclassing (this is a standard bean generation feature) and a hook-in mechanism for decorating events before they're passed on to all listeners.

Extending Events Without Subclassing

As a generated bean, also event classes can be given new properties without subclassing them by any extension inside their `xxx-beans.xml` file.

In the example below, a new property `remoteAddress` is added to the core `AfterSessionCreationEvent`:

```
<bean class="de.hybris.platform.servicelayer.event.events.AfterSessionCreationEvent" type="event">
    <property name="remoteAddress" type="String"/>
```

```
</bean>
```

After a rebuild of the hybris platform, the class now provides getters and setters for the new property.

Injecting Extra Payload via EventDecorator

Events extended as described above still need business logic that actually adds data for the new properties. If the creation of these events can't be replaced, it's possible to implement the `EventDecorator` Spring bean, which is processed as part of `EventService.publishEvent(event)` before invoking listeners.

Next, let's populate the new `remoteAddress` property:

```
package de.hybris.platform.eventgentest;
import de.hybris.platform.servicelayer.event.events.AfterSessionCreationEvent;
import de.hybris.platform.servicelayer.event.impl.AbstractEventDecorator;
import org.apache.commons.lang.StringUtils;
import org.apache.log4j.MDC;

public class AfterSessionCreationEventRemoteAddressDecorator extends AbstractEventDecorator<AfterSessionCreationEvent>
{
    @Override
    public AfterSessionCreationEvent decorate(final AfterSessionCreationEvent event)
    {
        // mind, this is just a sample using logic from Log4JFilter - doesn't need to make much sense
        final String address = (String) MDC.get("RemoteAddr");
        if (StringUtils.isNotBlank(address))
        {
            // adding extra data here
            event.setRemoteAddress(address);
        }
        return event;
    }
}
```

i Note

We evaluate the generic type of decorators for passing only those events that match the type (see `<AfterSessionCreationEvent>` in the previous example).

Add the new decorator to the spring config file.

```
<bean id="AfterSessionCreationEventRemoteAddressDecorator"
      class="de.hybris.platform.eventgentest.AfterSessionCreationEventRemoteAddressDecorator">

    <property name="priority" value="10"/>

</bean>
```

The `priority` property of the event decorator is used to order decorators if there's more than one priority for a particular event. The `priority` property is provided by `AbstractEventDecorator` - otherwise it would be necessary to implement the method `getPriority()` yourself.

Blocking Events

In rare cases, it may be necessary to prevent (some) events from being delivered to any listener. Again, if you are not able to replace all places where the event is being published, `EventDecorator` can help.

To block an event, the decorator simply needs to return `null`. This not only skips notification of listeners but also prevents calling other decorators.

```
public AfterSessionCreationEvent decorate(final AfterSessionCreationEvent event)
{
    if (mustNotPublish(event))
    {
        // stop it right here
        return null;
    }
    else
    {
        // decorate ... whatever
        return event;
    }
}
```

Related Information

[Event System](#)

[Generating Beans and Enums](#)

Product Modeling

There are two ways of modeling your product strategy in SAP Commerce: use the type system or the classification functionality.

Each of these two ways is a technically different approach with individual advantages and disadvantages. In many cases, a mixture of both typing and classification is a good idea.

Typing

You can do typing using the `item.xml` file.

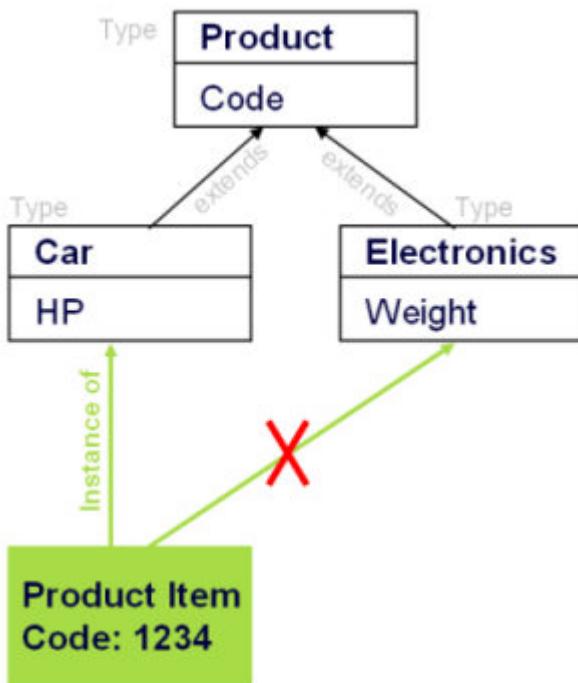


Figure: Types have TypeAttributes

⊕ Possible to generate JavaClasses out of types (runtime → configured types)

⊕ Less load on database

- No Multi-Inheritance

- No Re-Typing

Classification

Places to do Classification:

- In the Backoffice application: Classification Systems
- CSV-based: Create your own CSV Import.

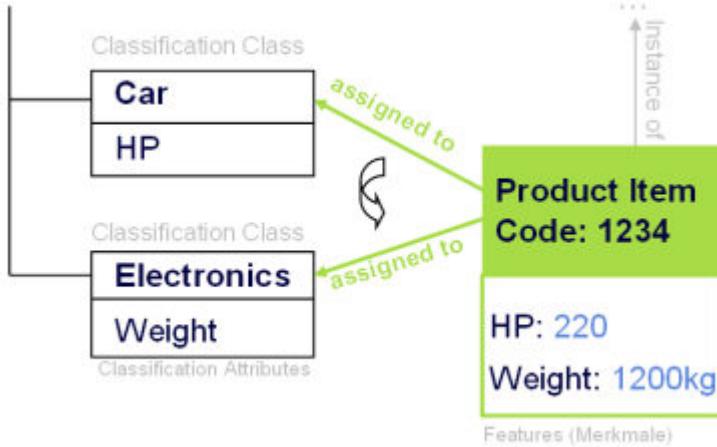


Figure: Classification classes have classification attributes. The associated values of classification attributes are called **Features**.

+ Multi-Assignments to Classification Classes

+ Re-Classification or Un-Classification at any time

- More load on database

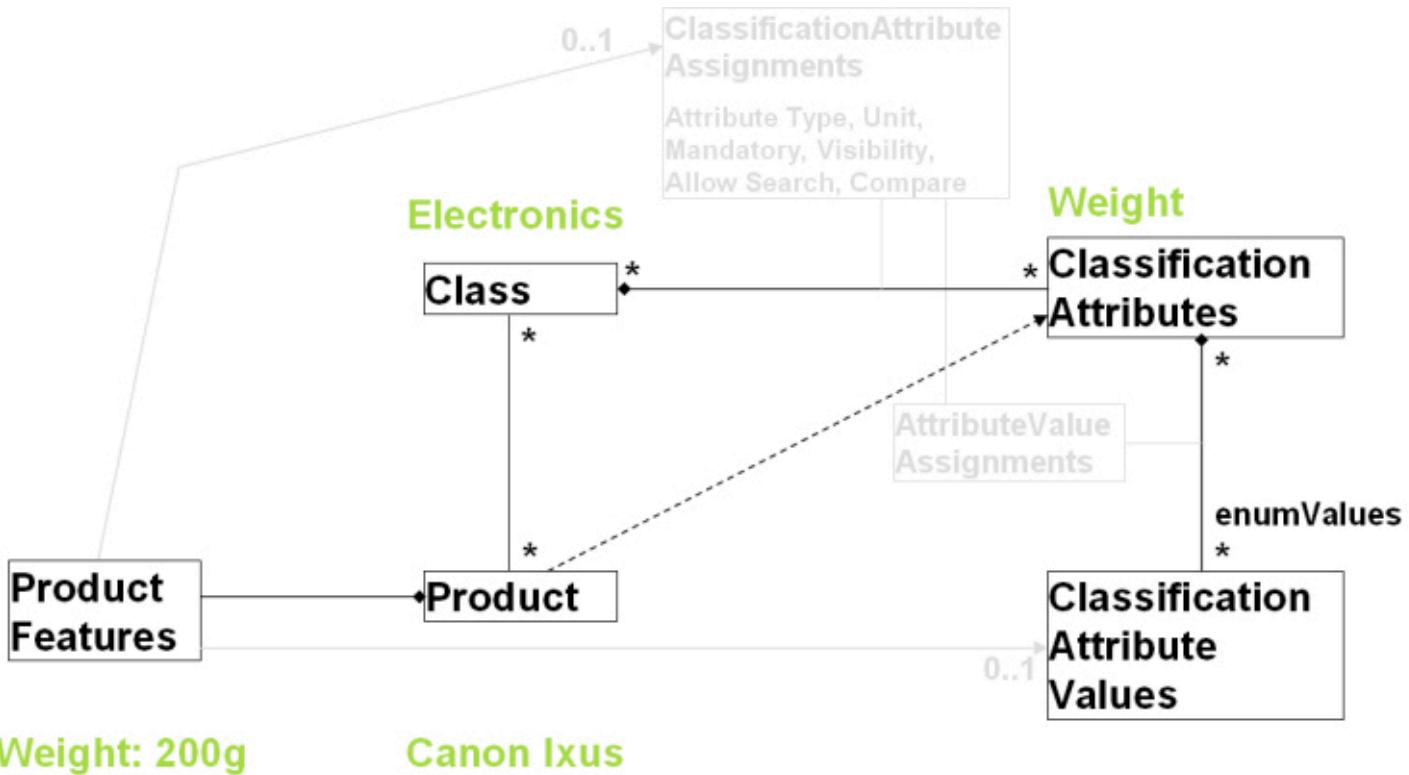
- No delegation to own java classes, therefore no logical implementation of attributes possible (JaloAttributes)

→ Tip

Best Practice

In many cases, a mixture of both typing and classification is useful, and a best practice approach is to use a subtype of Product. This subtype holds all the product properties, reflected using the type system attributes or classification features. Whether a certain product property should be modeled using the type system or the classification system depends on whether the product property is used in business logic, or whether the property is an item to display. A product property frequently used in business logic, such as an article number, should be modeled using a type system attribute. A display property such as a screen resolution of a monitor, which is seldom used in business logic, can be modeled using a classification feature. However, this is a general guideline and may not match the product model for your case exactly.

Product Classification



Related Information

[Defining Product Attributes](#)

[The Type System](#)

[Classification](#)

The Type System

A type is a template for objects. Types define product data that objects may carry and specify relations between objects, and also make product data persistent by categorizing the data and relating it to database fields. Every object stored in Platform is a type instance.

Main Functions of the Type System

SAP Commerce uses a system of types to organize data, for example product information, customer data, addresses, or orders.

Types define persistent objects in several aspects:

- **attributes** manage and store data for the object,
- the **deployment** defines the database table the object is stored in (see [Specifying a Deployment for Platform Types](#) for more details)
- the Java class of the object.

A Type is the type definition in `items.xml` and its Java implementation.



An object instance of a type is called an item:

Platform concept	Java concept
type	class
item	object / instance

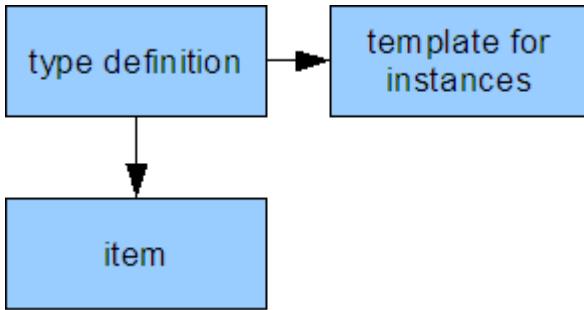
Items and Types

There are two major kinds of types: System-related types and business-related types.

	Data Types	Infrastructure Types
Java:	java.lang.Boolean java.lang.Integer java.lang.String java.lang.Enum	java.util.Date java.util.Map java.lang.Object
SAP Commerce:	java.lang.Boolean java.lang.Integer java.lang.String java.lang.Enum	GenericItem ComposedType Relation

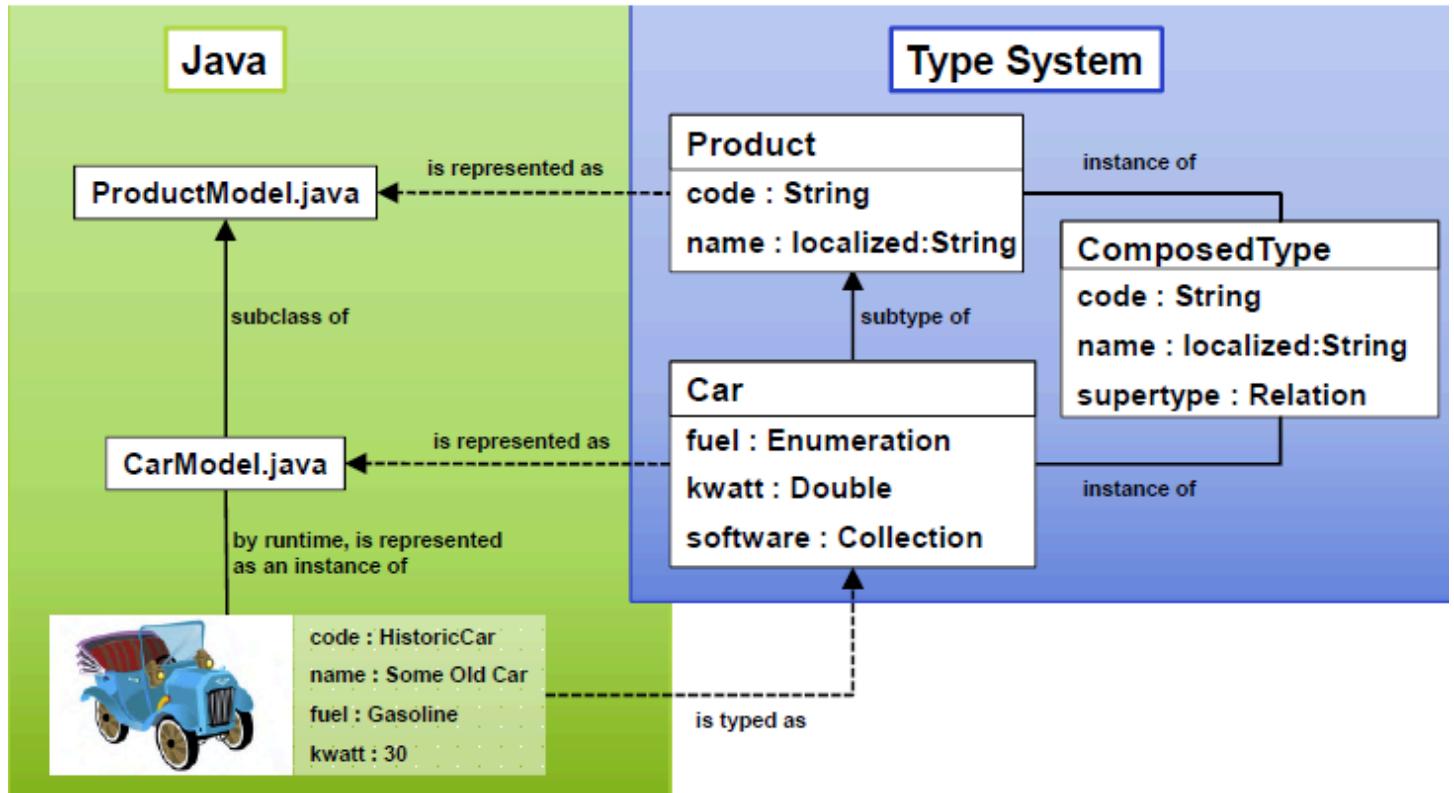
1. System-related types make up or extend the type system itself and deal with internal data management:
 - a. Infrastructure types: ComposedTypes (also referred to as ItemTypes) set up type definitions and may carry attributes to hold information. In the end, every persistent object in the SAP Commerce is an instance of ComposedType or of one of its subtypes.
 - b. Data types: CollectionTypes, MapTypes, EnumerationTypes, and AtomicTypes. These are used to describe attributes: carrying attribute values or representations for these values or creating links between objects
2. Business-related types (like Order, Discount, Shoe) allow you to manage product and / or customer information so that you can run your business.

For the out-of-the-box structure and hierarchy of the type system, consult the `core-items.xml` file.



As mentioned before, every object stored in SAP Commerce is an instance of a type. Even type definitions are instances of the type Type. This means that there are two aspects of a type definition: it is an item and, at the same time, it defines other items.

Java Classes vs Type System



The Item type is the supertype of all types in SAP Commerce.

i Note

Items and items

To differentiate between normal object instances and type definitions, non-type objects in Platform are referred to as **items**. The lower case spelling **item** refers to an object in Platform; the upper case spelling **Item** refers to the type definition.

Type definitions are stored as instances of ComposedType (or a subtype of ComposedType), so the definitions of Item and Product as items are instances of ComposedTypes.

You may want to define subtypes of ComposedTypes with additional attributes. The following code snippet defines an item called SpecialProduct that is a subtype of Product but its type is not defined as a ComposedType, but as a SpecialComposedType (via the metatype attribute). Instances of SpecialProduct are thus subtypes of Product, but the type definition is stored as a SpecialComposedType.

```
<item code="SpecialProduct"      metatype="SpecialComposedType"  extends="Product">
```

Types and Attributes

Types may define attributes, which is the equivalent to Java classes having fields. You may easily edit item values in Backoffice as they are displayed in graphical editor elements. On the API level, get hold of those attributes via getter and setter methods.

An attribute in SAP Commerce:

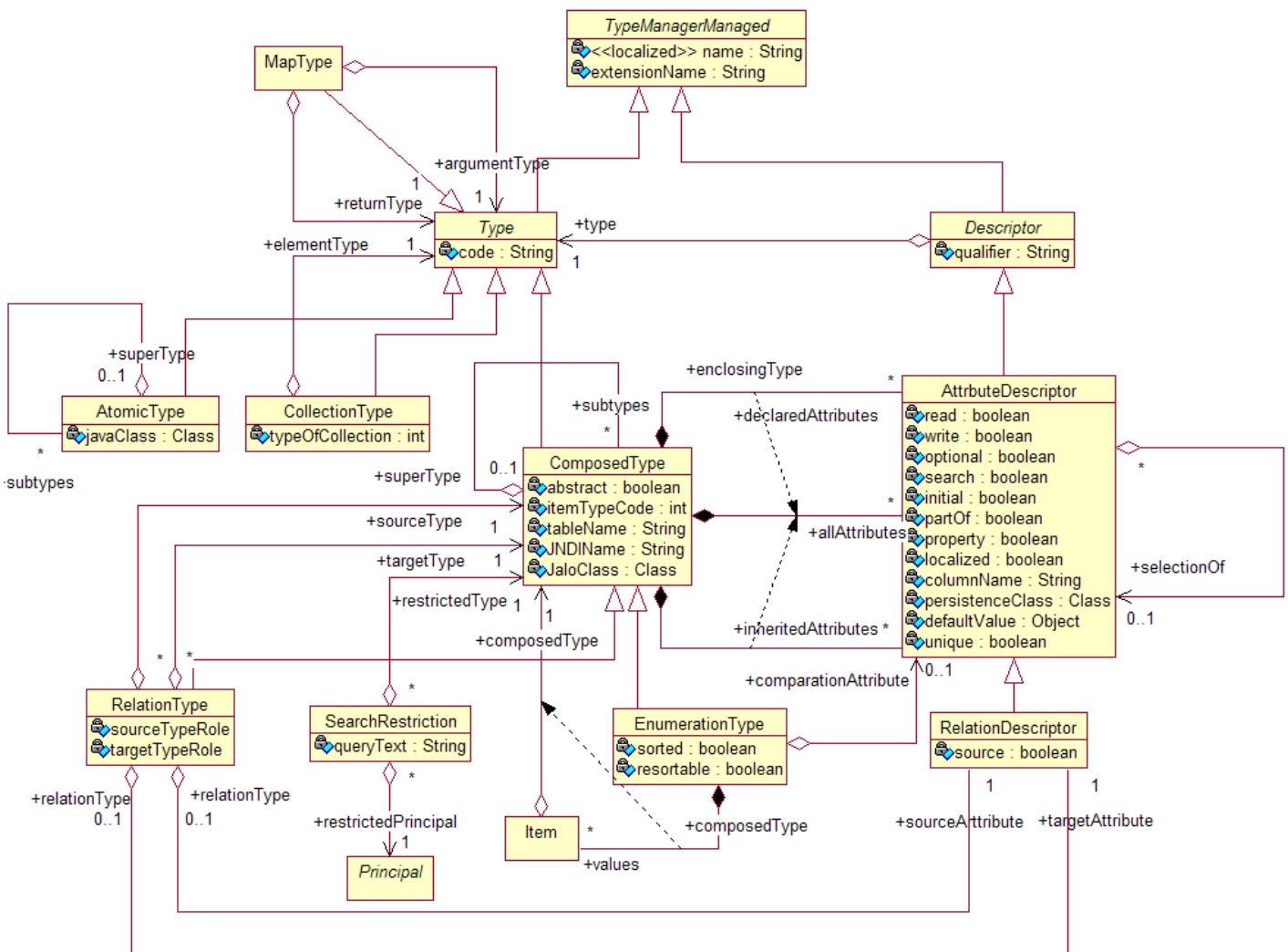
- can be a reference to
 - a composed type, for example the **Folder** in the **Media** type
 - a basic Java type, called the **atomic type**, for example the **Description** attribute is of type `java.lang.String`
- can have a localized name and description,
- can have a default value.

Configured Types vs Runtime Types

In terms of persistence, there are two kinds of types: configured and runtime.

Configured types are types that are defined in an `items.xml` file.

The following UML diagram gives you an overview of selected types and their attributes.



Runtime Types - with no definition in `items.xml` - are only defined at run time in Backoffice.

i Note

Be aware of the following limitations of runtime types:

- Persistence aspects

Runtime types are as persistent as all other items, since type definitions are stored in the database as well. However, when SAP Commerce is initialized, the entire type system is discarded and re-created from the contents of the `items.xml` files of all extensions. As runtime types are not backed by a file, they are removed and not re-created during a system initialization. In other words: whenever you initialize SAP Commerce, all runtime types are gone.

- Lack of Java sources

As the build process for extensions relies on the `items.xml` to generate Java source files, there will be no Java source files for runtime types. By consequence, this means that you cannot adapt runtime types programmatically.

In some use cases, using runtime properties might be more flexible. It is, however, strongly suggested to test and confirm if runtime properties meet all the requirements of your system.

Adding New Attributes to a Type

To add attributes to an SAP Commerce type, you can:

1. Extend the type and adding the attribute to a subtype

2. Add attributes to a type directly.

For a discussion of these two approaches, see [items.xml](#).

Checking Runtime Attributes Type by the Type System

When reading runtime attributes, the type system checks the types of primitive attributes. If the type declared in the attribute descriptor does not match the database value, a `null` value is returned. If you want to disable this behavior, navigate to the `local.properties` file and add the following property with the value `false`:

```
should.check.runtime.attributes.type.match=false
```

Available Types

There are a few available types and they serve specific purposes. For example, a `MapType` is a typed collection of key/value pairs that you can use for localized values.

AtomicTypes

`AtomicTypes` are the most basic types available in SAP Commerce. They are the representation of Java Number and String object types, such as `java.lang.Integer` or `java.lang.String`. SAP Commerce has mappings for the common Java number and String object types. As the factory default `AtomicTypes` represent the most common Java number and String types, most probably you might not have to define `AtomicTypes` anyway. When you define an `AtomicType` yourself, you need to assign a Java class object to it.

→ Tip

For details on a list of the `AtomicTypes` that are generated upon Platform initialization, please refer to the `atomictypes` section of the `core-items.xml` file in the `<HYBRIS_HOME>/bin/platform/ext/core/resources` directory.

Unlike the other types, an `AtomicType` definition does not have a `code` attribute to set the unique identifier. Instead, the `AtomicType` `class` attribute is used as its reference.

If you need to localize this referrer into various languages, you may do so in the type localization files (`locales_xx.properties`, located in an extension `resources/localization` directory) - the following code snippet assigns a localization to the `java.util.Date` `AtomicType`:

```
### Localization for type localized:java.util.Date
    type.localized:java.util.date.name=Date
    type.localized:java.util.date.description=This is a localized info
```

SAP Commerce stores `AtomicType` instances in the database as strings (`VARCHAR`) or numbers (`NUMBER`), if possible. If you define `AtomicTypes` yourself, you need to make sure that those types are serializable (that is, you need to find a storage format for them that your database system can manage).

CollectionTypes

i Note

Use `RelationTypes` whenever possible.

As the maximum length of the database field of a CollectionType is limited, a CollectionType with many values may end up getting its values truncated. In addition, the values of CollectionTypes are written in a CSV format and not in a normalized way. By consequence, SAP recommends using RelationTypes whenever possible.

As you need to define the type of items that are stored in a CollectionType from the very start, it is impossible to store any other type of items in the same CollectionType. In the example code snippet, the `StringCollection` type is restricted to String type items only. SAP Commerce blocks any attempt of storing, for example, Integer values into a `StringCollection` instance. A comparable concept is the idea of [Java 5 Generic Types](#).

A CollectionType contains a typed number of instances of types (a dozen Strings, for example). A CollectionType has a unique identifier (referred to as **code**) and a definition of the type of elements it contains (**elementtype**). This type definition may include any type of item within SAP Commerce, even other CollectionTypes.

CollectionTypes are based on the Java Collection class. Via the `type` attribute in a CollectionType definition, you can make use of the Collection class and some of its subclasses (List, Set, and SortedSet). A [Collection](#) is a list of elements. A [List](#) is a number of ordered items. Although there may be equal items in a List, the items order is relevant. Items in a List may be accessed by an index counter. A [Set](#) is an unordered number of items that must all be unique, no two items may be equal. A [SortedSet](#) is the combination to a List and a Set. It contains a number of ordered items that must all be unique.

There are two types of relations that you can build with CollectionTypes: **one to many** relations and **many to one** relations. Both kinds of relation are unidirectional by design.

One-to-many relations (1:n)	Many-to-one relations (n:1)
<p>Keep links to the respective values via an attribute on the source item, for example, a list of Primary Keys.</p> <pre> graph LR Country[Country] --> regions["regions
(list of PKs)"] regions --> Region1[Region1] regions --> Region2[Region2] regions --> Region3[Region3] regions --> Region4[Region4] </pre> <p>In the graphic, the <code>Country</code> type has an attribute <code>regions</code> that stores a list of the PKs of the regions.</p> <p>If the CollectionType contains AtomicTypes, the values are stored as binary fields in the database. If it stores a collection of items, then those items' Primary Keys (PKs) are stored in the database in string form - a list of PKs, basically. As all the values of one CollectionType instance are stored as one single field, reading in and writing the values is quite fast as it is done in a single database access (especially with caching). Processing them, however, is more delicate for three reasons:</p>	<p>Store the attribute values at the respective target items and have a getter method at the source type to retrieve the values.</p> <pre> graph LR Country[Country] --> getAllRegions["getAllRegions()"] getAllRegions --> Region1[Region1] getAllRegions --> Region2[Region2] getAllRegions --> Region3[Region3] getAllRegions --> Region4[Region4] </pre> <p>In the graphic, <code>Country</code> has a getter method called <code>getAllRegions()</code> that runs a FlexibleSearch statement on the database to find all instances of <code>Region</code>.</p> <p>Unlike the one-to-many kind of relation represented by a CollectionType, this kind of relation:</p> <ul style="list-style-type: none"> • has better performance as the results of FlexibleSearch statements are cached by SAP Commerce • does not suffer from a field length limitation as the relation is not established via an attribute and, therefore, no PKs are stored directly.

One-to-many relations (1:n)	Many-to-one relations (n:1)
<ul style="list-style-type: none"> If a collection contains a lot of PKs, the field value may reach the maximum length of field for the database implementation and entries may get truncated. That means that you can only store values of a certain length in that database field and every bit of information beyond that length gets lost. As the database entry only contains the PKs (in other words: links to items) and not the items themselves, you cannot run database searches on the entries directly. Instead, you need to run searches in memory via Java, which is often slower than searching on the database directly. If a single CollectionType instance has several AtomicType entries that match a search query, you are not able to detect the exact number of matches from the database directly. 	<ul style="list-style-type: none"> requires implementing the getter method (via the FlexibleSearch statement).

As CollectionTypes have technical limitations that make modeling n:m relations delicate, SAP recommends using **RelationTypes** to model complex correlations.

EnumerationTypes

Very much like the Enum concept in Java or C, EnumerationTypes (EnumTypes for short) map a predefined verbatim value to another, internal kind of value. EnumTypes are ComposedTypes and handle values in a special way: the values are also their instances. Therefore, an EnumType called `color` with the values `red`, `green`, and `blue` has three instances: `red`, `green`, and `blue`. This kind of type is useful for attributes whose values only have a limited number of choices (yes or no, for example).

If you define EnumTypes yourself with default Platform items as values, those values are always instances of this new EnumType. As a result, all the values for self-defined EnumTypes are stored in the same database table as the EnumType itself. Since all EnumTypes end up in one single database table, however, this table might become quite large when you define a lot of EnumTypes. If you know that an EnumType might have a large number of values, SAP recommends using a different database table to deploy this EnumType.

MapType

A MapType is a typed collection of key/value pairs. For each key (referred to as argument), there is a corresponding value (referred to as return type). The direction of mapping is always argument - return value.

A very common use of MapTypes is localized values - values that may differ in every language available in the system, like product descriptions in German and English, for example.

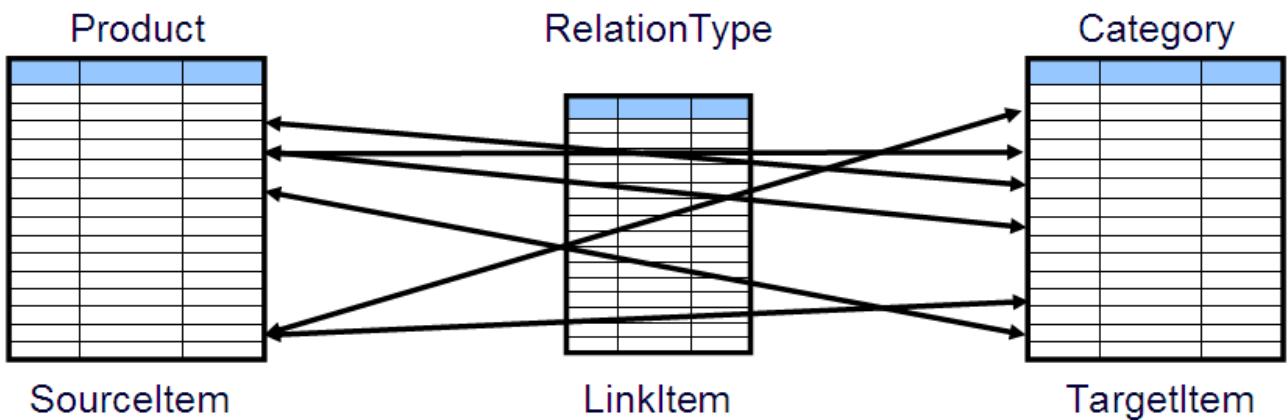
i Note

Storage of localized values

Localized values are stored in a separate database table, whose name is composed of the name of the table the type is stored in, plus the suffix `lp` (short for localized property). For example: if the type is stored in the table `samplertype`, then its localized values are stored in the table `samplertypelp`. You do not have to worry about the handling, though - Platform manages localized types transparently.

RelationTypes

RelationTypes model dependencies between numbers of items on both dependency sides. They represent n:m relations in SAP Commerce. RelationTypes allow you to reflect instances with many products belonging to several categories. The following diagram illustrates these relationships:



Internally, the elements on both sides of the relation are linked together via instances of a helper type called LinkItem. LinkItems hold two attributes, SourceItem and TargetItem, that hold references to the respective item (as in the graphic).

For each entry within a relation (in other words, for each link from one item to another), there is a LinkItem instance that stores the PKs of the related items. LinkItem instances are handled transparently and automatically by the Platform: On the API level, you only need to use the respective getter and setter methods.

If you delete an item from a relation, neither the item you seem to delete nor its related item is deleted, only the LinkItem is removed. In other words, both the source item and the target item remain, only the link between them is removed.

When the Platform runs a search for either side of a RelationType, it runs through all the relation's LinkItem instances and returns a Java Collection that contains the values. SAP Commerce sorts RelationType instances in source-to-target direction by the order the individual relation entries were created. The target-to-source direction is not sorted by the Platform, therefore the order of its results may vary depending on the database you use (that is, without any sorting by your web application or the SQL statements).

ItemTypes

ItemTypes (more commonly referred to as ComposedTypes) are the foundation of the SAP Commerce type system. All types (and therefore items) are ultimately derived of a ComposedType. ComposedTypes hold meta information on types and the types' attributes and relations, including the item type's code (in other words: unique identifier), its JNDI deployment location, the database table the item is stored in and the type's Java class.

Every type (and therefore item) may have any number of attributes. These attributes may either be defined by the type's AttributeDescriptors or inherited from its supertypes. Every attribute that is inherited downwards has its settings stored separately for each children type. That way, it is possible to override attribute access rights inherited from the supertype for a child type, so that you may set an attribute to be **writable** for your self-defined types that wasn't set **writable** on the type the attribute was originally defined with.

Moving a Type

It is possible to move types between extensions. This is useful, for example, when you want to refactor your extension.

Context

These are the only limitations for moving types:

- You are not allowed to change the deployment typecode of the type that you want to move or is already moved.
- You cannot move a type if it affects a classpath. Below you have an example scenario when it does not work:
 - You have three extensions: MyExtensionA, MyExtensionB, MyExtensionC
 - The MyExtensionA contains a TypeA
 - The MyExtensionB and the MyExtensionC are related to the MyExtensionA
 - The MyExtensionC contains a TypeC, which extends the TypeA
 - You want to move the TypeA from the MyExtensionA to the MyExtensionB.

In this case, compilation fails, because the MyExtensionC still needs the TypeA that is used to extend the TypeC.

Possible workaround would be to relate the MyExtensionC with the MyExtensionB, but it may not fit your business objectives.

Below you can find all steps based on example of compiled and initialized SAP Commerce with two extensions: MyExtensionA and MyExtensionB. You want to move TypeA from MyExtensionA to MyExtensionB:

This is the definition of a type that you want to move:

`myextensiona-items.xml`

```
<itemtype
    generate="true"
    code="myTypeA"
    jaloclass="de.hybris.myextensiona.jalo.MyTypeA"
    extends="GenericItem"
    autocreate="true" >

    <deployment table="mytype_deployment" typecode="12345"/>
    ...
</itemtype>
```

Procedure

1. Move `MyTypeA.java` from the MyExtensionA extension to the MyExtensionB extension.
2. Modify package in the `MyTypeA.java` so it reflects the new location.

```
package de.hybris.myextensionb.jalo;
```

3. Remove `GeneratedMyTypeA.java` that was generated for the type in the MyExtensionA extension.
4. Cut the TypeA definition from the `myextensiona-items.xml` and paste it in the `myextensionb-items.xml`.
5. Change package name in the `jaloclass` attribute to the new one.

```
<itemtype
    generate="true"
    code="myTypeA"
    jaloclass="de.hybris.myextensionb.jal
    extends="GenericItem"
    autocreate="true" >

    <deployment table="mytype_deployment">
    ...
</itemtype>
```

6. Build SAP Commerce

- a. Open a command shell.
- b. Navigate to the <HYBRIS_BIN_DIR> /platform directory.
- c. Make sure that a compliant Apache Ant version is used.

On Windows systems, call the <HYBRIS_BIN_DIR> /platform/setantenv.bat file. Do not close the command shell after this call as the settings are transient and would get lost if the command shell is closed. On Unix systems, call the <HYBRIS_BIN_DIR> /platform/setantenv.sh file, such as: . ./setantenv.sh.

- d. Call ant clean all to build SAP Commerce.

7. Enter hybrisserver.bat to start the SAP Commerce Server.

8. Open SAP Commerce Administration Console.

- a. Go to the Platform tab and select **Update** option.
- b. Click the **Update** button.

For details, see [Administration Console](#) and [Initializing and Updating SAP Commerce](#).

Creating Items

There are two ways to create instances of types, in the extension's manager and generically.

Even though you can set up many Platform items via Backoffice or during the System Initialization process (in the form of sample data), there are situations when you need to create instances of types at runtime (when a new customer registers with the web shop, for example).

i Note

Make sure to add the “unique” index on **key attributes** to any new instances of types to prevent creating duplicate items.

Creating Items in the Extension's Manager

Every extension has a Manager that is responsible for item handling. The Manager defines creation methods for all types in the extension. The CronJobManager type, for example, has methods like public Trigger createTrigger(Map params), public CronJob createCronJob(Map params), public BatchJob createBatchJob(Map params), and so on. In the end, these methods are delegates to the createItem(...) method.

By calling the Manager's creation methods and passing the necessary parameters for the new item as a Map (attribute - value), you can create items. The following code snippet gives you an example of this:

```
...
Map params = new HashMap();
params.put( HelloWorldWizardCronJob.SCREENTEXT, getScreenText() );
params.put( HelloWorldWizardCronJob.ACTIVATE, isActive() );
params.put( HelloWorldWizardCronJob.INTERVAL, getInterval() );
params.put( HelloWorldWizardCronJob.CODE, "HelloWorldWizardCronJob" + String.valueOf( jobNum ) );
params.put( HelloWorldWizardCronJob.JOB, hwwj );
HelloWorldWizardCronJob hwwcj = HelloWorldWizardManager.getInstance().createHelloWorldWizardCronJob
...

```

Creating Items Generically

The ComposedType type defines and implements the `public Item newInstance(SessionContext ctx, Map attributeAssignment) throws JaloGenericCreationException, JaloAbstractTypeException` method. To create a new instance of a certain type, select the respective ComposedType and call its `newInstance(...)` method, as in the following sample code snippet:

```
public MySampleItem createMySampleItem( SessionContext ctx, Map params )
{
    try
    {
        ComposedType item = getSession().getTypeManager().getComposedType( "mySampleItem" );
        return (MySampleItem) item.newInstance( ctx, params );
    }
    catch( JaloBusinessException e )
    {
        throw new JaloSystemException(e,"error creating mySampleItem.",0);
    }
}
```

If you create items that way, pass any critical parameters (such as `code` or `qualifier`, catalog versions, etc.) directly in the Map. Otherwise, the creation fails with an exception because these necessary bits of information are not available for the item at creation time.

Overriding the Default Creation Mechanism

In most cases, the out-of-the-box creation mechanism for instances of any type suffices and you do not have to implement a specific implementation of an item's creation. However, when you have very specific item definitions, it might be useful or even necessary to override the default creation logic.

There are two kinds of attributes an item can have: initial attributes and non-initial attributes. Initial attributes need to be set when the item is created. Initial attributes with no value set cause the item creation to fail (PK, qualifier, date, time, and catalog version, for example). Non-initial attributes can be set when the item exists. In other words: initial attributes must be set during item creation, non-initial values can be set later on.

You can set an attribute to be initial by setting its `initial` modifier in the `items.xml` file to true, as in the following code snippet (taken from the CronJob extension's `items.xml` file):

```
<attributes>
    <attribute qualifier="code" type="java.lang.String">
        <modifiers initial="true"/>
        <persistence type="property"/>
    </attribute>
</attributes>
```

An easy way of providing values for initial attributes is by passing those values to the `createItem(...)` method directly. The following code snippet shows you what a `createItem(...)` implementation might look like:

```
protected Item createItem( SessionContext ctx, ComposedType type, ItemAttributeMap allAttributes )
{
    // make sure that all attributes we need have values set
    Set missing = new HashSet();
    if(
        !checkMandatoryAttribute( CODE, allAttributes, missing ) ||
        !checkMandatoryAttribute( UNITTYPE, allAttributes, missing )
    )
    // throw exception if anything is missing
    throw new JaloInvalidParameterException( "missing parameter " + missing + " got " + allAttributes );

    // create new Unit based on the values
    return JaloSession.getCurrentSession().getProductManager().createUnit(
        (String) allAttributes.get( PK ),
```

```

        (String) allAttributes.get( UNITTYPE ),
        (String) allAttributes.get( CODE )
    );
}

```

Although there are several methods you can override, overriding the `createItem(...)` method will do for most cases.

Method Name	Description / Comment
<pre> protected Item createItem(SessionContext ctx, ComposedType type, ItemAttributeMap allAttributes) throws JaloBusinessException </pre>	The actual creation method. Implement it so that its return value is the new item. Marked <code>abstract</code> in the <code>Item</code> type.
<pre> protected ItemAttributeMap getNonInitialAttributes(SessionContext ctx, ItemAttributeMap allAttributes) </pre>	Implement this so that it filters <code>allAttributes</code> and removes values that were added during <code>createItem(...)</code> . If you override this method, be sure to call the super class' implementation of this via <code>super.getNonInitialAttributes(ctx, allAttributes)</code> otherwise you would have to implement the removal values the default implementation would have removed already.
<pre> public void setNonInitialAttributes(SessionContext ctx, Item item, ItemAttributeMap nonInitialAttributes) throws JaloBusinessException </pre>	This method has to set all (non-initial) values returned by the <code>getNonInitialAttributes(...)</code> in other words, all values that have not yet been set yet.
<pre> getInitialProperties() </pre>	Provides all properties that are written directly when the item is created in the database. All attributes that are not created during this method's run are only created when <code>setNonInitialAttributes()</code> runs.

i Note

The `getInitialProperties()` method is a bit special, so it is discussed here a bit more extensively. When a new instance of a `ComposedType` is created, its attributes' values are not set when `setNonInitialProperties()` is run. If you try to set an attribute that depends on another attribute to be set (like `TaxValue` that depends on `Currency`), you are likely to run into exceptions. To avoid this, you may use the `getInitialProperties()` method to find out what attributes are set and design your error handling accordingly. An example listing for both `getInitialProperties()` and `getNonInitialProperties()`:

```

// build a property container and append value
protected JaloPropertyContainer getInitialProperties( JaloSession jaloSession, ItemAttributeMap
{
    final JaloPropertyContainer cont = jaloSession.createPropertyContainer();
    cont.setProperty(MY_ATTR,(Serializable)allAttributes.get( MY_ATTR ));
    return cont;
}

// make sure the attribute is not written twice
protected ItemAttributeMap getNonInitialAttributes( SessionContext ctx, ItemAttributeMap allAttr
{
    final ItemAttributeMap copyMap = super.getNonInitialAttributes(ctx, allAttributes);
    copyMap.remove( MY_ATTR );
    return copyMap;
}

```

The following example is taken from the SAP Commerce core, and shows generic creation of the unit item type:

```

protected Item createItem( SessionContext ctx, ComposedType type, Map allAttributes )
throws JaloBusinessException
{
Set missing = new HashSet();
if ( !checkMandatoryAttribute( CODE, allAttributes, missing ) ||
    !checkMandatoryAttribute( UNITTYPE, allAttributes, missing ) )
    throw new JaloInvalidParameterException( "missing parameter "+missing+
    " got "+allAttributes, 0 );
return JaloSession.getCurrentSession().getProductManager().createUnit(
    (String)allAttributes.get( UNITTYPE ),
    (String)allAttributes.get( CODE )
);
}

protected Map getNonInitialAttributes( SessionContext ctx, Map allAttributes )
{
// super.getNonInitialAttributes provides a copied map so we dont need to copy it again
final Map copyMap = super.getNonInitialAttributes( ctx, allAttributes );
copyMap.remove( CODE );
copyMap.remove( UNITTYPE );
return copyMap;
}

```

Checking the Mandatory Item Attributes

When the `createItem(...)` method is called, one of the call parameters is a Map containing the initial values for the item's attributes. To find out during the `createItem(...)` method whether or not the `ItemAttributeMap` contains a value for each mandatory attribute, you can use the `checkMandatoryAttribute(...)` method. Be sure to test every mandatory attribute via an individual `checkMandatoryAttribute(...)` method call, such as:

```

...
checkMandatoryAttribute(MyType.MYATTRIBUTE1, allAttributes, missing, true);
checkMandatoryAttribute(MyType.MYATTRIBUTE2, allAttributes, missing, false);

```

This method has four parameters:

Parameter	Mandatory	Description
String qualifier	yes	Reference to the attribute to check
ItemAttributeMap allAttributes	yes	Map with the initial attribute values
Set missingSet	yes	Set which accepts references to all attributes for which no value has been set
Boolean nullAllowed	no	Specifies whether a null value in the <code>ItemAttributeMap</code> is written to the item as a null value (<code>true</code>) or whether the null value is treated as a missing value (<code>false</code>). Defaults to <code>false</code> .

The following code snippet gives an example on a `createItem(...)` method implementation with the `CustomerReview` type's three mandatory attributes `product`, `user`, and `rating`:

```

@Override
public CustomerReview createItem(SessionContext ctx, ComposedType type, ItemAttributeMap allAttributes)
{
    final Set missing = new HashSet();
    checkMandatoryAttribute(CustomerReview.PRODUCT, allAttributes, missing);
    checkMandatoryAttribute(CustomerReview.USER, allAttributes, missing);
    checkMandatoryAttribute(CustomerReview.RATING, allAttributes, missing);
    if (missing.size() != 0)
    {
        throw new JaloInvalidParameterException("missing " + missing + " for creating a new item");
    }
    return (CustomerReview) super.createItem(ctx, type, allAttributes);
}

```

Alternatively, you could use the return parameter of the `checkMandatoryAttribute(...)` method of Boolean type. This Boolean return value is `true` if there is a value for the attribute and `false` if the value is missing.

The `missing` HashSet contains every attribute for which no value has been provided. Therefore, if the `missing` HashSet is empty, every attribute has a value and the item can be safely created.

```

public CustomerReview createItem(SessionContext ctx, ComposedType type, ItemAttributeMap allAttributes)
{
    final Set missing = new HashSet();
    if (
        !checkMandatoryAttribute(CustomerReview.PRODUCT, allAttributes, missing) ||
        !checkMandatoryAttribute(CustomerReview.USER, allAttributes, missing) ||
        !checkMandatoryAttribute(CustomerReview.RATING, allAttributes, missing)
    )
    {
        throw new JaloInvalidParameterException("missing " + missing + " for creating a new item");
    }
    return (CustomerReview) super.createItem(ctx, type, allAttributes);
}

```

Relations

There are four basic kinds of relations: one-to-one, one-to-many, many-to-one, and many-to-many.

i Note

This section abides by the [EJB 3.0 Specification](#) terms for relations. Refer to the [JSR-000220 Enterprise JavaBeans 3.0 Final Release \(ejbcore\)](#) specification pages 153 through 171 if some of the terms are unfamiliar to you.

- One-to-one (1:1) relations represent a direct relationship between two items, for example a user name and the user account's ID. This relation is unambiguous; if you know the user name, you can (basically) retrieve its user account ID and vice versa.
- One-to-many (1:n) relations reflect an item that has to do with several other items. For example, a hotel that has several employees, or a tree that has several apples growing on it.
- Many-to-one (n:1) relations are very much like one-to-many relations, only the other way around. Even if implemented differently, from a logical point of view, a many-to-one relation is just another representation of a one-to-many relation.
- Many-to-many (n:m) relations are for those cases where several items of one kind are linked to several items of another kind. For example, if a company has several employees who work spread across several project teams. That way, an employee may be member of several teams and a team may consist of several members. Many database systems (including SAP Commerce) internally resolve an n:m relation into two separate 1:n relations, linked by a common entity. In the employee-project team example, the common entity is the project.

In addition, a relation can be unidirectional (one-sided) or bidirectional (dual-sided). Unidirectional relations point from A to B, whereas bidirectional relations point from A to B and from B to A.

SAP Commerce knows these kinds of relations:

Kind of Relation	SAP Commerce Type
one-to-one, unidirectional	(attribute definition, such as Product instance - Unit instance)
one-to many, unidirectional	CollectionType
many-to-one, unidirectional	CollectionType
many-to-many, bidirectional	RelationType

We have already seen 1:1 relations in our sample with the attributes that contain a product and a customer, respectively. Localized values are some sort of an 1:n relation - there is a single object (the value to be localized) with several other objects (localized text strings) assigned to it. An example for an n:m relation would be if several reviews were linked to more than one product (for example, because a book was published by several publishers its license and the reviews are valid for every one of these editions).

1:1 Relations

One-to-one relation is a direct relationship between one object and another. It is represented by attributes that are ComposedTypes. The example shows a relation between User and UserProfile:

```
<itemtype code="User" extends="Principal" jaloclass="de.hybris.platform.jalo.user.User" autocreate="true">
<attributes>
    ...
    <attribute autocreate="true" qualifier="userprofile" type="UserProfile">
        <modifiers read="true" write="true" partof="true"/>
        <persistence type="property"/>
    </attribute>
    ...
</attributes>
</itemtype>

<itemtype code="UserProfile" extends="GenericItem" jaloclass="de.hybris.platform.hmc.jalo.UserProfile" autocreate="true">
<attributes>
    <attribute autocreate="true" qualifier="owner" type="Principal" redeclare="true">
        <persistence type="cmp" qualifier="ownerPkString"/>
        <modifiers read="true" write="false" search="true" optional="true" private="false" initial="true"/>
    </attribute>
</attributes>
</itemtype>
```

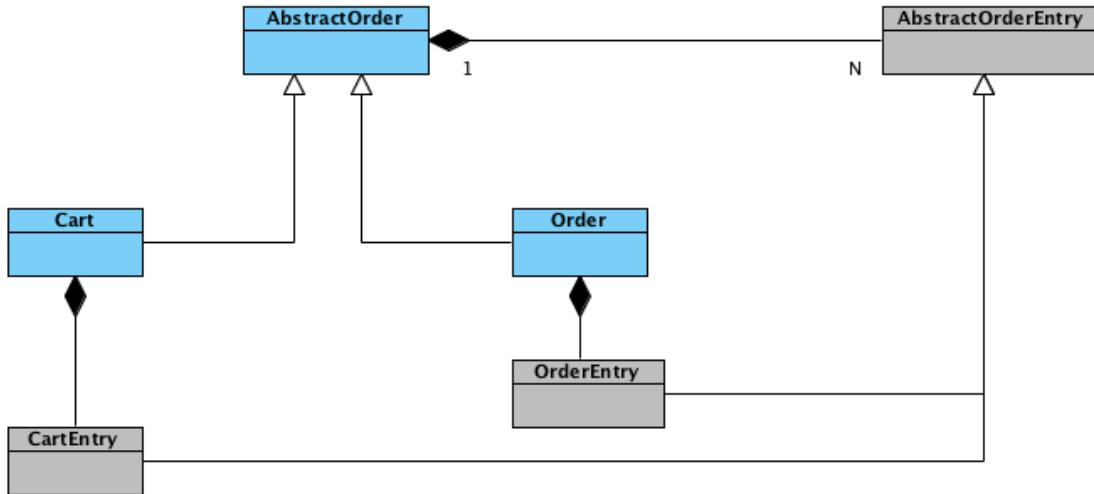
1:n Relations - One-to-Many vs Many-to-One

Depending on the way the relation is stored, the discussion will be about **one-to-many** or **many-to-one** relations. A **many-to-one** relation has its entries stored with the items themselves, while the **one-to-many** relation stores the entries by itself.

An example for **one-to-many** relations would be the Country type and the Regions it stores. Such relations are commonly modeled via a CollectionType. The advantage of a CollectionType-based **one-to-many** relation is that it is quite easy to implement, and that reading and writing CollectionTypes is a fast operation. However, there are two reasons why we do not recommend using CollectionTypes: first, they do not scale very well - the more values such a relation has, the more database search performance decreases. Second, as a database field only has a limited length, values longer than that length get cut off and are lost. For more technical details on CollectionTypes, have a look at the [reference section](#).

Redeclaring 1-n Relations

Sometimes it is useful to define a relation between two abstract types, and make the relation concrete in subclasses.



As you can see, there is a relation defined between **AbstractOrder** and **AbstractOrderEntry**. You can conveniently base your business code on these abstract classes, and reuse the code for concrete subclasses.

However, at runtime we deal with concrete relations:

- Order to OrderEntries
- Cart to CartEntries

The problem is, when querying for entries of **Order**, **CartEntries** table are queried, too, even though we know that only **OrderEntries** should be taken into account. The same issue exists for **Cart**, namely that the `getEntries()` method would also look into the **OrderEntries** table, although we know that only **CartEntries** should be queried.

In SAP Commerce, this performance problem is gone, because you can define a relation on an abstract level (between two abstract classes), and then redeclare this relation in concrete classes to point to concrete subclasses.

1. Define a relation between two abstract classes:

```

<relation code="AbstractOrder2AbstractOrderEntry" localized="false" generate="true" autocreate="true">
    <sourceElement type="Abstract"
        <modifiers read="true" write="true"/>
        <custom-properties>
            <property name="ordering.attr">
                <value>entryNumber</value>
            </property>
        </custom-properties>
    </sourceElement>
    <targetElement type="Abstract"
        <modifiers read="true" write="true"/>
    </targetElement>
</relation>
  
```

2. Then, at the relation-end we can redeclare the definition of **Order** item such that it contains only **OrderEntries**:

```

<itemtype code="Order"
    extends="AbstractOrder"
    jaloclass="de.hybris.platform
    autocreate="true"
    generate="true">
    <deployment table="Orders" ty>
        <attributes>
            <attribute autocreate="true">
        </attributes>
    </deployment>
</itemtype>
  
```

3. Additionally, the following collection type is needed:

```
<collectiontype code="OrderEntryCollection" elementtype="OrderEntry" autocreate="true" genera
```

4. The corresponding change should also be done on the other side of relation, in OrderEntry:

```
<itemtype code="OrderEntry"
          extends="AbstractOrderEntry"
          jaloClass="de.hybris.platform
          autocreate="true"
          generate="true">
<deployment table="OrderEntri
<attributes>
<attribute autocreate="true"
<modifiers read="true" write=
</attribute>
</attributes>
</itemtype>
```

This solution strikes a very good balance between flexibility and reuse on the code level, without sacrificing performance at the ORM layer. It ensures:

- Flexibility on code-level: In java, you can use abstract base classes in your business logic, not relying on concrete implementations, so you can reuse the same logic for, say, Order and Cart.
- Performance on persistence-level: When retrieving Order from the database, and then OrderEntries, SAP Commerce will query only the table containing OrderEntries. Other tables containing, for example, CartEntries are not included in the query.

Custom Ordering

SAP Commerce also provides a custom property ordering.attribute defined for AbstractOrder2AbstractOrderEntry relation.

By defining this property, it is possible to specify which attribute will be used to order the many-side items when retrieving from the database. In the example above, we defined the many-side as ordered=false, and specified a custom ordering attribute.

The SAP Commerce Service Layer takes care of putting OrderEntries in the correct order by setting the entryNumber, so there is no need for the ORM to add an additional ordering column, therefore the many side is ordered=false.

However, entries retrieved from the database should be ordered according to the entryNumber computed upon save, and this can now be achieved using the ordering.attribute property.

Tuning Ordered One-to-Many Relations

One of the relations you can define in Platform is the one-to-many relation. You can configure it as ordered. With an ordered one-to-many relation, an ordering attribute is used to order the items of the **-many** side when they're retrieved from the database. When you create a new instance of the item of the **-many** side, a new value is assigned to this attribute only when it isn't already provided. The new value is a result of the current highest value of this attribute retrieved through an SQL statement, plus 1 (one). The operation is handled by OneToManyHandler.

You can adjust this behavior using the following property:

```
relation.<relationName>.reordered=
```

Here is a use example:

```
relation.Product2FeatureRelation.reordered=sync
```

The property can have the following values:

- **false** is the setting where the SQL statement that retrieves the current highest value of an ordering attribute is always run
- **sync** means that only when a **sync job** is active, the SQL statement isn't run and no value for the ordering attribute (in case of **ProductFeature** it's the **featurePosition** attribute) is provided whenever a new instance of an item of the **-many** side is created (for example **ProductFeature**)
- **always** means that the SQL statement is never executed and no value for the ordering attribute (in case of **ProductFeature** it is the **featurePosition** attribute) is provided whenever a new instance of an item of the **-many** side is created (**ProductFeature**)

If this property isn't set for a given relation, the query isn't run when the value of the ordering attribute is provided. If the value isn't known, the query is run.

By default, the value of this property is set to **sync** for the **Product2FeatureRelation** relation.

Condition Query

The **condition.query** custom property is defined for the **User2Addresses** relation. The property holds a string that is later added to the **where** part of the select query generated for a one-to-many or many-to-one relation. The condition query is written in **FlexibleSearch** and must be valid in the context of a given relation query. The condition query shouldn't contain any order by part as it is added at the end of a generated query. The condition query custom property can only be defined in one end of the relation and only for relations of a one-to-many or many-to-one type. It must be defined in either **sourceElement** or **targetElement** that have the **many** cardinality.

```
<relation code="User2Addresses" generate="true" localized="false" autocreate="true">
  <sourceElement type="User" cardinality="one" qualifier="owner">
    <modifiers read="true" write="true" search="true" optional="true" />
  </sourceElement>
  <targetElement type="Address" cardinality="many" qualifier="address">
    <modifiers read="true" write="true" search="true" optional="true" />
    <custom-properties>
      <property name="condition.query">
        <value>{original} is null</value>
      </property>
    </custom-properties>
  </targetElement>
</relation>
```

In the example above, the User has many addresses, but only those that have the **original** property set to null.

i Note

Possible Orphans by Incorrect Use

There is a possibility to save an item that has a foreign key set to its owner. It will be a part of the relation despite the fact that the condition query hasn't been met. If the **many** element of the relation has the **partOf** property set to true and a given item doesn't meet the condition query, then this item won't be removed after the removal of the item model. In this way, there is a possibility to leave an orphan element in the persisted model.

n:m Relations

If you expect your type to contain many values, we recommend using a **RelationType** instead. This one stores the values not with the type itself, but with the items the type refers to and runs a search over those item values. Such a **many to one** relation scales better than a **CollectionType**.

Relation Caching

Relation caching allows the system to store results of FlexibleSearch queries containing collections of related user and user group items in a dedicated cache region called a relation cache. Results stored in this cache aren't subject to various processes that can slow down the retrieval of other FlexibleSearch results.

The relation cache improves the performance of storing and retrieving query results with relations of the `PrincipalGroupRelation` type containing related items as it doesn't validate queries using a modification counter. The counter represents the number of times any items of a given type are modified, for example when they're created or removed. Every query result stored in the FlexibleSearch cache always contains counter values that are present at the moment of running this query. When the system tries to fetch results stored in the FlexibleSearch cache, it compares the most recent counter values with the ones that are cached. If they don't match, the cached result is treated as invalid and the query is run in the database. Query results are invalidated whenever any items of a given type are modified, so modifying an arbitrary item leads to the system invalidating queries related to all the other items of the same type, which isn't always necessary. The relation cache doesn't use the modification counter and, as a result, prevents the system from running unnecessary processes that could decrease the performance of SAP Commerce.

Relation caching ensures that creating new user accounts through the SAP Commerce storefront doesn't invalidate cached query results containing relations between other user and user group items that aren't related to the new users.

Invalidation in Relation Cache

To track changes that are made in related items in the context of other items, additional information is added to invalidation events produced by the persistence layer, for example primary keys of related items before and after a change. The information allows relation caching to invalidate entries on both sides of the relation in case of any modifications.

In some cases in a clustered environment, invalidation events might not include any additional data, for example when relation caching is introduced during a rolling update or when a customized logic is responsible for producing such events. In these situations, all cached data is invalidated for a given relation to prevent the relation cache from being polluted with old entries. Results stored in the FlexibleSearch cache are invalidated in the same way.

Relation Cache Metrics

To see relation cache statistics, log in to the SAP Commerce Administration Console, go to the **Monitoring > Cache** tab, and select the `relationCacheRegion` region.

The screenshot shows the SAP Commerce Administration Console interface for monitoring cache metrics. The main area displays a tree view of cache regions, with the `relationCacheRegion` node selected and highlighted in blue. Below the tree, detailed statistics for the `relationCacheRegion` are shown in a table. At the bottom, there is a table listing specific entries with their metrics.

Type	Hits	Misses	Ratio	Invalidations	Evictions	Fetches	InstanceCount
<code>PrincipalGroupRelation.groups</code>	0	0	0	12	0	0	0

At the bottom right, there are navigation buttons for 'Previous' (with page 1 highlighted), 'Next', and a search bar.

[Configuring Relation Caching](#)

Configure relation caching using properties.

[Custom Implementation of Relation Caching](#)

You can customize the relation cache so that it stores items of types different than `PrincipalGroupRelation`.

Configuring Relation Caching

Configure relation caching using properties.

The relation cache is a separate `CacheRegion` that is defined in the `core` extension. It is enabled globally by default. You can configure relation caching using the following properties:

Property	Default Value	Description
<code>relation.cache.enabled=</code>	<code>true</code>	Enables or disables relation caching for all relations.
<code>relation.cache.default.capacity=</code>	<code>10000</code>	Defines the number of items stored by the relation cache. The property is applied to both sides of cached relations. As a result, the maximum cache size is equal to the doubled value of this property.
<code>relation.cache.<RelationTypeCode>.enabled=</code>	<code>false</code>	Enables or disables relation caching for selected relations. Before configuring this property, ensure that you have implemented relation caching for the selected relations as described in Custom Implementation of Relation Caching . Replace <code><RelationTypeCode></code> with the name of a relation as defined in the relevant <code>items.xml</code> file.
<code>relation.cache.<RelationTypeCode>.capacity=</code>	Not applicable	Defines the number of items stored by the relation cache. Replace <code><RelationTypeCode></code> with the name of a relation as defined in the relevant <code>items.xml</code> file. The property is applied to both sides of cached relations. As a result, the maximum cache size is equal to the doubled value of this property.

In addition to storing results of `FlexibleSearch` queries containing relations of the `PrincipalGroupRelation` type, relation caching also stores results obtained by calling the `Principal.getGroups()` and the `PrincipalGroup.getMembers()` methods. Use the following properties to configure relation caching for these methods:

Property	Default Value	Description
relation.cache.PrincipalGroupRelation.enabled=	true	Enables or disables relation caching for PrincipalGroupRelation methods.
relation.cache.PrincipalGroupRelation.capacity=	50000	Defines the number of results stored by the relation cache for PrincipalGroupRelation methods.

Custom Implementation of Relation Caching

You can customize the relation cache so that it stores items of types different than PrincipalGroupRelation.

If you want to extend the scope of relation caching, modify the way relation accessors work.

For example, in Principal and PrincipalGroup, check whether the relation cache is enabled for a given type of relations. See `de.hybris.platform.jalo.security.Principal.getGroups` as an example:

```
public Set<PrincipalGroup> getGroups(final SessionContext ctx)
{
    if(isPrincipalGroupRelationCachingEnabled())
    {
        return getFromRelationCache(ctx,CoreConstants.Relations.PRINCIPALGROUPRELATION,Principal.G
    }
    else
    {
        // basic logic to retrieve linked items
    }
}
```

In the example, the `isPrincipalGroupRelationCachingEnabled()` method checks whether relation caching is enabled for the PrincipalGroup relation. See the full method:

```
boolean isPrincipalGroupRelationCachingEnabled()
{
    return RelationsCache.getDefaultInstance().getSingleCacheableAttributes(TypeId.fromTypeName(Co
}
```

If relation caching is enabled, the system calls `getFromRelationCache()`:

```
<T extends Principal> Set<T> getFromRelationCache(final SessionContext ctx, final String relation,
{
    final List<PK> pks = RelationCacheUnit.createRelationCacheUnit(relation, manySide, ownerPk).ge
    return new HashSet(JaloSession.lookupItems(ctx, pks, true, false));
}
```

It's possible to use the API provided by `de.hybris.platform.cache.relation.RelationCacheUnit` only when caching is enabled for a given relation.

To extend relation caching, create a `CacheUnit` object with the relevant arguments and call the `getCachedPKs()` method to retrieve lists of primary keys related to parent items. As a last step, translate these keys to `JaloSession.lookupItems` items.

Relation Caching and Search Restrictions in FlexibleSearch

You can't use relation caching for relations with items that can't be accessed due to FlexibleSearch restrictions as the relation cache is unable to efficiently invalidate queries containing items with such restrictions. For more information on restrictions, see [Restrictions](#).

i Note

It isn't possible to use any search restrictions for items of the `PrincipalGroupRelation` type.

Managers

Managers handle any sort of action on items that items cannot perform themselves. Also, if you need to have a method for several types that are not related to one another, the Manager is a good place to put that method.

Managers handle creation, deletion, and searching for items. They also deal with searching for and deleting items. If you need to implement a method that handles items from the outside, putting that method into the Manager is a good idea.

A Manager is a Singleton - that is, there is always just one single instance of a certain Manager type, and you won't be able to create more than this single instance.

In consequence, if you want the Manager to do something for you (creating an item, for example), you cannot create a Manager instance in your code, but you need to get hold of the single existing Manager instance as in the following code snippets, for example:

```
// find the Job instance with the qualifier myJobCode
final Job exportJob = CronJobManager.getInstance().getJob( "myJobCode" );

// create various access rights
final UserRight read    = AccessManager.getInstance().getOrCreateUserRightBy
final UserRight create  = AccessManager.getInstance().getOrCreateUserRightBy
final UserRight remove  = AccessManager.getInstance().getOrCreateUserRightBy
final UserRight change  = AccessManager.getInstance().getOrCreateUserRightBy
```

Also note that the Manager class of your extension is where you need to implement / override getter and setter methods for attributes added to SAP Commerce types.

Across a multi-tenant SAP Commerce installation, every tenant has individual managers. In other words, the tenants **master** and **junit** have different instances of `ProductManager`, but within each individual tenant, the `ProductManager` is always the same instance. It is not possible to get hold of a manager instance of another tenant. Within the **junit** tenant, for example, you cannot access any Manager of the **master** tenant, and vice versa.

References

See the reference information for `ItemType` and `AttributeDescriptor` modifiers, as well as type creation examples.

References

`ItemType` Modifiers

Modifier	Description / Comment
<code>code</code>	The identifier of this <code>ItemType</code>

Modifier	Description / Comment
extends	The superclass of this ItemType
jaloclass	The fully qualified classpath of this ItemType
autocreate	If set to <code>true</code> , a new ItemType gets created when the Platform creates the type system during initialization. Set this to <code>false</code> if you are adding to an existing type that is defined elsewhere.
generate	If set to <code>true</code> , the Platform creates jalo classes.

AttributeDescriptor Modifiers

Modifier	Description / Comment
qualifier	The identifier of this Attribute
autocreate	If set to <code>true</code> , a new Attribute gets created when the Platform creates the type system during initialization. Set this to <code>false</code> if you are adding to an existing attribute that is defined elsewhere.
type	The identifier of the type this attribute is related to

Creating a Relation via the TypeManager

A simple relation type example:

```
TypeManager tm = jaloSession.getTypeManager()
ComposedType productType = tm.getComposedType( Product.class );

        // first create the relation end point attributes
CollectionType productColl = tm.createCollectionType( "followUpProd
AttributeDescriptor followUpsAttr = productType.createAttributeDes
AttributeDescriptor.READ_FLAG + AttributeDescriptor.WRITE_FLAG + A
AttributeDescriptor.SEARCH_FLAG );
AttributeDescriptor usedAsFollowUpForAttr = productType.createAttr
productColl, AttributeDescriptor.READ_FLAG +
AttributeDescriptor.WRITE_FLAG + AttributeDescriptor.OPTIONAL_FLAG

RelationType followUpRelation = tm.createRelationType(
"FollowUpRelation",           // the relation code
false,                         // not localized
followUpsAttr,                // the relation end point attr belonging
// to the source type ( this collection is sorted ! )
usedAsFollowUpForAttr // the relation end point attr belonging to t
);
```

Values can be read and written by different ways. Either the jalo API methods `Item.getLinkedItems(...)` and `Item.setLinkedItems(...)` or the generic `Item.getAttribute(...)`.

```
Product myProduct = ...
Collection followUps = myProduct.getLinkedItems(
true,                      // here the item is considered to be the source
"FollowUpRelation" // the relation code
null                      // no language needed since this relation is un
);

Collection usedAsFollowUpFor = myProduct.getLinkedItems(
false,                     // now the item is considered to be the target
"FollowUpRelation" // the relation code
null                      // no language needed since this relation is un
);
```

```
// slightly easier the generic access
Collection followUps = (Collection)myProduct.getAttribute( "followUpFor" );
Collection usedAsFollowUpFor = (Collection)myProduct.getAttribute( "usedAsFollowUpFor" );
```

Please note that it is not required to implement a AttributeAccess for relation attributes.

Creating an EnumType

```
ComposedType myEnumValueType = tm.getComposedType( "MySpecialEnumValueType" );
EnumerationType enum = jaloSession.getEnumerationManager().createEnumeration(
    "MySpecialEnumValueType", myEnumValueType );
```

Creating a MapType

```
TypeManager tm = jaloSession.getTypeManager(); // create a map type for localized attribute MapType
localizedDateType = tm.createMapType(
    "locDateType",
    tm.getComposedType( Language.class ),
    tm.getAtomicType( Date.class ) );
// append a attribute descriptor to a item type
ComposedType userType = tm.getComposedType( User.class );
userType.createAttributeDescriptor( "locDateField", locDateType, A1,
AttributeDescriptor.WRITE_FLAG + AttributeDescriptor.OPTIONAL_FLAG );
```

Additional Information

Find out more about type systems across multi-tenant installations and orphaned types.

Type Systems across Multi-tenant Installations

A SAP Commerce installation running a multi-tenant system can have an individual type system for each individual tenant, depending on the tenant's configured extensions.

Orphaned Types

Running an SAP Commerce update might result in having type definitions in the database that are not backed by an `items.xml` file definition. Such types are referred to as orphaned types.

For details, refer to [Initializing and Updating SAP Commerce](#), section **Orphaned Types**.

Creating Sample Data using ImpEx Extension

The ImpEx extension allows you to import and export Platform items into a CSV file. Refer to [Using ImpEx with Backoffice or SAP Commerce Administration Console](#) for details.

Related Information

[ServiceLayer](#)

[Models](#)

[Working with Enumerations](#)

[Product Content and Catalogs](#)

[Product Modeling](#)

Using Encryption for Attribute Values

The point of encryption is to store sensitive data (such as passwords) in a way that it is not easily readable. SAP Commerce contains an encryption mechanism for the data stored in the SAP Commerce database. Instead of writing values to the database in plain text, the SAP Commerce encryption mechanism saves an encrypted representation of the value.

This encryption mechanism is not intended for user authentication or other functionality with SAP Commerce directly. The mechanism encrypts attribute values on-the-fly when writing to the database and decrypts on-the-fly when reading from the database. The intention of encrypted values in the database is to block sensitive data when accessing the database. By writing encrypted attribute values only, the encryption mechanism makes it much harder for a database attacker to read out data - no matter whether the attack is run against a running database or whether the attacker has a database dump.

Specifying Encryption Mechanism

SAP Commerce uses one encryption mechanism at a time for all attributes to be encrypted. You need to specify the encryption mechanism in the `project.properties` or `local.properties` file. Please refer to [Configuring the Behavior of SAP Commerce](#) for additional information on the `local.properties` file.

To specify encryption parameters, use these properties:

```
encryption.provider.signature=BC
encryption.provider.class=org.bouncycastle.jce.provider.BouncyCastleProvider

symmetric.algorithm=PBEWITHSHA-256AND256BITAES-CBC-BC
symmetric.key.file=256bit-symmetric.key
```

Property name	Description
<code>encryption.algorithm</code>	Specifies the algorithm used to encrypt attribute values. The possible values are determined by the value of <code>encryption.provider.class</code> .
<code>encryption.provider.class</code>	The fully qualified classpath of the Java class to handle the encryption. Determines the possible values for <code>encryption.algorithm</code> .

Setting Encryption for Attributes

To activate encryption for data storage of an attribute, you need to specify the `encrypted` modifier in the attribute definition in your extension's `items.xml` file, such as:

```
<attribute qualifier="number" autocreate="true" type="java.lang.String">
  <persistence type="property"/>
  <modifiers optional="false" encrypted="true"/>
</attribute>
```

Jalo-only Attributes

SAP Commerce allows for a non-persistent kind of attribute referred to as a jalo attribute or jalo-only attribute.

i Note

Jalo attributes are deprecated and replaced by [Dynamic Attributes](#).

Overview of Jalo Attributes

Jalo attributes have non-persistent values, and are defined in a `<persistence type="jalo">` tag in the `items.xml` file. Persistent attributes, by contrast, are defined in a `<persistence type="property">` tag.

Unlike persistent attributes, the values of jalo-only attributes are held in memory and not written to the SAP Commerce database. The values of jalo-only attributes exist only during runtime. If SAP Commerce is shut down or terminated, the values of jalo-only attributes will be lost irretrievably. In other words, use jalo-only attributes only for values that do not matter if lost, or for values which you can restore from another source. You cannot use Flexible Search on Jalo attributes.

Defining a Jalo-only Attribute

Define a jalo-only attribute by specifying that the attribute is jalo-only, as previously described, and by implementing getter and setter methods for the attribute values. If you add jalo-only attribute to an existing type definition, getters and setters are generated in the Manager class of your extension. This way of adding attributes to types is not recommended. For details see [items.xml](#).

Persistence Definition

To specify a jalo-only attribute, assign the value `jalo` to the attribute's `<persistence>` tag in the `items.xml` file, as follows:

```
<attribute qualifier="myJaloAttribute" type="java.lang.Object">
    <persistence type="jalo"/>
</attribute>
```

Implementing Getter and Setter Methods

Unlike persistent attributes, jalo-only attributes do not use the SAP Commerce persistence layer. By consequence, SAP Commerce must rely on a customized attribute saving mechanism. This means that you will need to implement the getter and setter methods for the attribute yourself.

If you try to build an extension whose `items.xml` file specifies jalo-only attributes, the extension will fail to build if you have not implemented getter and setter methods for the jalo-only attribute.

```
[yjavac] Compiling 5 source files to /opt/hybris/bin/extensions/training/classes
[yjavac] -----
[yjavac] 1. ERROR in /opt/hybris/bin/extensions/training/src/org/training/jalo/MyPro
[yjavac]     public class MyProduct extends GeneratedMyProduct
[yjavac]           ^
[yjavac] The type MyProduct must implement the inherited abstract method GeneratedM
[yjavac] -----
[yjavac] 2. ERROR in /opt/hybris/bin/extensions/training/src/org/training/jalo/MyPro
[yjavac]     public class MyProduct extends GeneratedMyProduct
[yjavac]           ^
[yjavac] The type MyProduct must implement the inherited abstract method GeneratedM
[yjavac] -----
[yjavac] 3. ERROR in /opt/hybris/bin/extensions/training/src/org/training/jalo/MyPro
[yjavac]     public class MyProduct extends GeneratedMyProduct
[yjavac]           ^
[yjavac] The type MyProduct must implement the inherited abstract method GeneratedM
[yjavac] -----
[yjavac] 4. ERROR in /opt/hybris/bin/extensions/training/src/org/training/jalo/MyPro
[yjavac]     public class MyProduct extends GeneratedMyProduct
[yjavac]           ^
[yjavac] The type MyProduct must implement the inherited abstract method GeneratedM
```

```
[yjavac] -----
[yjavac] 5. ERROR in /opt/hybris/bin/extensions/training/src/org/training/jalo/MyPro
[yjavac]     public class MyProduct extends GeneratedMyProduct
[yjavac]           ^
[yjavac] The type MyProduct must implement the inherited abstract method GeneratedMy
[yjavac] -----
[yjavac] 6. ERROR in /opt/hybris/bin/extensions/training/src/org/training/jalo/MyPro
[yjavac]     public class MyProduct extends GeneratedMyProduct
[yjavac]           ^
[yjavac] The type MyProduct must implement the inherited abstract method GeneratedMy
[yjavac] -----
[yjavac] 6 problems (6 errors)

BUILD FAILED
/opt/hybris/bin/platform/build.xml:23: The following error occurred while executing
/opt/hybris/bin/platform/resources/ant/antmacros.xml:366: The following error occuri
/opt/hybris/bin/platform/resources/ant/antmacros.xml:385: The following error occuri
/opt/hybris/bin/platform/resources/ant/util.xml:21: The following error occurred whi
/opt/hybris/bin/platform/resources/ant/antmacros.xml:387: The following error occuri
/opt/hybris/bin/platform/resources/ant/antmacros.xml:472: The following error occuri
/opt/hybris/bin/platform/resources/ant/antmacros.xml:570: The following error occuri
/opt/hybris/bin/platform/resources/ant/antmacros.xml:591: The following error occuri
/opt/hybris/bin/platform/resources/ant/util.xml:123: Compile failed; see the compile
```

When the extension generates the class files for the type system during the build phase, it creates an abstract and a non-abstract class for the type, for example `MyProduct.java`.

Specifying the `jalo` value for an attribute causes SAP Commerce to skip generating the getter and setter methods for this attribute. In other words: if you create a `jalo` attribute and do not implement getter and setter methods for that attribute, there will be no getter and setter methods at all.

However, neither the abstract nor the non-abstract class definition files contain an implementation of those methods at first. The abstract class file contains abstract getter and setter methods only, and the non-abstract class file will be empty. Therefore, there are no getter and setter methods and the build will fail. You need to implement the getter and setter methods in the non-abstract class file for the build to work.

Sample Use Case

The `jalo`-only attribute `mirroredAddress` maps to the value of the user's `defaultPaymentAddress` attribute. On the getter method side, the `mirroredAddress` attribute reads out and displays the value of the `defaultDeliveryAddress` attribute. On the setter method side, the `mirroredAddress` attribute sets the value for the `defaultDeliveryAddress` attribute.

For details on how to integrate an extensions, see [Creating a New Extension](#). The following code sample shows an `items.xml` file that defines the `mirroredAddress` attribute by extending the `User` type and adding the `mirroredAddress` attribute definition with the type set to `jalo`.

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

  <itemtypes>
    <itemtype generate="false"
              code="User"
              autocreate="false" >
      <attributes>
        <attribute qualifier="mirroredAddress" type="Address" >
          <persistence type="jalo" />
        </attribute>
      </attributes>
    </itemtype>
  </itemtypes>
</items>
```

To avoid a build failure for the extension, implement a getter and a setter method for the `mirroredAddress` attribute in the extension's Manager class, such as the following:

```

@Override
    public Address getMirroredAddress(SessionContext ctx, User item) {
        return item.getDefaultDeliveryAddress();
    }

@Override
public void setMirroredAddress(SessionContext ctx, User item, Address value) {
    item.setDefaultDeliveryAddress( value );
}

```

Because the `User` type is not created in the `JaloTest` extension, the getter and setter methods for the `mirroredAddress` attribute need to be implemented in the `JaloTestManager` class. If the `User` type was created in the `JaloTest` extension, then the getter and setter methods would need to be implemented in the Java file for the `User` type. For more details, see [The Type System](#).

Sample Screenshots

The following screenshots show examples of the `mirroredAddress` attribute. By setting the value for the `mirroredAddress`, the value for the `defaultDeliveryAddress` is also set. However, be aware that only the value stored in the `defaultDeliveryAddress` attribute is stored in the database. The values for the `mirroredAddress` attribute are not persistent and will be lost if SAP Commerce shuts down.

The screenshot shows the 'Addresses' tab with two address entries:

	Postal Code	Town	Street Name
<input type="checkbox"/>	12345	n/a	Jalo Road
<input type="checkbox"/>	23456	n/a	Payment Avenue

Below the table, there are dropdown menus for 'Default Payment Address' and 'Default Shipment Address'. The 'Default Payment Address' dropdown contains 'Payment Avenue - 12 - 23456 - n/a'. The 'Default Shipment Address' dropdown contains 'Jalo Road - 5 - 12345 - n/a'. A cursor arrow is pointing towards the 'Default Shipment Address' dropdown.

The Address Tab

The screenshot shows the 'Unbound' section with the following fields:

- Base DN:
- [mirroredAddress]: Jalo Road - 5 - 12345 - n/a
- [orderProcess]:

	Code	Job	Current status
The list is empty.			

The Administration Tab of the Unbound Section

Related Information

[The Type System](#)

[items.xml](#)

[Jalo Layer](#)

items.xml

The `items.xml` file specifies types of an extension. By editing the `items.xml` file, you can define new types or extend existing types. In addition, you can define, override, and extend attributes in the same way.

Elements allowed within the file are available in the [items.xml Element Reference](#).

→ Tip

XML Editor Recommended

If you create XML files based on the `items.xsd` file, you can create valid `items.xml` files easily. The IDE Eclipse, for example, contains such XML editors. They not only make the content of `items.xml` files valid, but also show description of the XML elements from the XSD file.

i Note

Both ServiceLayer And Jalo Layer Covered in This Document

This document discusses both principal API layers of SAP Commerce: ServiceLayer and Jalo Layer. Because the Jalo Layer is closely related to the type system (much more closely than the ServiceLayer), there are many connections between the `items.xml` file and the Jalo Layer.

Although the API layer of choice for SAP Commerce is the ServiceLayer, this document needs to discuss the `items.xml` file with references to the Jalo Layer. Parts of the document that are primarily or exclusively related to the ServiceLayer are marked with a *ServiceLayer* label, whereas parts that are primarily or exclusively related to the Jalo Layer are marked with a *Jalo Layer* label. Parts not marked explicitly refer to both the ServiceLayer and the Jalo Layer.

Location

The `items.xml` is located in the `resources` directory of an extension. The `items.xml` files are prefixed with the name of their respective extension in the form of `extension-name-items.xml`. For example:

- For the `core` extension, the file is called `core-items.xml`.
- For the `catalog` extension, the file is called `catalog-items.xml`.

Structure

The `items.xml` defines the types for an extension in XML format.

Basic Structure

The basic structure of an `items.xml` file is as follows:

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

    <atomicatypes>
        ...
    </atomicatypes>

    <collectiontypes>
        ...
    </collectiontypes>
```

```

<enumtypes>
...
</enumtypes>

<maptypes>
...
</maptypes>

<relations>
...
</relations>

<itemtypes>
...
</itemtypes>

```

</items>

As the `items.xml` file is validated against an XSD file (`items.xsd`), the order of type definitions must conform to this order. For a discussion of the kinds of type, refer to the [Type System Documentation](#).

A type definition order that doesn't conform to the `items.xsd` causes SAP Commerce to fail the extension build.

Build failure message:

```

[echo] building extension 'myextension'...
[yxmlschemavalidator]
[yxmlschemavalidator] ERROR(S): [file:///C:/hybris/trunk/bin/myextension/resources/myextension-iter
[yxmlschemavalidator]
[yxmlschemavalidator] line:24,column:15 : cvc-complex-type.2.4.d: Invalid content was found startin
[yxmlschemavalidator]
[yxmlschemavalidator] ERROR(S): [file:///C:/hybris/trunk/bin/myextension/resources/myextension-iter

```

Defining Types Through `items.xml`

To specify a SAP Commerce type, define the type's attributes and details, such as code or description, in the XML format. See the following example:

```

<itemtype
  code="Publication"
  jaloclass="de.hybris.platform.print.jalo.Publication"
  extends="GenericItem"
  generate="true"
  autocreate="true">
  <deployment table="Publications" typecode="23402"/>
  <attributes>
    <attribute qualifier="code" type="java.lang.String">
      <modifiers optional="false" />
      <persistence type="property"/>
    </attribute>
    <attribute qualifier="sourceCatalogVersion" type="CatalogVersion">
      <persistence type="property"/>
    </attribute>
    <attribute qualifier="rootChapters" type="ChapterCollection">
      <modifiers write="false" search="false"/>
      <persistence type="jalo"/>
    </attribute>
  </attributes>
</itemtype>

```

i Note

Out-of-Order Type Definition Not Supported

The `items.xml` file is parsed and evaluated in running order in one single pass. SAP Commerce doesn't allow multipass processing of the `items.xml` file (like the ImpEx framework does, for example), which would allow defining types in any order.

This means that you need to define types in order of inheritance. More abstract types need to be defined more to the beginning of the `items.xml` file and more concrete types need to be defined more to the end. For example, the following itemtype definition would fail due to being out of order:

```
<itemtype code="SpecialMyType"
    extends="MyType"
    autocreate="true"
    generate="true" >
</itemtype>

<itemtype code="MyType"
    extends="Product"
    autocreate="true"
    generate="true" >
</itemtype>
```

Validation

During an SAP Commerce build, the build process makes sure that every extension's `/resources` directory contains a copy of a main XSD file (`items.xsd`). This main file allows SAP Commerce to validate the extension's `items.xml` file against the main `items.xsd` file. The build process also makes sure that the `items.xml` file is well-formed XML and doesn't contain errors or invalid parts, for example incorrectly defined attributes. If this check fails, SAP Commerce causes the extension build to fail. In other words, if your extension's `items.xml` doesn't conform to the `items.xsd`, you aren't able to get the extension to compile. This prevents you from integrating a broken type system definition.

→ Tip

XML Editor Recommended

If you create XML files based on the `items.xsd` file, you can create valid `items.xml` files easily. The IDE Eclipse, for example, contains such XML editors. Not only are your `items.xml` files content valid, but Eclipse also shows description of XML elements from the XSD file.

→ Tip

Changes in the `items.xml` file take effect automatically using Eclipse.

SAP Commerce comes with preconfigured builders for the [Eclipse IDE](#) that support working with the `items.xml` file. Using Eclipse, whenever you edit an `items.xml` file, SAP Commerce automatically:

- Jalo Layer: Generates `Generated*.java` source files (item classes) for all item types of your extension to the `gensrc` directory of your extension.
- Jalo Layer: Refreshes the `gensrc` directory of your extension.
- ServiceLayer: Generates `*Model.java` source files (model classes) for all item types of configured extensions to the `bootstrap/gensrc` directory
- ServiceLayer: Refreshes the `bootstrap/gensrc` directory

This means that when using Eclipse, changes that you make in the `items.xml` file take effect immediately for your application code. You can immediately use getter and setter methods for newly added types and attributes. Also, model classes are always up-to-date. However, this mechanism doesn't affect SAP Commerce's data model by runtime. You still have to initialize or update SAP Commerce explicitly.

Furthermore, builders don't start the compilation as they only generate source files, so you can immediately use the item/model classes for programming. A build has to be started explicitly.

Adding Types and Attributes

There are two methods to add an attribute to existing types.

- Creating a subtype and adding the attributes to the subtype
- Adding the attribute to the type directly

The following section discusses these two methods and their consequences.

Creating a Subtype and Adding the Attributes to the Subtype

This is the method recommended by SAP as it keeps SAP Commerce's core types untouched. On the Jalo Layer, you also have an individual Java class available where you can implement your business logic.

You reference the type from which to extend, specify a name for the subtype, and add the attribute. For example, the following `items.xml` snippet creates a subtype `MyProduct` extending from `Product` and adds an attribute `oldPrice` of type `java.lang.Double`:

```
<itemtype code="MyProduct" autocreate="true" generate="true" extends="Product" jaloclass="org.trair
    <attributes>
        <attribute qualifier="oldPrice" type="java.lang.Double" generate="true">
            <persistence type="property"/>
            <modifiers read="true" write="true" optional="true"/>
        </attribute>
    </attributes>
</itemtype>
```

In this case, you need to set the value of the `autocreate` element to `true`, which lets SAP Commerce create a new database entry for this type at initialization/update process. Setting the `autocreate` modifier to `false` causes a build failure; the first definition of a type has to enable this flag.

Jalo Layer: Setting the `generate` modifier to `true` results in Java class files being generated for this type (additional details). Setting the `generate` modifier to `false` results in no Java class file being generated for this type. Having no Java class file available means that you aren't able to implement a custom business logic (such as getter and/or setter methods) for the type. You have to use the supertype's business logic implementation.

Adding Attributes to a Type Directly

i Note

This method is discouraged by SAP unless you know the implications and side effects well and you know that you have no alternative to taking this manner.

Where extending a type and using the subtype is complex or not feasible, it's possible to extend the SAP Commerce type directly, such as `Product` or `Customer`:

```
<itemtype code="Product" autocreate="false" generate="false">
    <attributes>
        <attribute qualifier="oldPrice" type="java.lang.Double" generate="true">
            <persistence type="property"/>
            <modifiers read="true" write="true" optional="true"/>
        </attribute>
    </attributes>
</itemtype>
```

This manner isn't recommended by SAP for these reasons:

- You create a direct dependency to an SAP Commerce type.
- **Jalo Layer:** The generated methods for the attributes are written into your extension's manager, but not into the corresponding type class. In essence, this means that you have to address your extension's manager class to set values for these attributes.

As the type basically exists already, you need to set the `autocreate` modifier for the type definition to `false`:

```
<itemtype code="Product" autocreate="false" generate="true">
```

Setting the `autocreate` modifier to `true` results in a build failure.

The value of the `generate` modifier is ignored.

Redeclaring Attributes

You can redeclare an attribute to:

- Change its behaviour. For example, you can add a “unique” flag, or disallow writing.
- Make the type of the attribute more specific for subtypes.

i Note

You can set uniqueness only for supertype attributes. When you add a “unique” flag to an attribute at the subtype level, SAP Commerce ignores it.

Let's take the abstract order item:

```
<itemtype code="AbstractOrder"
          extends="GenericItem"
          jaloClass="de.hybris.platform.jalo.order.AbstractOrder"
          autocreate="true"
          generate="true"
          abstract="true">
    <custom-properties>
        <property name="legacyPersistence">
            <value>java.lang.Boolean.TRUE</value>
        </property>
    </custom-properties>
    <attributes>
        (....)
        <attribute autocreate="true" qualifier="entries" type="AbstractOrderEntryList">
            <persistence type="jalo"/>
            <modifiers read="true" write="true" search="true" partof="true" optional="true"/>
        </attribute>
        (....)
    </attributes>
</itemtype>
```

In the following example, the item cart extends the abstract order item:

```
<itemtype code="Cart"
          extends="AbstractOrder"
          jaloClass="de.hybris.platform.jalo.order.Cart"
          autocreate="true"
          generate="true">
    <deployment table="Carts" typecode="43"/>
    <attributes>
        <attribute autocreate="true" redeclare="true" qualifier="entries" type="CartEntryCollection">
            <modifiers read="true" write="true" search="true" removable="true" optional="true"/>
        </attribute>
        <attribute type="java.lang.String" qualifier="sessionId">
            <persistence type="property"/>
            <modifiers read="true" write="true"/>
        </attribute>
    </attributes>
</itemtype>
```

```
</attributes>
</itemtype>
```

In this example, you can redeclare the type of an item so that it's more specific.

i Note

You can't use types unrelated to their hierarchy.

ServiceLayer

i Note

ServiceLayer-only section

This section only discusses [ServiceLayer](#)-related aspects of the `items.xml` file. The discussion of Jalo-related aspects is located [below](#). For new projects, consider using a ServiceLayer-based approach instead of using the Jalo Layer.

The ServiceLayer facilitates item handling through the use of [Models](#). A model is a [POJO](#)-like representation of an SAP Commerce item. Models have automatically generated getter and setter methods for attribute values.

Using the `items.xml` file in combination with the ServiceLayer is simple:

- You define the data model in the form of types and attributes.
- You call the SAP Commerce's `all` ant target.

The models are then generated automatically and are ready for use.

Null Value Decorators in Models

Null decorator expression is a ServiceLayer feature that allows you to customize the behavior of a getter method. You can specify a code fragment in an item definition (`items.xml`) that computes a result instead of returning null. This assures that the method never returns a null value. We introduced this feature in order to bring ServiceLayer to feature parity with deprecated Jalo-only features that allowed customization of a method body. In the following example Abstract Order has an attribute named `calculated`, and we want to make sure that calling `getCalculated()` never returns null:

```
<attribute autocreate="true" qualifier="calculated" type="java.lang.Boolean" generate="true">
    <custom-properties>
        <property name="modelPrefetchMode">
            <value>java.lang.Boolean.TRUE</value>
        </property>
    </custom-properties>
    <defaultvalue>java.lang.Boolean.FALSE</defaultvalue>
    <persistence type="property"/>
    <modifiers read="true" write="true" search="true" optional="true"/>
    <model>
        <getter default="true" name="calculated">
            <nullDecorator>Boolean.valueOf(false)</nullDecorator>
        </getter>
    </model>
</attribute>
```

You can use the **nullDecorator** tag to specify an expression that is put inside the generated method of the **AbstractOrderModel** class. In this case, we prefer to get **false** instead of **null**. If you generate model classes, you can verify the code of the **getCalculated** method of the **AbstractOrderModel** class:

```
public Boolean getCalculated()
{
    final Boolean value = getPersistenceContext().getPropertyValue(CALCULATED);
    return value != null ? value : Boolean.valueOf(false);
}
```

Because the expression is embedded inside a code, you can be sure that invoking **getCalculated()** on an **AbstractOrderModel** instance never returns null.

Jalo Layer

i Note

Jalo-Only Section

This section only discusses Jalo-related aspects of the **items.xml**. This section addresses developers and technical consultants who are familiar with the Jalo structure.

The API layer of choice to get started and for new projects is the [ServiceLayer](#).

For attributes SAP Commerce optionally generates getter and setter methods automatically.

- Sample attribute definition:

```
<attribute qualifier="oldPrice" type="java.lang.Double" generate="true">
    <persistence type="property"/>
    <modifiers read="true" write="true" optional="true"/>
</attribute>
```

- Getter and setter methods being generated to the corresponding type class:

```
public Double getOldPrice(final SessionContext ctx)
...
public Double getOldPrice()
...
public double getOldPriceAsPrimitive(final SessionContext ctx)
...
public double getOldPriceAsPrimitive()
...
public void setOldPrice(final SessionContext ctx, final Double value)
...
public void setOldPrice(final Double value)
...
public void setOldPrice(final SessionContext ctx, final double value)
...
public void setOldPrice(final double value)
...
```

You can override these method implementations in the nonabstract Java class.

When using the manner of [Adding Attributes to a Type Directly](#), the getter and setter methods for the newly defined attributes are generated into your extension's manager. By consequence, you have to implement custom getter and setter method logic in your extension's manager. For a new implementation, try to avoid this approach and use a [ServiceLayer](#)-based approach instead.

As the SAP Commerce is delivered as a precompiled binary release without source code, adding getter and setter methods to a SAP Commerce class directly isn't possible. (For details see [Type System Documentation](#).) By consequence, you have to implement the getter and setter methods in your extension's **Manager** class. However, regardless of the actual location into which getter and setter methods for attribute values are generated, the mechanism follows the same basic rules.

Setting the **generate** modifier on an attribute definition to **false** results in no getter and setter method being generated whatsoever. In other words: Setting **generate** to **false** results in no automatically generated getter and setter methods:

```
<attribute type="java.lang.String" qualifier="myAttribute" generate="false">
```

Setting the **generate** modifier on an attribute definition to **true** results in getter and setter method being generated, depending on the values of the **modifiers** tag on the attribute definition.

- Setting the **read** modifier to **true** results in a getter method being generated for this attribute:

```
<attribute type="java.lang.String" qualifier="myAttribute" generate="true">
    <modifiers read="true" />
</attribute>
```

- Setting the **read** modifier to **false** results in no getter method being generated for this attribute:

```
<attribute type="java.lang.String" qualifier="myAttribute" generate="true">
    <modifiers read="false" />
</attribute>
```

i Note

Attribute value unreadable

If the getter method isn't generated, there's no way of reading the attribute value. The `getAttribute(...)` method internally also relies on the generated getter and isn't operable either.

- Setting the **write** modifier to **true** results in a setter method being generated for this attribute:

```
<attribute type="java.lang.String" qualifier="myAttribute" generate="true">
    <modifiers write="true" />
</attribute>
```

- Setting the **write** modifier to **false** results in no setter method being generated for this attribute:

```
<attribute type="java.lang.String" qualifier="myAttribute" generate="true">
    <modifiers write="false" />
</attribute>
```

i Note

Attribute value unwritable

If the setter method isn't generated, there's no way of writing the attribute value. The `setAttribute(...)` and `setAllAttributes(...)` methods internally also rely on the generated setter method and aren't operable either.

Element Discussion

For a discussion of the allowed elements within the `items.xml`, refer to the [items.xml Element Reference](#).

Setting Custom Types for New Columns

SAP Commerce can change the type of database table columns during an update process. Technically, this runs ALTER statements on the respective database table column and therefore changes the persistence setting of attributes directly on the database. For example, you can modify a database table column from **CHAR(255)** to **VARCHAR(255)** or the other way round.

It isn't possible to change the types of already existing database table columns by updating SAP Commerce. You can, however, set a required type and see it in place for a **new** table or a **new** column that you add to **items.xml** before you perform system update. Technically, by adding a new column, this runs ALTER statements on the respective database table column and therefore changes the persistence setting of attributes directly on the database. For example, you can modify a database table column from **CHAR(255)** to **VARCHAR(255)** or vice versa. Initialization always results in setting a required type for a table or column.

Caution

Loss of Data And Database Corruption Possible

We don't allow changing column types on existing tables. To prevent breaking the existing data, the database admin should decide on the correct way of introducing required changes.

Despite the fact that **java.lang.String** sets a default type for columns, you can have SAP Commerce modify a column type of a database table by modifying or explicitly specifying the value for the **<columntype>** element in the attribute definition in your **\$extensionname-items.xml** file:

```
<attribute qualifier="MyAttribute" type="java.lang.String">
    <description>Identifier of the store.</description>
    <modifiers read="true" write="true" search="true" optional="false" />
    <persistence type="property">
        <columntype>
            <value>VARCHAR</value>
        </columntype>
    </persistence>
</attribute>
```

Basically, there are two ways the database column type can be set:

- Explicitly, by specifying the database column type in the **items.xml** file, such as:

```
<persistence type="property">
    <columntype>
        <value>VARCHAR</value>
    </columntype>
</persistence>
```

You can also define this in more detail by specifying database systems such as:

```
<persistence type="property">
    <columntype database="oracle">
        <value>CL0B</value>
    </columntype>
    <columntype database="sap">
        <value>NCL0B</value>
    </columntype>
    <columntype database="sqlserver">
        <value>nvarchar(max)</value>
    </columntype>
    <columntype database="mysql">
        <value>text</value>
    </columntype>
    <columntype>
        <value>varchar(4000)</value>
    </columntype>
</persistence>
```

- Implicitly, by omitting the database column type in the `items.xml` file, such as:

```
<persistence type="property"/>
```

This causes the database to fall back to default values. For example, for an attribute of type `java.lang.String`, MySQL chooses `VARCHAR(255)` as default database column type.

Overriding Column Types

You can override a database column type, whether it's defined explicitly or implicitly. The property that allows you to override an XML configuration, uses the `persistence.override.<typeCode>.<qualifier>.columnType=<columnTypeValue>` naming convention. Platform checks whether a column type definition is given for `typeCode` and `qualifier` in the configuration files first. If not, it's read from `items.xml`.

The property overrides column type values for all database systems if they're defined as persistence property in `items.xml`. For example, for the `id` attribute for the `Catalog` type code, the following configuration makes all databases choose `VARCHAR(400)` as the database column type:

```
persistence.override.Catalog.id.columnType=varchar(400)
```

System Update Possibilities

Enumeration, Map, Collection

Although it's advisable to consult the support team if there are specific update procedures, the following rules apply. Scenarios that aren't described in the following table may fail or exceptions may occur.

Creating	removing, changing	Adding values	Removing values, changing values	Changing modifiers
Works	Works but cleanup necessary	Works	Doesn't work	Works

Relation

Creating	removing	changing	change partOF attribute	source/target attribute name change
Works	works but cleanup necessary	manual remapping needed, for (1-n) cleanup doesn't remove created instances, for (n-m) it does	works	works, cleanup necessary

Type

Creating	Removing	Changing Code, Deployment, Typecode attr in itemtype	Added Element to Inheritance Path	Removed Element from Inheritance Path	Inheritance path change to GenericItem
Works	Works if type is removed from composedtypes and attributedescriptors tables. Instances of types need to be removed, too.	Works if all three are changed, cleanup removes orphaned type	Works	Works, after update it's possible to add orphaned attribute but after cleanup it isn't possible	Works, cleanup necessary

Attribute

Creating	Removing	Changing	Changing attribute's persistence qualifier	Uniqueness change	Mandatory change	Persistence change	from Jalo to Dynamic	Deployment change
Works	Not working. Cleanup doesn't recognize orphaned types	Old attribute can still be added. Can't use cleanup as it doesn't recognize any orphaned type in this scenario. If new attribute is mandatory, expected validator is set	Remapping needs to be done manually	Doesn't work - no unique validator is set	Works, field is mandatory from now on	Works	some extra implementation has to be done	If data has to be truncated - not working. If changing to wider data type - works

Defining Index with Included Columns in `items.xml` for Microsoft SQL Server

In indices defined in the `item` type for Microsoft SQL Server, you can define which attribute qualifiers are added as included columns during the creation of indices, for example:

```
<itemtype code="ItemCode" ...>
  ...
  <indexes>
    <index name="indexName">
      <key attribute="keyAttribute"/>
      <include attribute="inclAttribute1"/>
      <include attribute="inclAttribute2"/>
    </index>
  </indexes>
</itemtype>
```

Such configuration translates into creating the following index:

```
CREATE INDEX indexName ON ItemCodeTable (keyAttributeCol) INCLUDE (inclAttribute1Col, inclAttribute2Col)
```

Defining Index with Included Columns in Configuration Files

You can include or override columns using the following property:

```
extend.index.for.<typeCode>.<indexName>.with.include=<attributeName1>,<attributeName2>
```

Separate multiple attributes using a comma ("").

Platform checks first whether there's a definition for included columns for typeCode and indexName in the configuration files. If such a definition doesn't exist, the system reads it from the `items.xml` file.

For example, the following property adds "email" and "postal code" as included columns for the existing index of Address_Owner for the Address type code:

```
extends.index.for.Address.Address_Owner.with.include=postalcode,email
```

Primitive Types

It's possible to use primitive Java types instead of the related wrapper classes. As an effect the jalo layer still uses the related wrapper classes, but the attribute definition gets a default value automatically (same default as Java uses). This ensures that you can change the type to a primitive class without any migration on jalo layer. At the servicelayer, the primitive type is used and the need for handling null values disappears.

Assumed that you have a definition like this:

```
<attribute qualifier="myAttribute" type="java.lang.Boolean">
    <modifiers read="true" write="true" initial="true" optional="false"/>
    <persistence type="property"/>
</attribute>
```

You should think of the null value situation. If you want to exclude the possibility of null values (by specifying a default value) then you can convert it to:

```
<attribute qualifier="myAttribute" type="boolean">
    <modifiers read="true" write="true" initial="true" optional="false"/>
    <persistence type="property"/>
</attribute>
```

An update system is sufficient to make the change effective.

Support for the @Deprecated Annotation Attributes

The `beans.xml` and `items.xml` files can have the optional `since` and `forRemoval` attributes of the `@Deprecated` annotation. You can use these attributes for classes, methods, or enums. The generated java classes have the `@Deprecated(since="xxxx", forRemoval="true")` annotation.

The value of the `java.lang.Deprecated#forRemoval` attribute must always be set to `true` by the code generator.

For more information, see <https://docs.oracle.com/javase/9/docs/api/java/lang/Deprecated.html>.

Related Information

[Type System Documentation](#)

[Build Framework](#)

[Initializing and Updating the SAP Commerce](#)

[Jalo Layer](#)

[Using Encryption for Attribute Values](#)

[Specifying a Deployment for hybris Platform Types](#)

[Jalo-only Attributes](#)

[Working with Enumerations](#)

<http://java.sun.com/products/jdo/JDOCMPFAQ.html> ↗

items.xml Element Reference

The `items.xml` file of each extension contains definitions of types for the corresponding extension.

Below you may find a list and explanation of all elements that may be configured in `items.xml` file.

→ Tip

XML Editor Recommended

If you create XML files based on the `items.xsd` file, you can create valid `items.xml` files easily. The IDE Eclipse, for example, contains such XML editors. Not only will your `items.xml` files content be valid, but Eclipse also shows description of XML elements from the XSD file.

Schema Document Properties

Target Namespace	None
Element and Attribute Namespaces	<ul style="list-style-type: none"> Global element and attribute declarations belong to this schema's target namespace. By default, local element declarations belong to this schema's target namespace. By default, local attribute declarations have no namespace.

Declared Namespaces

Prefix	Namespace
xml	http://www.w3.org/XML/1998/namespace
xs	http://www.w3.org/2001/XMLSchema

Global Declarations

Element: items

Type	Locally-defined complex type
------	------------------------------

Nillable	no
Abstract	no
Documentation	Defines the types of your extension.

XML Instance Representation

<items>	
<atomictypes>atomictypesType</atomictypes> [0..1]	Defines the list of atomictypesTypes for your extension.
<collectiontypes>collectiontypesType</collectiontypes> [0..1]	Defines the list of collectiontypesTypes for your extension.
<enumtypes>enumtypesType</enumtypes> [0..1]	Defines the list of enumtypesTypes for your extension.
<maptypes>maptypesType</maptypes> [0..1]	Defines the list of maptypesTypes for your extension.
<relations>relationsType</relations> [0..1]	Defines the list of relationsTypes for your extension.
<itemtypes>itemtypesType</itemtypes> [0..1]	Defines the list of itemtypesTypes for your extension.
</items>	

Global Definitions

Complex Type: atomictypesType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines a list of atomic types.

XML Instance Representation

<...>	
<atomictype>atomictypeType</atomictype> [0..*]	An atomictype represents a simple Java object. (The name 'atomic' just means 'non-composed' objects.)
</...>	

Complex Type: atomictypeType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	An AtomicType represents a simple java object. (The name 'atomic' just means 'non-composed' objects.)

XML Instance Representation

<...>	
class="classType" [1]	Corresponding Java class in the hybris Suite; will also be used as the code of the atomic type.
autocreate=" boolean " [0..1]	If 'true', the AtomicType will be created during initialization.
generate="boolean" [0..1]	Deprecated. Has no effect for atomic types. Default is 'true'.
extends="classType" [0..1]	Defines the class that will be extended. Default is 'java.lang.Object'.
/>	

Complex Type: attributeModelType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Allows to configure model generation for this attribute used at servicelayer.

XML Instance Representation

<...>	
generate="boolean" [0..1]	Whether getter and setter methods for the model representation of the attribute will be generated. Default is 'true'.
>	
<getter>modelMethodType</getter> [0..*]	Allows to configure alternative getter methods at generated model.
<setter>modelMethodType</setter> [0..*]	Allows to configure alternative setter methods at generated model.
</...>	

Complex Type: attributesType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures a list of attributes.

XML Instance Representation

<...>	
<attribute>attributeType</attribute> [0..*]	Defines a single attribute.

</...>

Complex Type: attributeType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines an attribute of a type.

XML Instance Representation

<...>	
redeclare="boolean" [0..1]	Lets you re-define the attribute definition from an inherited type. In essence, you can use a different type of attribute as well as different modifier combinations than on the supertype. Default is 'false'.
qualifier="string" [1]	Qualifier of this attribute. Attribute qualifiers must be unique across a single type.
type="string" [1]	The type of the attribute, such as 'Product', 'int' or 'java.lang.String'. Primitive java types will be mapped to the corresponding atomic type. For example: 'int' will be mapped to the atomic type 'java.lang.Integer' with implicit default value.
metatype="string" [0..1]	Advanced setting. Specifies the metatype for the attributes definition. Must be a type extending AttributeDescriptor. Default is 'AttributeDescriptor'.
autocreate="boolean" [0..1]	If 'true', the attribute descriptor will be created during initialization. Default is 'true'.
generate="boolean" [0..1]	If 'true', getter and setter methods for this attribute will be generated during a hybris Suite build. Default is 'true'.
isSelectionOf="string" [0..1]	References an attribute of the same type. Only values of the referenced attribute can be selected as values for this attribute. Typical example: the default delivery address of a customer must be one of the addresses set for the customer. Default is 'false'.
>	
<defaultValue>defaultValueType</defaultValue> [0..1]	Configures a default value for this attribute used if no value is provided. The default value is calculated by initialization and will not be re-calculated by runtime.
<description>string</description> [0..1]	Gives a description for this attribute only used for the javadoc of generated attribute methods.
<persistence>persistenceType</persistence> [0..1]	Defines how the values of the attribute will be stored. Possible values: 'cmp' (deprecated), 'jalo' (not persistent, deprecated), 'property' (persistent), 'dynamic' (not persisted).
<modifiers>modifiersType</modifiers> [0..1]	Configures advanced settings for this attribute definition.

<custom-properties>customPropertiesType</custom-properties> [0..1]	Allows to configure custom properties for this attribute.
<model>attributeModelType</model> [0..1]	Allows to configure model generation settings for this attribute. Models are used by the hybris ServiceLayer.
</...>	

Complex Type: collectiontypesType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines a list of collection types.

XML Instance Representation

<...>	
<collectiontype>collectiontypeType</collectiontype> [0..*]	A CollectionType defines a collection of typed elements.
</...>	

Complex Type: collectiontypeType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	A CollectionType defines a collection of typed elements. Attention: If using a collection type for persistent attributes (not jalo) you cannot search on that attribute and you are limited in size of collection. Consider to use a relation instead.

XML Instance Representation

<...>	
code="codeType" [1]	The code (that is, qualifier) of the CollectionType.
elementtype="codeType" [1]	The type of elements of this CollectionType.
autocreate="boolean" [0..1]	If 'true', the CollectionType will be created during initialization.
generate="boolean" [0..1]	Deprecated. Has no effect for collection types. Default is 'true'.
type=" NMTOKEN (value comes from list: {'set' 'list' 'collection'})" [0..1]	Configures the type of this collection: 'set', 'list', 'collection'. The getter / setter methods will use corresponding Java collection interfaces. Default is 'collection'.

/>>

Complex Type: columnType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures a persistence definition for a specific database.

XML Instance Representation

<...>	
database="string" [0..1]	The database the given definition will be used for. One of 'oracle', 'mysql', 'sqlserver' or 'hsqI'. Default is empty which configures fallback for non-specified databases.
>	
<value> string </value> [1]	The attribute type used in the create statement of the database table, such as varchar2(4000).
</...>	

Complex Type: customPropertiesType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines custom properties.

XML Instance Representation

<...>	
<property>customPropertyType</property> [0..*]	Defines a custom property.
</...>	

Complex Type: customPropertyType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines a custom property.

XML Instance Representation

<...	
name="string" [1]	The name of the custom property.
>	
<value>defaultValueType</value> [1]	The value of the custom property.
</...>	

Complex Type: deploymentType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	A deployment defines how a (generic) item or relation is mapped onto the database.

XML Instance Representation

<...	
table="string" [1]	The mapped database table. Must be globally unique.
typecode="positiveshort" [1]	The mapped item type code. Must be globally unique.
propertytable="string" [0..1]	The mapped dump property database table to be used for this item. Default is 'props'.
/>	

Complex Type: enumModelType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures a single enum model pojo.

XML Instance Representation

<...	
package="string" [0..1]	Defines the package for the actual enum model pojo.
/>	

Complex Type: enumtypesType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines a list of enumeration types.

XML Instance Representation

<...>	
<enumtype>enumtypeType</enumtype> [0..*]	An EnumerationType defines fixed value types. (The typesystem provides item enumeration only.)
</...>	

Complex Type: enumtypeType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	An EnumerationType defines fixed value types. (The typesystem provides item enumeration only)

XML Instance Representation

<...	
code="codeType" [1]	The unique code of this Enumeration.
autocreate="boolean" [0..1]	If 'true', the item will be created during initialization.
generate="boolean" [0..1]	If 'false' no constants will be generated at constant class of extension as well as at corresponding servicelayer enum class. Default is 'true'.
jaloClass="classType" [0..1]	Specifies the name of the associated jalo class. The specified class must extend de.hybris.platform.jalo.enumeration.EnumerationValue and will not be generated. By specifying a jalo class you can change the implementation to use for the values of this enumeration. By default EnumerationValue class is used.
dynamic="boolean" [0..1]	Whether it is possible to add new values by runtime. Also results in different types of enums: 'true' results in 'classic' hybris enums, 'false' results in Java enums. Default is false. Both kinds of enums are API compatible, and switching between enum types is possible by running a system update.
>	
<description>string</description> [0..1]	Provides possibility to add meaningful description phrase for a generated model class.

<model> enumModelType </model> [0..1]	Allows changing enum model settings.
<value> enumValueType </value> [0..*]	Configures one value of this Enumeration.
</...>	

Complex Type: enumValueType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures a single enum value.

XML Instance Representation

<...>	
code="enumCodeType " [1]	The unique code of this element.
>	
<description> string </description> [0..1]	Provides possibility to add meaningful description phrase for a generated model class.
</...>	

Complex Type: indexesType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures a list of indexes.

XML Instance Representation

<...>	
<index>indexType </index> [1..*]	Configures a single index.
</...>	

Complex Type: indexKeyType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures a single index key.

XML Instance Representation

<...	
attribute="string" [1]	Type attribute to be indexed.
lower="boolean" [0..1]	Elements will be indexed case-insensitive. Default is 'false'.
/>	

Complex Type: indexType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures a database index for enclosing type.

XML Instance Representation

<...	
name="string" [1]	The name prefix of the index.
remove="boolean" [0..1]	If 'true' this index will be omitted while in initialization process even if there were precedent declarations. This attribute has effect only if replace = true.
replace="boolean" [0..1]	If 'true' this index is a replacement/redeclaration for already existing index.
unique="boolean" [0..1]	If 'true', the value of this attribute has to be unique within all instances of this index. Attributes with persistence type set to 'jalo' cannot be unique. Default is 'false'.
>	
<key>indexKeyType</key> [0..*]	Configures a single index key.
</...>	

Complex Type: itemModelType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Allows to configure model generation for this item used at servicelayer.

XML Instance Representation

<...>	
generate="boolean" [0..1]	Whether a model for the type and models for subtypes will be generated. Default is 'true'.
>	
<constructor> modelConstructorType</constructor> [0..*]	Allows to configure model constructor signatures.
</...>	

Complex Type: itemtypesType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines a grouping of item types.

XML Instance Representation

<...>	
<itemtype>itemtypeType</itemtype> [0..*]	Specifies a specific ComposedType.
<typegroup>typeGroupType</typegroup> [0..*]	Specifies a group of ComposedTypes to allow better structuring within the items.xml file.
</...>	

Complex Type: itemtypeType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Specifies a specific ComposedType.

XML Instance Representation

<...>	
code=" codeType " [1]	The unique code of this type.
extends="classType" [0..1]	Defines the class, which will be extended. Default is 'GenericItem'.
jaloClass="classType" [0..1]	Specifies the name of the associated jalo class. Default is [extension-root-package].jalo.[type-code] which will be generated if not existent.
deployment="deploymentRefType" [0..1]	Deprecated, please use separate deployment sub tag. All instances of this type will be stored in a separated database table. The value

	of this attribute represents a reference to the specified deployment in the corresponding 'advanced-deployment.xml'. Default is empty.
singleton="boolean" [0..1]	If 'true', type gets marked as singleton which will be evaluated by some modules, or impex, with that allowing only one instance per system. Default is 'false'.
jaloonly="boolean" [0..1]	DEPRECATED. Use 'implements JaloOnlyItem' in your bean. If 'true', the item will only exists in the jalo layer and isn't backed by an entity bean. Default is 'false'.
autocreate="boolean" [0..1]	If 'true', the item will be created during initialization. Default is 'true'.
generate="boolean" [0..1]	If 'true', the sourcecode for this item will be created. Default is 'true'.
abstract="boolean" [0..1]	Marks type and jalo class as abstract. If 'true', the type cannot be instantiated. Default is 'false'.
metatype="string" [0..1]	The (meta)type which describes the assigned type. Must be a type extending ComposedType. Default is 'ComposedType'.
>	
<description>string</description> [0..1]	Provides possibility to add meaningful description phrase for a generated model class.
<deployment>deploymentType</deployment> [0..1]	A deployment defines how a (generic) item or relation is mapped onto the database.
<custom-properties>customPropertiesType</custom-properties> [0..1]	Defines a list of custom properties for this type.
<attributes>attributesType</attributes> [0..1]	Defines the list of item attributes.
<indexes>indexesType</indexes> [0..1]	Defines the database indexes for this type.
<model>itemModelType</model> [0..1]	Allows to configure model generation for this item used at service layer.
</...>	

Complex Type: maptypesType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Specifies a list of map types.

XML Instance Representation

<...>	
<maptype>maptypeType</maptype> [0..*]	Like the java collection framework, a type, which defines map objects. Attention: When used as type for an attribute, the attribute

	will not be searchable and the access performance is not effective. Consider to use a relation.
</...>	

Complex Type: mapTypeType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Like the java collection framework, a type, which defines map objects. Attention: When used as type for an attribute, the attribute will not be searchable and the access performance is not effective. Consider to use a relation.

XML Instance Representation

<...	
code=" codeType " [1]	The unique code of the map.
argumenttype="classType" [1]	The type of the key attributes.
returntype="classType" [1]	The type of the value attributes.
autocreate="boolean" [0..1]	If 'true', the item will be created during initialization. Default is 'true'.
generate="boolean" [0..1]	Deprecated. Has no effect for map types. Default is 'true'.
redeclare="boolean" [0..1]	Deprecated. Has no effect for map types. Default is 'false'.
/>	

Complex Type: modelConstructorType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Allows to configure model constructor signatures.

XML Instance Representation

<...	
signature="string" [1]	Add here, as comma-separated list, the attribute qualifiers for the constructor signature in the model.
/>	

Complex Type: modelMethodType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Allows to configure alternative methods at generated model.

XML Instance Representation

<...	
name="string" [1]	Name of the alternative getter method.
deprecated="boolean" [0..1]	Will the method be marked deprecated? Default is false.
default="boolean" [0..1]	Will this method be the default method and replace the original one instead of adding it additional? Default is false.
/>	

Complex Type: modifiersType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Specifies further properties of an attribute which can be redeclared at other extensions.

XML Instance Representation

<...	
read="boolean" [0..1]	Defines if this attribute is readable (that is, if a getter method will be generated). Default is 'true'. The visibility of the getter depends on the value of the private attribute.
write="boolean" [0..1]	Defines if this attribute is writable (that is, if a setter method will be generated). Default is 'true'. The visibility of the setter depends on the value of the private attribute.
search="boolean" [0..1]	Defines if this attribute is searchable by a FlexibleSearch. Default is 'true'. Attributes with persistence type set to 'jalo' cannot be searchable.
optional="boolean" [0..1]	Defines if this attribute is mandatory or optional. Default is 'true' for optional. Set to 'false' for mandatory.
private="boolean" [0..1]	Defines the Java visibility of the generated getter and setter methods for this attribute. If 'true', the visibility modifier of generated methods is set to 'protected'; if 'false', the visibility modifier is 'public'. Default is 'false' for 'public' generated methods. Also, you will have no generated methods in the ServiceLayer if 'true'.

initial="boolean" [0..1]	If 'true', the attribute will only be writable during the item creation. Setting this to 'true' is only useful in combination with write='false'. Default is 'false'.
removable="boolean" [0..1]	Defines if this attribute is removable. Default is 'true'.
partof="boolean" [0..1]	Defines if the assigned attribute value only belongs to the current instance of this type. Default is 'false'.
unique="boolean" [0..1]	If 'true', the value of this attribute has to be unique within all instances of this type. If there are multiple attributes marked as unique, then their combined values must be unique. Will not be evaluated at jalo layer, if you want to manage the attribute directly using jalo layer you have to ensure uniqueness manually. Default is 'false'.
	<p>⚠ Caution</p> <p>With some databases, a UNIQUE index permits multiple NULL values for columns that can contain NULL. As a result, you may find that unique indexes are non-functional when one of the unique index attributes is a null.</p> <p>For example, assume that you have a unique index enabled on attribute X, and a saved item with attribute X value set to null. When you create a new item with attribute X set to null, and save it:</p> <ul style="list-style-type: none"> • with databases that permit multiple null values in the unique column, the new item is saved without any error • with databases that prohibit multiple null values in the unique column, you get an exception and the new item isn't saved.
dontOptimize="boolean" [0..1]	If 'true' the attribute value will be stored in the 'global' property table, otherwise a separate column (inside the table of the associated type)will be created for storing its values. Default is 'false'.
encrypted="boolean" [0..1]	If 'true', the attribute value will be stored in an encrypted way. Default is 'false'.
/>	

Complex Type: persistenceType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines how the values of the attribute will be stored. Possible values: 'cmp' (deprecated), 'jalo' (not persistent), and 'property' (persistent).

XML Instance Representation

<...	
type=" MTOKEN (value comes from list: {'property' 'jalo' 'cmp' 'dynamic'})" [1]	Defines how the values of the attribute will be stored. Possible values: 'cmp' (deprecated), 'jalo' (not persistent, deprecated), 'property' (persistent), 'dynamic' (not persisted).
qualifier="string" [0..1]	Deprecated. Only usable in relation with 'cmp' and 'property'(compatibility reasons) persistence type. Default is empty.
attributeHandler="string" [0..1]	Spring bean id that handles dynamic attributes implementation.
>	
<columntype>columntypeType</columntype> [0..*]	Configures a persistence definition for a specific database used at create statement.
</...>	

Complex Type: relationElementType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures the generated attribute at one relation end.

XML Instance Representation

<...	
type="codeType" [1]	Type of attribute which will be generated at type configured for opposite relation end.
qualifier="string" [0..1]	Qualifier of attribute which will be generated at type configured for opposite relation end. If navigable is not set to false the qualifier is mandatory. Default is empty.
metatype="string" [0..1]	The (meta)type which describes the attributes type. Must be type extending RelationDescriptor. Default is 'RelationDescriptor'.
cardinality=" NMTOKEN (value comes from list: {'one' 'many'})" [0..1]	The cardinality of this relation end. Choose 'one' for 'one' part of a 1:n relation or 'many' when part of a n:m relation. A 1:1 relation is not supported. Default is 'many'.
navigable="boolean" [0..1]	Is the relation navigable from this side. Can only be disabled for one side of many to many relation. If disabled, no qualifier as well as modifiers can be defined. Default is 'true'.
collectiontype=" NMTOKEN (value comes from list: {'set' 'list' 'collection'})" [0..1]	Configures the type of this collection if element has cardinality 'many'. Related attribute getter / setter will use corresponding java collection interfaces. Default is 'Collection'.
ordered="boolean" [0..1]	If 'true' an additional ordering attribute will be generated for maintaining ordering. Default is 'false'.
>	

<description>string</description> [0..1]	Documents this relation attribute. Will be cited at javadoc of generated getters/setters.
<modifiers>modifiersType</modifiers> [0..1]	Defines properties for the attribute.
<model>attributeModelType</model> [0..1]	Allows to configure model generation for this relation attribute used at service layer.
<custom-properties>customPropertiesType</custom-properties> [0..1]	Allows to configure custom properties for the relation attribute.
</...>	

Complex Type: relationsType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines a list of relation types.

XML Instance Representation

<...>	
<relation>relationType</relation> [0..*]	A RelationType defines a n-m or 1-n relation between types.
</...>	

Complex Type: relationType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	A RelationType defines a n-m or 1-n relation between types.

XML Instance Representation

<...>	
code="codeType" [1]	The type code.
localized="boolean" [1]	A localized n-m relation can have a link between two items for each language.
deployment="deploymentRefType" [0..1]	Deprecated, please use separate deployment sub tag. All instances of this type will be stored in a separated database table. The value of this attribute represents a reference to the specified deployment in the corresponding 'advanced-deployment.xml'. Default is empty.

<code>autocreate="boolean" [0..1]</code>	If 'true', the item will be created during initialization.
<code>generate="boolean" [0..1]</code>	Deprecated. Will have no effect for relations.
<code>></code>	
<code><description>string</description> [0..1]</code>	Provides possibility to add meaningful description phrase for a generated model class.
<code><deployment>deploymentType/deployment> [0..1]</code>	Configures deployment information for this relation (table name and type code).
<code><sourceElement>relationElementType/sourceElement> [1]</code>	Configures the generated attribute at source relation end.
<code><targetElement>relationElementType/targetElement> [1]</code>	Configures the generated attribute at target relation end.
<code></...></code>	

Complex Type: typeGroupType

Parent type:	None
Sub-types:	None
Abstract	no

XML Instance Representation

<code><...></code>	
<code>name="string" [0..1]</code>	Defines the name of this group. Only for structural purpose, will have no effect on runtime. Default is empty.
<code>></code>	
<code><itemtype>itemtypeType</itemtype> [0..*]</code>	Specifies a specific ComposedType.
<code></...></code>	

Complex Type: valueType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures a single element.

XML Instance Representation

<code><...></code>	
<code>code=" codeType " [1]</code>	The unique code of this element.

/>

Simple Type: classType

Parent type:	normalizedString (derivation method: restriction)
Sub-types:	None
Content	Base XSD Type: normalizedString
Documentation	Configures the class to use for enclosing element.

Simple Type: codeType

Parent type:	normalizedString (derivation method: restriction)
Sub-types:	None
Content	Base XSD Type: normalizedString
Documentation	Configures the code of an element.

Simple Type: defaultValueType

Parent type:	string (derivation method: restriction)
Sub-types:	None
Content	Base XSD Type: string
Documentation	Defines a default value text.

Simple Type: deploymentRefType

Parent type:	normalizedString (derivation method: restriction)
Sub-types:	None
Content	Base XSD Type: normalizedString
Documentation	Deprecated. Defines a reference to a deployment definition.

Simple Type: enumCodeType

Parent type:	normalizedString (derivation method: restriction)
Sub-types:	None
Content	<ul style="list-style-type: none"> • Base XSD Type: normalizedString • <pattern> = ([a-zA-Z_])+([a-zA-Z\$0-9])*
Documentation	Configures the code of an enumeration value element. Must start with a letter or underscore.

Simple Type: positiveshort

Parent type:	short (derivation method: restriction)
Sub-types:	None
Content	<ul style="list-style-type: none"> Base XSD Type: short 0 <= <value> <= 32767

Impex with Ordered Relations in a Multithreaded Mode

Keeping the order for an ordered relation is crucial while impexing changes, or exporting and importing items between environments. The `ordering.attribute` property allows you to keep a correct order of items of an ordered relation while using impex in a multithreaded mode.

One-to-Many Relation

With one-to-many relations, where the **many** part should have a particular order, impex executed in a multithreaded mode doesn't provide a correct result. The lines are executed in a random order and the order is lost.

To keep the correct order when using impex, add `ordering.attribute` in `items.xml` for a given relation, and expose that attribute in the **many** item. Here is how `ordering.attribute` is defined out-of-the-box for the `AbstractOrder2AbstractOrderEntry` relation:

```

<relation code="AbstractOrder2AbstractOrderEntry" localized="false" generate="true" autocreate="true">
    <sourceElement type="AbstractOrder" qualifier="order" cardinality="one">
        <modifiers read="true" write="true" search="true" optional="true"/>
        <custom-properties>
            <property name="ordering.attribute">
                <value>entryNumber</value>
            </property>
        </custom-properties>
    </sourceElement>
    <targetElement type="AbstractOrderEntry" qualifier="entries" cardinality="many" collection="true" ordered="false">
        <modifiers read="true" write="true" search="true" optional="true" partof="true"/>
        <custom-properties>
            <property name="query.filter">
                <value>{original} is null</value>
            </property>
        </custom-properties>
    </targetElement>
</relation>
...
...
...

<attribute autocreate="true" qualifier="entryNumber" type="java.lang.Integer" generate="true">
    <defaultValue>Integer.valueOf(de.hybris.platform.jalo.order.AbstractOrder.APPEND_AS_LAST)</defaultValue>
    <persistence type="property"/>
    <modifiers read="true" write="true" search="true" optional="true" unique="true"/>
</attribute>
```

Adding these settings afterwards works fine on condition you use the same column name as in the original deployment. If you start from scratch, and the item was never released to production and no data exists, then you can choose your own name for the `ordering.attribute` value.

With those settings in place, you can add your new ordering column to your impex, as well as numbers for order, and it doesn't matter anymore in which order the impex lines are executed by a multi-threaded executor.

Specifying a Deployment for Platform Types

Items within SAP Commerce are made persistent by writing values into a database. Within the database, the values are stored in tables. SAP Commerce allows you to explicitly define the database tables where the values of instances of a given type will be written. This process is referred to as deployment.

Technical Background

SAP Commerce stores instances of types in tables within a database. Every instance is stored as one row in a database table.

The table within a database where instances of a type are stored is called the type deployment and is specified with the type itself. Every type needs to have a deployment to store its instances. Deployment is inherited from a type to its subtypes. The deployment that is active for a given type is the deployment specified closest to the type in the type's hierarchy. The topmost deployment is **GenericItem**, which is therefore the default deployment. This means if a type has no explicit specification of deployment, that type's instances are deployed in the same table as **GenericItem**.

This means that the default deployment of any subtype, which you extend from **GenericItem**, is the deployment of **GenericItem**. In other words: if you do not specify a deployment for a subtype of **GenericItem**, the instances of that subtype are stored in the same table as instances of **GenericItem**.

For example, the Catalog and CronJob types in SAP Commerce are subtypes of **GenericItem**. If there were no deployment specified for Catalog and CronJob, all Catalog and CronJob instances are written into one single database table. Firstly, this is not intuitive. Secondly, storing instances of many different types in one single database table causes that database table to have quite a lot of columns to store all attributes from all these types (a CronJob has different attributes than a Catalog, and both types need to store attribute values).

i Note

Specify Deployment for All Types

Deployment of a large number of types in a single table can markedly decrease the performance. Therefore, by default, builds fail if you do not specify the deployment table and you may encounter errors such as:

```
[ycheckdeployments] No deployment defined for relation <RELATIONNAME> in file: <FILENAME>
```

Otherwise it would be easy to forget to specify the deployment, and, as a result, some types would go to the **GenericItem** table. The build failure reminds you that deployment is not specified for some types. To change this default behaviour, set the property **build.development.mode** to **false** in the **local.properties** file:

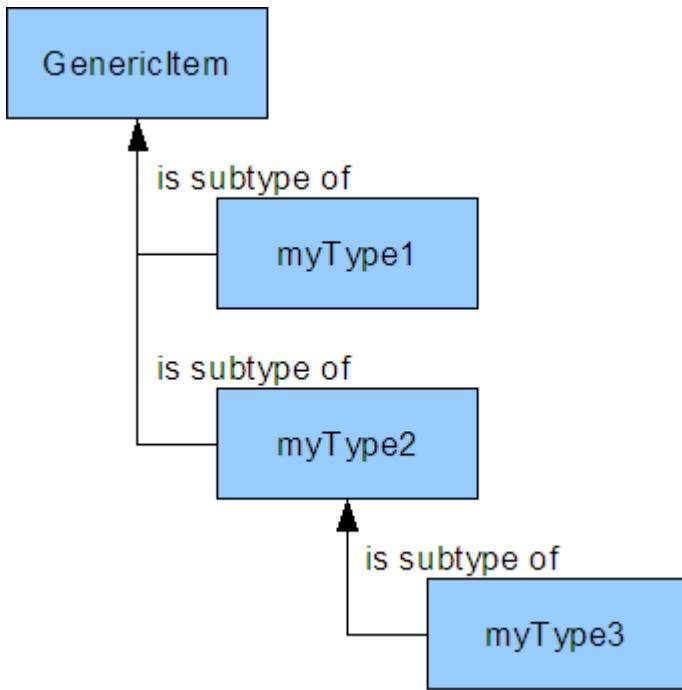
```
build.development.mode=false
```

Setting this value to false is useful for some legacy projects that keep all items in default tables (genericitems, links).

Keeping individual types' instances deployed in different tables keeps the number of columns down to a minimum.

By consequence, SAP recommends specifying a deployment for direct subtypes of **GenericItem** only. Subtypes of subtypes of **GenericItem** usually do not need a specific deployment as the number of database table columns in the deployment is not likely to get out of hand. In contrast, specifying a deployment for a subtype whose supertype has a deployment already is likely to reduce

database performance (especially during long and complex database transactions, such as synchronization between catalog versions).



For example, let's assume you extend `myType1` and `myType2` from `GenericItem`. Then it is recommended for `myType1` and for `myType2` to have a specific deployment (to avoid having their instances stored in the `GenericItem` database table).

A subtype of `myType1` or `myType2` (for example, `myType3` in the diagram) is not likely to also need a specific deployment. The instances of `myType3` fit into the `myType2` type's tables without any negative side effects. In fact, running FlexibleSearch statements on `myType2` requires JOINs to include the `myType3` type as well. The more deployments there are within a type hierarchy, the more JOINs in a database statement are necessary, and the longer complex database actions take to complete.

In other words:

- If you create a subtype of `GenericItem`, use a deployment.
- If you create a subtype of `Product`, which is a subtype of `GenericItem` already, using a specific deployment is discouraged by SAP. It is technically possible to use a deployment for subtypes whose supertypes already have an individual deployment, but it is not recommended. The JOINs required to construct database statements reduce performance.

Solution: Defining a Deployment in SAP Commerce

i Note

Database Limitations Apply

This form of specifying a deployment is affected by database limitations, which you have to comply with. Therefore, Platform only allows a deployment string of 24 characters maximum.

For example, Oracle databases only allow a maximum of 30 characters overall for the `table` attribute value.

This 30-character limit includes table prefixes. So if you use a table prefix of `myDataBase` (for a total of 10 characters), you have only 20 characters left for the `table` attribute value.

To specify a deployment, add a `<deployment table="tablename" typecode="typecode_number"/>` tag to the type definition in the `items.xml` file:

1. Open the **items.xml** in your extension.

2. Locate the type definition where you want to specify a deployment, such as

```
<itemtype code="MyItem" extends="GenericItem">
  <attribute qualifier="myAttribute" ... >
  ...
</itemtype>
```

3. Add the **<deployment>** tag nested into the item definition. You need to specify a value for the **deployment** and the **typecode** attributes :

```
<deployment table="mytype_deployment" />
```

- o The **deployment** attribute specifies the table name into which the instances of the type are written, such as **table="mytype_deployment"**
- o The **typecode** attribute must specify a unique number to reference the type. The value of the **typecode** attribute must be a positive integer between 0 and 32767 ($2^{15}-1$) and must be unique throughout SAP Commerce as it is part of the PK generation mechanism. Typecode values between 0 and 10000 are reserved for SAP Commerce-internal use. Typecode values larger than 10000 are generally free for you to use but there are lots of exceptions to that rule.

i Note

Not all typecode values larger than 10000 are free for you to use. There are many exceptions. Here are some examples:

- **commons** extension (132xx)
- **processing** extension (327xx)
- Legacy **xprint** extension (244xx,245xx)
- **b2bcommerce** extension (100xx)

For a full list of exceptions, see the

`<HYBRIS_BIN_DIR>/platform/ext/core/resources/core/unittest/reservedTypecodes.txt` file.

The entire type definition might look like this:

```
<item code="MyItem" extends="GenericItem">
  <deployment table="mytype_deployment" typecode="12345"/>
  <attribute name="myAttribute" ... >
  ...
</item>
```

Using a typecode that is already in use causes SAP Commerce to fail the build with the error message **due to duplicate deployment code**, as in:

```
[java] java.lang.IllegalArgumentException: cannot merge namespace ((customerreview)) into ((<merged> de.hybris.platform.persistence.customerreview_CustomerReview::((customerreview))::YDeployment de.hybris.platform.persistence.europel_DiscountRow::((europel))::YDeployment[europel.items..
```

Specifying or Changing Deployment for Relations

There are two important facts related to deployment for relations.

Firstly, you must specify a deployment for m:n relation, otherwise Platform will not build. Previously it was allowed to define an m:n relation which had no deployment. The relation was then maintained by the Links table. If there were multiple relations without a

specified deployment, they all resided in the Links table. This caused bad performance and is not a good practice. Therefore, whenever you try to define an m:n relation without a deployment, initialisation fails with the following error:

```
[ycheckdeployments] No deployment defined for relation <RELATIONNAME> in file: <FILENAME>
```

Secondly, bear in mind that Platform does not allow changing an existing deployment. This is done in order to protect data. If there were already some records in current deployment of a type or relation, and the deployment would get changed afterwards, then access would be lost to all those records.

This has a special consequence in conjunction with the previous paragraph (specifying a deployment), because if an m:n relation did not have a deployment before, and now (according to best practices) someone wants to assign it a deployment, this change will not be performed.

Every time Platform refuses to make a deployment change, one of the following messages is printed:

- If there was no deployment before:

```
Addition of the deployment for type <TYPECODE> from <OLDDEPLOYMENT> to <NEWDEPLOYMENT> will n
```

- If there was another deployment defined before:

```
Modification of the deployment for type <TYPECODE> from <OLDDEPLOYMENT> to <NEWDEPLOYMENT> wi
```

Using a Custom Property Table

If you want to use an own propertytable, you have to specify an own `extensionname-advanced-deployment.xml` in the resource folder of your extension. To use the table `testtable`, you would need the content of this file like this:

```
<model name="hybris" description="...">
    <package name="de.hybris.jakarta.session" description="all session beans">
        <package name="de.hybris.jakarta.session.property">

            <object name="Property">
                <object-mapping>
                    <table name="testtable"/>
                    <index name="ITEMPK">
                        <index-key attribute="itemPK"/>
                    </index>
                    <index name="NAMEIDX">
                        <index-key attribute="name"/>
                    </index>
                </object-mapping>
                <attribute name="itemPK" type="HYBRIS.PK" primkey-field="true">
                    <attribute-mapping persistence-name="ITEMPK" null-allowed="false"/>
                </attribute>
                <attribute name="itemTypePK" type="HYBRIS.PK" >
                    <attribute-mapping persistence-name="ITEMTYPEPK" null-allowed="false"/>
                </attribute>
                <attribute name="name" type="java.lang.String" primkey-field="true">
                    <attribute-mapping persistence-name="NAME" null-allowed="false"/>
                </attribute>
                <attribute name="langPK" type="HYBRIS.PK" primkey-field="true">
                    <attribute-mapping persistence-name="LANGPK" null-allowed="false"/>
                </attribute>
                <attribute name="realName" type="java.lang.String">
                    <attribute-mapping persistence-name="REALNAME"/>
                </attribute>
                <attribute name="type1" type="int">
                    <attribute-mapping persistence-name="TYPE1"/>
                </attribute>
                <attribute name="valueString1" type="HYBRIS.LONG_STRING">
                    <attribute-mapping database="sqlserver" persistence-name="VALUESTRING1" persiste
                    <attribute-mapping database="oracle" persistence-name="VALUESTRING1" persiste
                    <attribute-mapping persistence-name="VALUESTRING1"/>
```

```

        </attribute>
        <attribute name="value1" type="java.io.Serializable">
            <attribute-mapping persistence-name="VALUE1"/>
            <attribute-mapping database="oracle" persistence-name="VALUE1" persistence-type='
        </attribute>
    </object>

</package>
</model>

```

SAP Commerce contains the    Maintenance > Deployment page in Administration Console that gives an overview of the typecodes and the deployments in use.

Advanced Deployment

To see how SAP Commerce types are mapped to different databases, see the file

bin/platform/ext/core/resources/core-advanced-deployment.xml.

```

<database-schema database="hsqldb" primary-key="primary key" null="" not-null="not null" >
    <type-mapping type="java.lang.String" persistence-type="VARCHAR(255)" />
    <type-mapping type="String" persistence-type="VARCHAR(255)" />

    <type-mapping type="java.lang.Float" persistence-type="float" />
    <type-mapping type="java.lang.Double" persistence-type="double" />
    <type-mapping type="java.lang.Byte" persistence-type="smallint" />
    <type-mapping type="java.lang.Character" persistence-type="smallint" />
    <type-mapping type="java.lang.Short" persistence-type="smallint" />
    <type-mapping type="java.lang.Boolean" persistence-type="tinyint" />
    <type-mapping type="java.lang.Long" persistence-type="bigint" />
    <type-mapping type="java.lang.Integer" persistence-type="int" />

    <type-mapping type="float" persistence-type="float default 0" />
    <type-mapping type="double" persistence-type="double default 0" />
    <type-mapping type="byte" persistence-type="smallint default 0" />
    <type-mapping type="char" persistence-type="smallint default 0" />
    <type-mapping type="short" persistence-type="smallint default 0" />
    <type-mapping type="boolean" persistence-type="tinyint default 0" />
    <type-mapping type="long" persistence-type="bigint default 0" />
    <type-mapping type="int" persistence-type="int default 0" />

    <type-mapping type="java.util.Date" persistence-type="timestamp" />
    <type-mapping type="java.math.BigDecimal" persistence-type="DECIMAL(30,8)" />
    <type-mapping type="java.io.Serializable" persistence-type="longvarbinary" />

    <type-mapping type="HYBRIS.LONG_STRING" persistence-type="LONGVARCHAR" />
    <type-mapping type="HYBRIS.JSON" persistence-type="LONGVARCHAR" />
    <type-mapping type="HYBRIS.COMMA_SEPARATED_PKS" persistence-type="LONGVARCHAR" />
    <type-mapping type="HYBRIS.PK" persistence-type="BIGINT" />

</database-schema>

```

Related Information

[Third-Party Databases](#)

Cleaning Up the Type System

The update process of SAP Commerce doesn't remove data from the database. Especially the type system representation held in the database is not affected in case of a type removal. This leads to the situation that types removed from the `items.xml` file still have their representation in the database.

To clean up the type system, it is necessary to delete from the database all instances of items defined within the type that should not exist and the related type representation. It is possible to do this using SAP Commerce Administration Console.

Type Cleaner deletes all types, their attributes, relations, enums, and collection types. Note that it does not delete attributes unless the types they belong to have been deleted.

Orphaned Types

An orphaned type is a type that is not defined in the `items.xml` file but still exists in the type system representation held in the database. To make sure that essential data are not deleted from the database accidentally, the update process doesn't remove data from database, even if a type is removed from `items.xml` file. By keeping the type system representation in database the generic access to this data is still assured. It is possible by using the SAP Commerce API, for example

`ModelService.getAttributeValue` method. The disadvantage is that you may encounter warnings on the console saying that the related model class cannot be found anymore when accessing the data and that the database itself will keep a lot of possibly unneeded data.

Cleaning All Orphaned Types

1. Open Administration Console.
2. Go to the **Maintenance** tab and select **Cleanup** option.
3. The **Cleanup** page in the **Type system** tab displays.
4. Ensure both check boxes are selected and click the **Clear all orphaned types** button.
5. Orphaned types are removed

Related Information

[The Type System](#)

[Administration Console](#)

Dynamic Attributes

All SAP Commerce Models are automatically generated during the Platform build, thus they cannot hold any custom changes. Dynamic Attributes enable you to add attributes to the Models, and to create custom logic behind them, without touching the SAP Commerce Model class itself.

In contrast to attributes with the `persistence` type set to `property` that are persisted in a database, Dynamic Attributes have non-persistent values. The fact that there is some custom logic written in a separate class is absolutely transparent to the user. The implementation of Dynamic Attributes is supported by Spring, therefore, you can benefit from it by using a dependency injection, for example.

Overview of Dynamic Attributes

A Dynamic Attribute is defined for a type in the `items.xml` file. You need to set the `persistence` type of the attribute to `dynamic`. For each Dynamic Attribute, a Spring bean ID known as `attributeHandler` is automatically generated. However, you may provide the custom bean ID.

→ Tip

The naming convention of the automatically generated `attributeHandler` object is:

`<ItemtypeCode>_<attributeQualifier>AttributeHandler`. For example, the bean ID for the Dynamic Attribute with the

qualifier `daysHumanReadable` for the `Accommodation` type is `Accommodation_daysHumanReadableAttributeHandler`.

The following example shows the configuration of the Dynamic Attribute in the `items.xml` file:

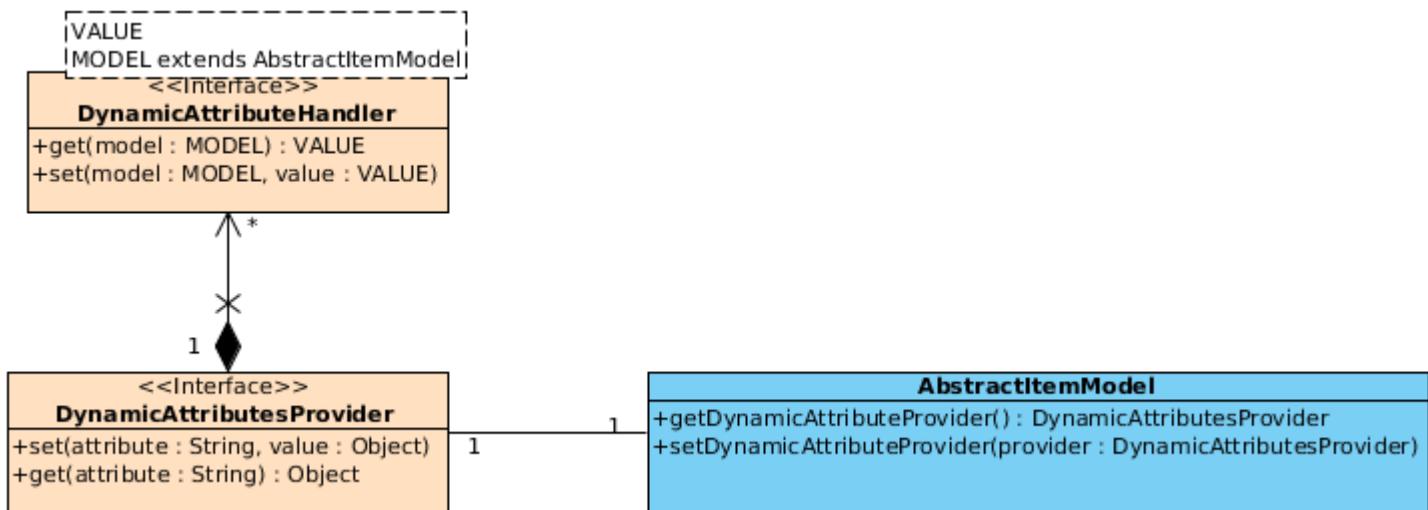
```
<itemtype code="Accommodation" autocreate="true" generate="true" jaloclass="de.hybris.platform.test.Accommodation">
    <attributes>
        <attribute type="int" qualifier="daysHumanReadable">
            <persistence type="dynamic" attributeHandler="myOwnSpringBeanId" />
            <modifiers read="true" write="false" />
        </attribute>
    </attributes>
</itemtype>
```

Because the write modifier is set to `false`, only getter for the attribute is generated in the `AccommodationModel` class.

See also the [How To Create Dynamic Attributes - Tutorial](#), section **Define a Dynamic Attribute in the items.xml File**.

How Dynamic Attributes are Related to the SAP Commerce Model

The following UML diagram illustrates how the SAP Commerce Model is internally related to the Dynamic Attribute interfaces:



The `AbstractItemModel` type holds a reference to the `DynamicAttributesProvider` object, which in turn, holds a collection of `DynamicAttributeHandler` objects. The crucial feature here is that by using the methods `get` and `set` on the `DynamicAttributesProvider` object, the call is delegated to the valid `DynamicAttributeHandler` instance. From the user perspective, this process is transparent, so the only interaction with the `DynamicAttributeHandler` instance is done by calling `getter` on the SAP Commerce Model. From the perspective of the life cycle of the SAP Commerce Model, the `DynamicAttributeHandler` instance is called exactly at calling `setter` or `getter` on the SAP Commerce Model:

```
// AccommodationModel instance
String days = accommodation.getDaysHumanReadable();
```

Using Dynamic Attributes

Before creating a Dynamic Attribute for any type, you need to be sure that the logic behind the Dynamic Attribute concerns directly the type, for which you want to define it. Otherwise, it is better to keep the logic in a proper service.

For the above example, the generated `AccommodationModel` type has an attribute of `int` type that is persisted in the database. This attribute represents the number of days, that a specific apartment is booked for. Imagine you intend to create a method that

returns a String representation of the number of days, concatenated with the word day or days, depending on the number. It can be achieved using the Dynamic Attribute and named, for example, daysHumanReadable in items.xml:

```
<itemtype code="Accommodation" autocreate="true" generate="true" jaloClass="de.hybris.platform.test.Accommodation">
    <attributes>
        <!-- ... -->
        <attribute type="java.lang.String" qualifier="daysHumanReadable">
            <persistence type="dynamic" attributeHandler="accommodationDays" />
            <modifiers read="true" write="false" />
        </attribute>
        <!-- ... -->
    </attributes>
</itemtype>
```

The implementation of the `DynamicAttributeHandler` in class looks like this:

AccommodationDays.java

```
public class AccommodationDays implements DynamicAttributeHandler<String, AccommodationModel>
{
    @Override
    public String get(final AccommodationModel model)
    {
        String daysWord;
        final int days = model.getDays();
        if (days == 1)
        {
            daysWord = "day";
        }
        else
        {
            daysWord = "days";
        }
        return days + " " + daysWord;
    }

    @Override
    public void set(final AccommodationModel model, final String value)
    {
        throw new UnsupportedOperationException();
    }
}
```

To get the information for how many days a user has booked an apartment, you only need to call `getter` on an instance of `AccommodationModel`:

```
// AccommodationModel instance
String days = accommodation.getDaysHumanReadable();
```

It is recommended to implement the above method to throw an exception, because the write modifier of the `daysHumanReadable` attribute is set to `false`. Thus there is no `setter` generated in the `AccommodationModel`.

See also [How To Create Dynamic Attributes - Tutorial](#), section **Implement the DynamicAttributeHandler**.

Using Localized Dynamic Attributes

You can localize dynamic attributes. If you want to use dynamic localized attributes, you need to implement the `DynamicLocalizedAttributeHandler` object. In addition, it gives you the possibility to use locales in your setters and getters. Below is an example `DynamicLocalizedAttributesStringSampleBean` implementation:

```

public class DynamicLocalizedAttributesStringSampleBean implements DynamicLocalizedAttributeHandler
{
    private I18NService i18NService;

    @Override
    public String get(final TestItemType2Model item)
    {
        if (item == null)
        {
            throw new IllegalArgumentException("Item model is required");
        }

        return item.getTestProperty2(i18NService.getCurrentLocale());
    }

    @Override
    public void set(final TestItemType2Model item, final String value)
    {
        if (item != null && value != null)
        {
            item.setTestProperty2(value, i18NService.getCurrentLocale());
        }
    }

    @Override
    public String get(final TestItemType2Model item, final Locale loc)
    {
        if (item == null)
        {
            throw new IllegalArgumentException("Item model is required");
        }

        return item.getTestProperty2(loc);
    }

    @Override
    public void set(final TestItemType2Model item, final String value, final Locale loc)
    {
        if (item != null && value != null)
        {
            item.setTestProperty2(value, loc);
        }
    }

}
//....
}

```

Below is an example of how the localized attribute can be declared in `items.xml`:

```

<itemtype code="TestItemType2"
          extends="TestItem"
          jaloclass="de.hybris.platform.jalo.test.TestItem"
          autocreate="true"
          generate="false">
    <attributes>
        <attribute autocreate="true" qualifier="testProperty2" type="localized:java.lang.String">
            <modifiers read="true" write="true" search="false" optional="true" />
            <persistence type="property" />
        </attribute>
    // 
        <attribute type="localized:java.lang.String" qualifier="localizedFooBar">
            <persistence type="dynamic" attributeHandler="dynamicLocalizedAttributesStringSample" />
            <modifiers read="true" write="true" optional="true" unique="false"/>
        </attribute>
    </attributes>
</itemtype>

```

The next step is to register the `dynamicLocalizedAttributesStringSampleBean` class in Spring context. For details on how to do this, read the [How To Create Dynamic Attributes - Tutorial](#), section **Register the Spring Bean**.

Every call on a model for the get or set methods using the `localizedFoobar` attribute leads to an update, or gets the localized `testProperty2` field using given locales. It is also up to you how you implement the methods without the locale parameter. In the example above, the current locale is fetched using the `i18nService` object.

Related Information

[Models](#)

[items.xml](#)

[ServiceLayer](#)

Creating Dynamic Attributes

Use Dynamic Attributes to define attributes whose values are not persisted in a database.

Define the Dynamic Attribute in the items.xml File

1. In the `items.xml` file, create new `ClientName` type with two attributes, which `persistence type` is set to `property`. Their values are persisted in the database.

```
<itemtype code="ClientName"
    extends="GenericItem"
    jaloclass="de.hybris.tutorial.jalo.ClientName"
    autocreate="true"
    generate="false">
<attributes>
    <attribute autocreate="true" qualifier="firstName" type="java.lang.String">
        <modifiers read="true" write="true" search="false" optional="true" />
        <persistence type="property" />
    </attribute>
    <attribute autocreate="true" qualifier="lastName" type="java.lang.String">
        <modifiers read="true" write="true" search="false" optional="true" />
        <persistence type="property" />
    </attribute>
</attributes>
</itemtype>
```

2. Create an attribute that can compute its value in the memory by using values from persisted attributes and return the result. For this you may use the Dynamic Attribute:

- a. Add the new Dynamic Attribute named `longName`, which uses existing attributes `firstName` and `lastName` to compute their values.
- b. Set the `persistence type` of new attribute to `dynamic`

```
<!-- ... -->
<attribute type="java.lang.String" qualifier="longName">
    <persistence type="dynamic"/>
    <modifiers read="true" write="true" optional="true" unique="false"/>
</attribute>
<!-- ... -->
```

3. If you do not provide the custom bean ID for `attributeHandler`, it is automatically generated in the following way: `ClientName_longNameAttributeHandler`.

4. Provide a custom `attributeHandler` by assigning a bean ID of the class that implements the `DynamicAttributeHandler` interface and holds the logic:

```
<!-- ... -->
<attribute type="java.lang.String" qualifier="longName">
    <persistence type="dynamic" attributeHandler="dynamicAttributesStringSample"/>
    <modifiers read="true" write="true" optional="true" unique="false"/>
</attribute>
<!-- ... -->
```

Implement the DynamicAttributeHandler

1. Create a new class named `DynamicAttributesStringSample` that implements `DynamicAttributeHandler` interface.
2. Override the getter and setter methods:
 - a. `get` method returns a String that concatenates values of attributes `firstName` and `lastName` with a defined delimiter.
 - b. `set` method splits the String passed as a parameter into two parts and assigns them to attributes `firstName` and `lastName`.

`DynamicAttributesStringSample.java`

```
public class DynamicAttributesStringSample implements DynamicAttributeHandler<String, ClientNameModel>
{
    public static final String VALUE_DELIMITER = " ";

    @Override
    public String get(final ClientNameModel item)
    {
        if (item == null)
        {
            throw new IllegalArgumentException("Item model is required");
        }
        return item.getFirstName() + VALUE_DELIMITER + item.getLastName();
    }

    @Override
    public void set(final ClientNameModel item, final String value)
    {
        if (item != null && value != null)
        {
            final String[] split = value.split(VALUE_DELIMITER);
            item.setFirstName(split[0]);
            item.setLastName(split[1]);
        }
    }
}
```

You can write custom logic using `ClientNameModel item` and `String value`, that are passed as parameters. It is possible to use any Spring service and just inject it into `DynamicAttributeHandler` implementation.

Register the Spring Bean

Register the newly created `attributeHandler` in Spring context:

`core-spring.xml`

```
<bean id="dynamicAttributesStringSample" class="de.hybris.platform.servicelayer.test.DynamicAttributeHandler">
```

Update Running System

Update running system using the hybris Administration Console or the command line using the following command:

```
ant all updatesystem -Dtenant=master
```

Sources are compiled and in the `ClientNameModel` proper getter and setter are generated.

Migrating Jalo Attributes to Dynamic Attributes

This document outlines how to migrate Jalo attributes to dynamic attributes.

i Note

Jalo is deprecated since SAP Commerce 4.3.0 version.

Since SAP Commerce 4.4.0 version, Dynamic Attributes replace Jalo attributes.

Migrate Jalo attributes to Dynamic Attributes using the following procedure. It is separated into two examples based on real migrations:

Basic Example

Follow the steps to migrate the `allSubcategories` Jalo attribute to a dynamic attribute.

Complex Example

Follow these steps to migrate the `description` Jalo attribute to a dynamic attribute.

Basic Example

Follow the steps to migrate the `allSubcategories` Jalo attribute to a dynamic attribute.

Modify the `category-items.xml` File

This example is taken from the `category` extension, where prior to SAP Commerce 4.4.0 version `Category` type contained `allSubcategories` Jalo attribute. The goal is to migrate this attribute to the Dynamic Attribute. The following code sample presents a definition of `Category` type and its `allSubcategories` Jalo attribute for the SAP Commerce version prior to 4.4.0. It has `persistence type` set to `jalo`:

```
<itemtype code="Category"
    generate="true"
    jaloclass="de.hybris.platform.category.jalo.Category"
    extends="GenericItem"
    autocreate="true">
    <deployment table="Categories" typecode="142"/>
    <attributes>
        <!-- ... -->
        <attribute qualifier="allSubcategories" type="CategoryCollection">
            <modifiers read="true" write="false" search="false" optional="true"/>
            <persistence type="jalo"/>
        </attribute>
        <!-- ... -->
    </attributes>
</itemtype>
```

To migrate `allSubcategories` attribute to the Dynamic Attribute, you need to change its definition in the `items.xml` file:

1. Modify the `persistence type` to `dynamic`.
2. Provide the custom bean id for the `attributeHandler`.

```
...
    <attribute qualifier="allSubcategories" type="CategoryCollection">
        <modifiers read="true" write="false" search="false" optional="true"/>
        <persistence type="dynamic" attributeHandler="categoryAllSubcategories"/>
```

```
</attribute>
...
```

Implement the DynamicAttributeHandler in CategoryAllSubcategories Class

Having defined the bean for the `attributeHandler`, you need to implement the corresponding class:

1. Write custom logic for `categoryAllSubcategories` in a bean class that implements `DynamicAttributeHandler` interface:

```
public class CategoryAllSubcategories implements DynamicAttributeHandler<Collection<CategoryModel>>
{
    private CategoryService categoryService;

    @Override
    public Collection<CategoryModel> get(final CategoryModel category)
    {
        return categoryService.getAllSubcategoriesForCategory(category);
    }

    @Override
    public void set(final CategoryModel model, final Collection<CategoryModel> value)
    {
        throw new UnsupportedOperationException();
    }

    @Required
    public void setCategoryService(final CategoryService categoryService)
    {
        this.categoryService = categoryService;
    }
}
```

2. Register the newly created `attributeHandler` in Spring context:

`category-spring.xml`

```
<bean id="categoryAllSubcategories" class="de.hybris.platform.category.attribute.CategoryAllSubcategories">
    <property name="categoryService" ref="categoryService" />
</bean>
```

The `categoryAllSubcategories` bean contains one property `categoryService`, which injects `CategoryService` class.

Ensure the Backward Compatibility of Category

i Note

This step is optional. Perform this when backward compatibility with Jalo layer is required.

To ensure the backward compatibility, you need to make some changes before the compilation. For every type, an abstract class `GeneratedItemtypeCode` is generated. Prior to 4.4.0, each generated class contained getters and setters for every attribute defined for a type. Since SAP Commerce 4.4.0 Jalo attributes are replaced by Dynamic Attributes. Getters and setters are not generated for Dynamic Attributes in `GeneratedItemtypeCode` class. It means that every call for `getAllSubcategories()` method on `Category` class fails. You may use Dynamic Attributes only using the ServiceLayer API on `CategoryModel` class.

In below example from the `category` extension, the following getters are automatically generated in `GeneratedCategory` class prior to 4.4.0:

```
public abstract class GeneratedCategory extends GenericItem
{
```

```

// ...
public Collection<Category> getAllSubcategories()
{
    return getAllSubcategories(getSession().getSessionContext());
}

public abstract Collection<Category> getAllSubcategories(final SessionContext ctx);
// ...
}

```

The `getAllSubcategories(final SessionContext ctx)` method is marked as `abstract`, so you need to implement it in each class extending `GeneratedCategory` class.

Move these two methods to `Category` class to ensure that it is not overwritten during next compilation. They should look similar but with additional annotation `@Deprecated` and without `@Override` annotation.

```

public class Category extends GeneratedCategory
{
    // ...
    @Deprecated
    public Collection<Category> getAllSubcategories()
    {
        return getAllSubcategories(getSession().getSessionContext());
    }
    @Deprecated
    public Collection<Category> getAllSubcategories(final SessionContext ctx)
    {
        //Method implementation.
    }
    // ...
}

```

Complex Example

Follow these steps to migrate the `description` Jalo attribute to a dynamic attribute.

Modify the `cms2-items.xml` File

This example is taken from the `cms2` extension, where prior to SAP Commerce 4.4.0 version `AbstractRestriction` type contained `description` Jalo attribute. There are also few classes extending `AbstractRestriction` and each of them has different custom logic for `description` attribute. The goal is to migrate this attribute to the Dynamic Attribute. The following code sample presents a definition of `AbstractRestriction` type and its `description` Jalo attribute. It has `persistence type` set to `jalo`:

```

<itemtype code="AbstractRestriction"
          jaloclass="de.hybris.platform.cms2.jalo.restrictions.AbstractRestriction"
          extends="CMSSItem"
          autocreate="true"
          generate="true"
          abstract="true">
    <attributes>
        <!-- ... -->
        <attribute qualifier="description" generate="true" autocreate="true" type="localized:java.1"
                  <persistence type="jalo" />
                  <modifiers write="false" />
        </attribute>
        <!-- ... -->
    </attributes>
</itemtype>

```

There are several types in the cms2 extension that extend `AbstractRestriction` type. In this case, the implementation of `description` Jalo attribute goes into concrete types that extend `AbstractRestriction` type. Introduce the following changes in the `items.xml` file:

1. Change `persistence type` of `description` attribute to `dynamic`:

```
...
<attribute qualifier="description" generate="true" autocreate="true" type="java.lang.String">
    <persistence type="dynamic" />
    <modifiers write="false" />
</attribute>
...
```

2. Define separate Dynamic Attributes for each type that extends `AbstractRestriction` type:

- o Add `redeclare` attribute in `<attribute />` tag and set its value to `true`.

Redeclaration in subtypes is required, because you want to provide different logic for each subtype. It means that each subtype will have its own attribute handler. Without these changes, it is not possible to build Platform.

- o Change `persistence type` of attribute to `dynamic`.
- o Provide a custom bean if for `attributeHandler`.

```
<itemtype code="CMSCatalogRestriction"
    jaloClass="de.hybris.platform.cms2.jalo.restrictions.CatalogRestriction"
    extends="AbstractRestriction"
    autocreate="true"
    generate="true">
    <attributes>
        <attribute qualifier="description" type="java.lang.String" redeclare="true">
            <persistence type="dynamic" attributeHandler="catalogRestrictionHandler" />
            <modifiers write="false" />
        </attribute>
    </attributes>
</itemtype>

<itemtype code="CMSInverseRestriction"
    jaloClass="de.hybris.platform.cms2.jalo.restrictions.CMSInverseRestriction"
    extends="AbstractRestriction"
    autocreate="true"
    generate="true">
    <attributes>
        <attribute qualifier="description" redeclare="true" type="java.lang.String" attributeHandler="inverseRestrictionHandler" />
        <persistence type="dynamic" />
        <modifiers write="false" />
    </attribute>
    </attributes>
</itemtype>
```

Implement the `DynamicAttributeHandler` in `InverseRestrictionDescription` and `CatalogRestrictionDescription` Classes

1. Write custom logic for `description` in a bean class that implements `DynamicAttributeHandler` interface. Do it for both bean classes:

```
public class InverseRestrictionDescription implements DynamicAttributeHandler<String, CMSInverseRestriction>
{
    @Override
    public String get(final CMSInverseRestrictionModel model)
    {
        return Localization.getLocalizedString("type.cmsinverserestriction.description")
            + model.getOriginalRestriction().getDescription();
    }

    @Override
    public void set(final CMSInverseRestrictionModel model, final String value)
```

```

        {
            throw new UnsupportedOperationException();
        }

public class CatalogRestrictionDescription implements DynamicAttributeHandler<String, CMSCatalog>
{
    @Override
    public String get(final CMSCatalogRestrictionModel model)
    {
        final Collection<CatalogModel> catalogs = model.getCatalogs();
        final StringBuilder result = new StringBuilder();
        if (!catalogs.isEmpty())
        {
            final String localizedString = Localization.getLocalizedString("type");
            result.append(localizedString == null ? "Page only applies on catalog"
            for (final CatalogModel cat : catalogs)
            {
                result.append(" ").append(cat.getName()).append(" (").append(
            }
        }
        return result.toString();
    }

    @Override
    public void set(final CMSCatalogRestrictionModel model, final String value)
    {
        throw new UnsupportedOperationException();
    }
}

```

2. Register the newly created attributeHandlers in Spring context:

`cms2-spring.xml`

```

<bean id="inverseRestrictionDynamicDescription" class="de.hybris.platform.cms2.model.InverseR
<bean id="catalogRestrictionDynamicDescription" class="de.hybris.platform.cms2.model.CatalogR

```

Ensure Backward Compatibility of AbstractRestriction

i Note

This step is optional. You need to perform this when backward compatibility with Jalo layer is required.

To ensure a backward compatibility, you need to make some changes before the compilation. For more detail, see Ensure the Backward Compatibility of Category section above. You may use Dynamic Attributes only using the ServiceLayer API on `CatalogRestrictionModel` class.

In below example from the `cms2` extension, the following getters are automatically generated in `GeneratedAbstractRestriction` class prior to 4.4.0:

```

public abstract class GeneratedAbstractRestriction extends CMSItem
{
    // ...
    public abstract String getDescription(final SessionContext ctx);

    public String getDescription()
    {
        return getDescription(getSession().getSessionContext());
    }
    // ...
}

```

Similar as in the Basic Example above, `getDescription(final SessionContext ctx)` method is marked as `abstract`, so you need to implement it in each class extending `GeneratedAbstractRestriction` class.

- Move these two methods to `AbstractRestriction` class to ensure that it is not overwritten during next compilation. They should look similar but with additional annotation `@Deprecated` and without `@Override` annotation.

```
public abstract class AbstractRestriction extends GeneratedAbstractRestriction
{
    // ...
    @Deprecated
    public abstract String getDescription(final SessionContext ctx);

    @Deprecated
    public String getDescription()
    {
        return getDescription(getSession().getSessionContext());
    }
    // ...
}
```

- Now, you can add `@Deprecated` annotation in any subclass implementation like this:

```
public class CatalogRestriction extends GeneratedCatalogRestriction
{

    /**
     * @deprecated use {@link de.hybris.platform.cms2.model.restrictions.CMSCatalogRestri
     * instead.
     */
    @Deprecated
    @Override
    public String getDescription(final SessionContext ctx)
    {
        //Method implementation.
    }
}
```

Working with Enumerations

SAP Commerce enables you to use attributes with pre-defined values, called enumeration. That way, a user will have limited number of possible choices at their disposal in a drop-down menu.

Creating an Enumeration

```
<enumtypes>
    <enumtype code="enumSampleType" autocreate="true" generate="true">
        <value code="sample1"/>
        <value code="sample2"/>
        <value code="sample3"/>
    </enumtype>
</enumtypes>
```

Using an Enumeration

```
HybrisEnumValue result = enumerationService.getEnumerationValue("enumSampleType", "sample1");

HybrisEnumValue result = enumerationService.getEnumerationValue(JobLogLevel._TYPECODE, JobLogLevel.
```

Localizing an Enumeration Value

1. Go to your **extension/resources/localization/ directory**.
2. Open the first locale file you wish to edit.
3. Add: **type. YourComposedTypeCode.YourComposedTypeValueCode.name = locale string**
4. Repeat this for each localization.

```
type.enumSampleType.sample1.name = Beispiel1
                                type.enumSampleType.sample2.name = Beispiel2
                                type.enumSampleType.sample3.name = Beispiel3
```

Setting a Default Value

1. Go to your item definition.
2. Go to the property that holds the enumeration:

```
<defaultvalue>em().getEnumerationValue("enumSampleType", "sample2")</defaultvalue>
```

```
<itemtype code="SampleProduct" extends="Product"
          jaloClass="de.hybris.projects.sample.jalo.product.SampleProduct"
          generate="true"
          autocreate="true"
          >
  <attributes>
    ...
    <attribute qualifier="sampleEnumeration" type="enumSampleType">
      <modifiers optional="false"/>
      <!-- That's the tricky part. Everything between defaultvalue is interpreted as Java code -->
      <defaultvalue>em().getEnumerationValue("enumSampleType", "sample2")</defaultvalue>
    </attribute>
    ...
  </attributes>
```

Use always **em()** as it stands for **enumerationManager.getEnumerationValue(typeCode, enumValueCode)** method expects two parameters. The first one is the enumeration type code. The second one is the value you like to set as default.

Dynamic Against Fixed Enumerations

→ Tip

Add **dynamic="true"** to the definition of your enum if you want it to be modifiable.

In the **items.xml** definition there is the **dynamic** attribute for enumtypes available. An example:

```
<enumtype code="MyEnum" dynamic="false" generate="true" autocreate="false" >
  <value code="new1"/>
  <value code="new2"/>
</enumtype>
```

The value of the **dynamic** attribute is set to false by default. Then no enumeration values can be created at runtime, only at initialization or update via **items.xml**. In this case values are fixed and it is ensured that nobody except the developer can add values.

You can set the **dynamic="true"** to allow adding at runtime. This can be also done in a different extension, when extending an enum of extension A at extension B. The generated class in the ServiceLayer is a SAP Commerce enum, not a Java enum anymore, so that the **valueOf** method can be used for dynamic values.

An attempt to create a new value if **dynamic="false"** yields an **ConsistencyCheckException**.

Related Information

[Type System Documentation](#)

[items.xml](#)

[Transitioning to the ServiceLayer](#)

[ServiceLayer](#)

Working with Catalogs

Platform catalogs hold, structure, and manage products and product information.

Catalogs offer the following features and functionality:

- **Structuring Your Collection with Products and Categories:** Products are the basic elements of each catalog. By grouping them, you can arrange your collection into categories. In order to build a hierarchical product structure, products can be kept in categories. For more information, see [Structuring Your Collection with Products and Categories](#).
- **Catalogs and Catalog Versions:** Categories are held in catalog versions. A catalog version can represent your collection of products at a certain point in time. To enable you to maintain your collection within a basic process, you usually contain two or more catalog versions in an object called a **catalog**. For example, you can have one catalog version for editing the content (the **Staged** version) and you can have another catalog version for propagation as a web shop (the **Online** version). For more information, see [Catalogs and Catalog Versions](#).
- **Managing Multiple Product Catalogs in a Catalog System:** A catalog system is a group of SAP Commerce catalogs. It represents a framework for managing multiple output catalogs enabling you to maintain the varying content of several product catalogs. A catalog system can also include multiple input catalogs. Typically, a catalog system is set up around a master catalog that contains the leading versions of product data. The different catalog contents are synchronized among defined catalog versions, thus reflecting your catalog maintenance process. For more information, see [Managing Multiple Product Catalogs in a Catalog System](#).
- **Synchronizing Catalog Versions:** Catalog versions can be synchronized in a one-directional way in order to update a catalog version based on another one. This enables you to transfer catalog content between the catalog versions that represent steps in your catalog maintenance process. You can synchronize catalog versions, even if they are from the same or different catalogs within your SAP Commerce. For more information, see [Synchronizing Catalog Versions](#).
- **Defining Product Attributes:** Product attributes specify properties of products. They are used to describe a product and to hold related information. A product attribute has a name and a value. For example, the value of the attribute **color** is **blue**; SAP Commerce provides different modeling methods for defining product attributes: typing and classification. For more information, see [Defining Product Attributes](#).
- **Classification Guide:** The classification functionality enables you to define product attributes in a way different to the typing method. Classification-based attributes are called **category features**; they can also sometimes referred to **classification attributes**. Through classification, you can flexibly allocate category features that may change frequently. You can easily define and modify them because you manage them independently of the product type.

Available category features can be organized independently from product catalog structures in separate **classification systems**. Here they can be structured into **classifying categories**. Through such classifying categories, you can assign a category feature to one or multiple product categories used in catalogs. That means that all category features of the classifying categories are available within all products included in the assigned product categories. Therefore, each category feature assigned to a category of a catalog structure is inherited by all subcategories. For more information, see [Classification](#).

Structuring Your Collection with Products and Categories

Products are the basic elements of each catalog. By grouping them, you can arrange your collection in categories. In order to build a hierarchical product structure, products can be kept in categories.

About Products

Products are goods or services that can be offered to a market. Products are typically the basic elements of both your collection and your sales catalogs.

In SAP Commerce, a product is a data structure that represents an item of merchandise, for example a shirt or a camera. It typically includes product attributes such as product name, description, sales unit, and price. Optionally, a product can hold product variants and link to images or other media.

Products are imported from third-party sources such as Enterprise Resource Planning (ERP) systems, Product Information System (PIM) systems, or by file import. Products can be created and edited manually in the Backoffice Framework.

Creating and Nesting Categories

To organize products, you can group them into categories that represent logical groups. This enables you to bring related products together, for example by using a `shirts` category and a `jackets` category. You can compare categories with departments of a traditional store.

Categories can hold other categories, so that you can assemble corresponding categories including their products in a supercategory, also called parent category. Thus, you can nest categories arbitrarily and build up hierarchical category structures.

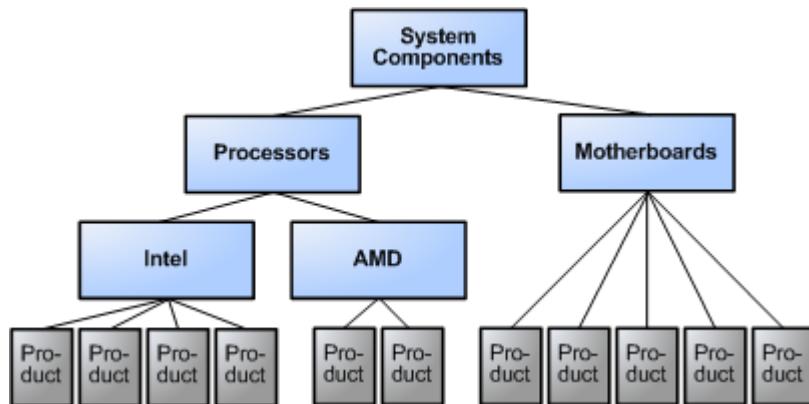


Figure: Sample Category Structure

Structuring Categories

The actual structure of your categories depends on the purpose of the catalog. If you create the category structure for a collection catalog to be propagated in a web shop, the category structure is typically used to provide customer navigation tools of the web shop front-end. Proper orientation and navigation in a web shop should reflect customer expectations and enable the customer to effectively access product information using easy pathways through the web shop.

The screenshot shows a web-based storefront. At the top left is the SAP logo and the text "STORE FOUNDATION". Below the header, a navigation bar has a red button labeled "Computer hardware". The main content area has a sidebar on the left containing a "Categories" tree:

- Peripheral equipment
 - Monitors
- Photography
- System components
 - Processors
 - Intel
 - AMD
 - Motherboards
 - Graphic cards
- Topseller

The main panel shows the results for a search under "System components > Processo". It includes filters for "Speed" (2 GHz) and "FSB" (1,000), and buttons for "Search" and "Reset". A "Compare" button is present above a product image of an AMD Athlon 64 Processor.

Figure: Sample Web Shop navigation based on a category structure.

If you create the category structure for a master catalog of a Product Information System (PIM), you can organize the category structure to group product attributes. This is usually related to the classification of categories. Such classifications enable you to define product attributes for selected categories only. Therefore, you should organize the category structure carefully because you need to consider the product attributes that you want to define by classification afterwards. For more information, see [Classification](#).

[Creating Categories](#)

Categories, like products, are usually imported from third-party sources such as Enterprise Resource Planning (ERP) systems, PIM systems, or by file import. They can also be created and edited manually in Backoffice Framework.

[Controlling the Visibility of a Category](#)

To make a category visible in the storefront, you must explicitly define the user groups or individual users that are allowed to view the category. The visibility of a category also controls the visibility of all subcategories, products, and other objects that are contained in it. In contrast, a category is not visible if it does not contain any visible products.

[Adding and Managing Products in Categories](#)

Categories and products, including their category assignment, are imported from third-party sources, such as from ERP systems, from PIM systems, or by file import. Products can also be assigned to categories manually in the Backoffice Framework.

[Adding Media Items to a Category](#)

Media items, such as product images, are usually referenced by the product. On the **Multimedia** tab of the product editor you can reference images and other multimedia items.

Creating Categories

Categories, like products, are usually imported from third-party sources such as Enterprise Resource Planning (ERP) systems, PIM systems, or by file import. They can also be created and edited manually in Backoffice Framework.

Context

To create a new category, perform the following steps:

Procedure

1. Navigate to **Catalog > Categories**.

2. Click **+ Product** to open the **Create New Category** window.

3. Fill in all mandatory fields. It is recommended not to use the special characters "<" or ">" within the **Category Name** or **Code (Identifier)** fields. The tags result in error when connecting to the CDS Catalog service endpoint.

4. Click **Finish**.

5. In the collection browser, select your newly created category and go to the **Category Structure** tab.

6. In the **Category Structure** tab, insert the category within the target hierarchical category structure.

The screenshot shows the SAP Backoffice Administration Cockpit interface. The title bar reads "[engraving_extraLargeFont] - Electronics Product Catalog : Online". The top navigation bar includes links for GENERAL, CATEGORY STRUCTURE (which is underlined in blue), MULTIMEDIA, and ADMINISTRATION. On the right side of the header are buttons for REFRESH and SAVE. The main content area is divided into three sections: ESSENTIAL, SUPERCATEGORIES, and SUBCATEGORIES AND PRODUCTS. The ESSENTIAL section contains fields for Identifier (engraving_extraLargeFont), Name, and Catalog version (Electronics Product Catalog : Online). The SUPERCATEGORIES section contains a field for Supercategories, which lists "[engraving] - Electronics Product Catalog : Online". The SUBCATEGORIES AND PRODUCTS section contains fields for Subordinate Categories and Included Products, with a list item "DSC-H20 Green [1978440_green] - Electronics Pro...".

- o Select one or more **Supercategories** to group this category together with comparable categories below one or more supercategories.
- o Select one or more **Subordinate Categories** to group other categories below this category; consequently the latter category becomes the supercategory of the selected ones.
- o Select one or more **Included Products** to logically group products in this category.

7. In the **Multimedia tab**, specify a category image and other media relevant for this category. You can also use the **Additional Multimedia Objects** group to manage product images that belong to products contained in this category.

Only SAP Commerce administrators can access the **Administration** tab. It provides editing data including primary key, changes, and copies.

Controlling the Visibility of a Category

To make a category visible in the storefront, you must explicitly define the user groups or individual users that are allowed to view the category. The visibility of a category also controls the visibility of all subcategories, products, and other objects that are contained in it. In contrast, a category is not visible if it does not contain any visible products.

Context

To set the visibility of the category, perform the following steps:

Procedure

1. On the Backoffice Administration Cockpit, access **Catalog > Categories** to display the list of categories.
2. Select the category that you want to update.
3. In the **General** tab, go to the **Category Visibility** section.

4. Update the **Visible to** list by adding one or more user groups or individual users in the fields provided. You can also control the category visibility using personalization rules that check the category settings to verify which users are authorized.

i Note

For more information, see [Catalog Guide](#).

5. Click **Save** to continue action. Otherwise, click **Refresh**.

For more information on controlling visibility, refer to the following topics:

- [Visibility Control](#)
- [Users in Platform](#)

Adding and Managing Products in Categories

Categories and products, including their category assignment, are imported from third-party sources, such as from ERP systems, from PIM systems, or by file import. Products can also be assigned to categories manually in the Backoffice Framework.

Context

→ Tip

You can use the SAP Commerce Product Cockpit to assign products to categories in a graphical way by dragging and dropping. For details see:

- Catalog Perspective of the SAP Commerce Product Cockpit, section *Assign Product Button* of [Catalog Perspective](#)
- [Creating Product Assignments](#)

To add a product to a category, perform the following:

Procedure

1. On the Backoffice Administration Cockpit, access **Catalog** **Products** to display the list of products.
2. Select the product that you want to add to a category.
3. Click **Category System**.

The screenshot shows the SAP Commerce Product Cockpit interface. On the left, there is a navigation sidebar with the following items: Catalog (selected), Catalogs, Catalog Versions, Categories, Products, and Product Variant Types. Below this is a SAVED QUERIES section with the message "No queries". The main content area displays a product detail view for "Cap Blue Tomato BT Snow Trucker Cap black [300441142] - Apparel Product C...". At the top right are buttons for REFRESH and SAVE. Below the product name are buttons for PROPERTIES, ATTRIBUTES, CATEGORY SYSTEM (which is underlined in blue, indicating it is active), PRICES, MULTIMEDIA, VARIANTS, and EXTEN. A horizontal line follows. The next section is titled "CATEGORIES CONTAINING THIS PRODUCT". It has a heading "Supercategories" and a list containing two items: "Blue Tomato [Blue Tomato] - Apparel Product Catalog : Staged" and "Caps [caps] - Apparel Product Catalog : Staged". There is also a "... More" button at the bottom right of this list.

4. In the **Categories Containing This Product** section, define one or more Supercategories in the fields provided.

5. Click **Save** to add the product to the category. Otherwise, click **Refresh**.

Adding Media Items to a Category

Media items, such as product images, are usually referenced by the product. On the **Multimedia** tab of the product editor you can reference images and other multimedia items.

Context

Instead of using the product images suggested in the product objects, you can reference any product image directly in the corresponding product category. In this way, they are referenced separately from the corresponding product, which enables versioning of media items. This ensures that provided media items held separately in categories do not change if the media item referenced in the product object is updated. To accomplish this, you need a media naming strategy that allows you to match products to their related images.

You also can use this method if you want to maintain media items in a separate media maintenance catalog.

i Note

Ensure that you have already added a media item. To add media items, refer to [Creating a Media Item](#).

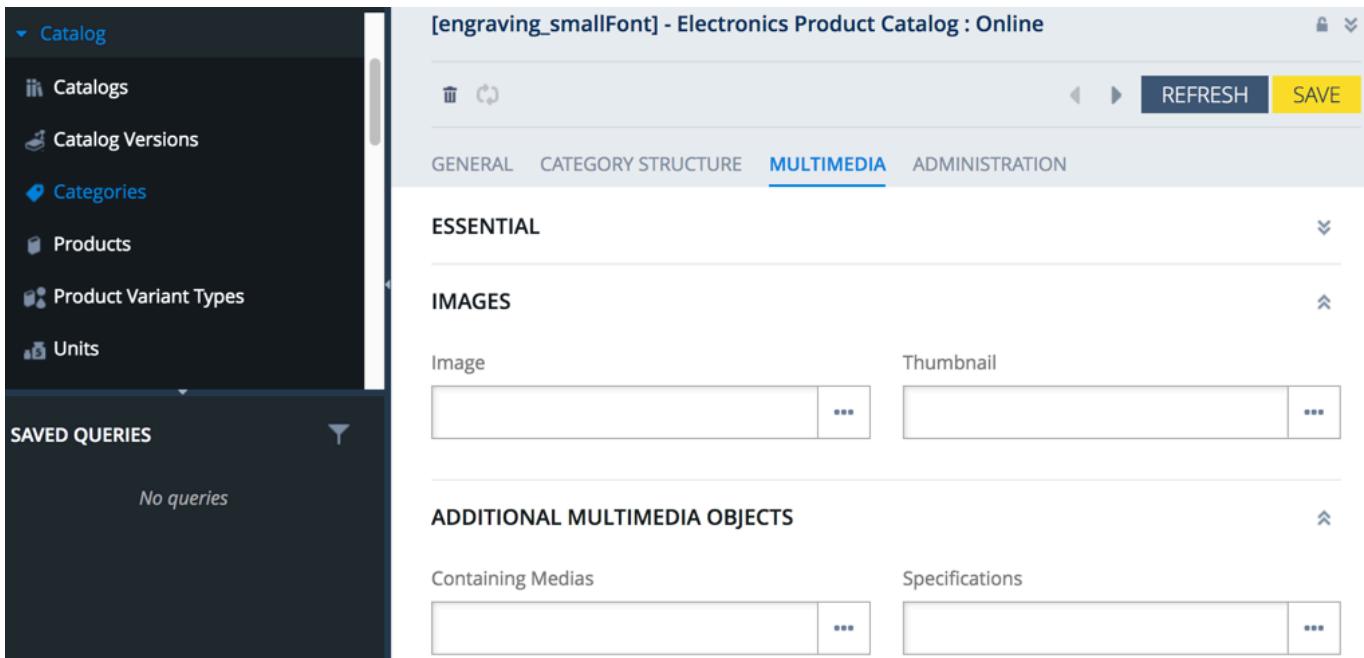
To add a media item to a Category, perform the following procedure:

Procedure

1. On the Backoffice Administration Cockpit, access **Catalog** **Categories** to display the list of categories.

2. Select the category where you want to add the media item.

3. Click the **Multimedia** menu.



4. Define a visual for the category in the **Images** section.

5. Go to the **Additional Multimedia Objects** section to reference media items related to the products in the category.

6. Click **Save** to add the media item. Otherwise, click **Refresh**.

Catalogs and Catalog Versions

Categories are contained in catalog versions. A catalog version can represent the collection of products you offer at a certain point in time. To maintain your collection within a basic process, you typically contain two or more catalog versions in an object called a catalog. For example, while you can have one catalog version for editing the content (the Staged version), you can use another catalog version for propagation as a web shop (the Online version).

Holding Your Category Structure in Catalog Versions

To use a category structure to represent your collection—or parts of it—you contain it in catalog versions. A catalog version is a container for categories that contains products in one or multiple categories.

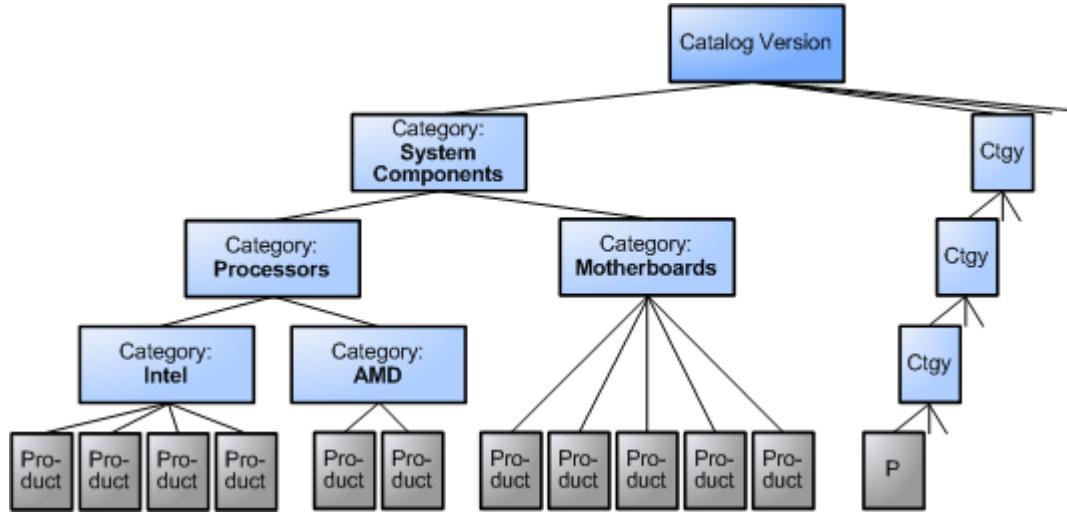


Figure: Sample Structure of a Catalog Version.

Both a SAP Commerce catalog version and a typical printed product catalog, for example, represent the collection of products that you offer at a certain point in time. The differences between the two is that a printed product catalog is a list of sequenced products. In SAP Commerce catalog versions, you can organize products in hierarchical category structures. It allows you to automate content modification. It enables you to maintain more and very specific information types, such as metadata, hidden product data, localizations, views, and restrictions. Instead of paper only, you can propagate your products to multiple channels like web sites, electronic catalogs, mobile, point of sales terminals, or tele-commerce applications. To enable channel-specific propagation, the layout is defined separately from the content.

i Note

A SAP Commerce catalog version item is similar to the concept of a product catalog. Contrary to this concept, a SAP Commerce **catalog** item is a container for multiple catalog versions.

Adding a Category to a Catalog Version

You can add categories to a catalog version by different methods:

- Assign a category to another catalog version in Backoffice. For information, see [Adding a Category to a Catalog Version](#).
- Add a category to a supercategory of another catalog version. For information, see [Structuring Your Collection with Products and Categories](#).
- The category, one of its supercategories, or one of its catalog versions is synchronized. For information about synchronization, see [Synchronizing Catalog Versions](#).

Details

The CatalogVersion item has the rootCategories property set to a read-only dynamic attribute. Invoking the CatalogVersionModel.getRootCategories() method returns all root categories ordered by the category order property. To set root categories for CatalogVersion, edit the Category model. Make sure the categories don't have a parent category, assign them to the catalog version, and save them. Don't use the setRootCategories() method for that purpose.

Controlling Visibility and Access to a Catalog Version

The visibility of a catalog version in a web front end, such as a webshop, is determined by the visibility of the categories included in the catalog version. For details, see [Structuring Your Collection with Products and Categories](#). You can also create a web front-end implementation that checks the language settings of a user group against those of the catalog version.

To control the access of a catalog version in Backoffice, you must specify which user groups or individual users are allowed to access the catalog version for writing or reading.

To set access rights for a catalog version, go to the **Permissions** tab of the catalog version editor. You can edit the lists for **User accounts with write permissions** and **User accounts with read permissions**. For each list, you can add one or more user groups or individual users.

The screenshot shows the SAP Commerce Backoffice interface for managing catalog versions. On the left, a sidebar navigation includes Home, Inbox, System, Catalog (selected), Catalogs, Catalog Versions (selected), Categories, Products, Product Variant Types, Units, Keywords, Classification Systems, Multimedia, and User. Below this is a SAVED QUERIES section. The main content area has a search bar at the top. A table lists catalog versions, with 'default catalog' selected. The table columns are Catalog, Catalog Version, and is active catalog version (Staged, false). Below the table, a message says '0 ITEMS SELECTED'. Under the selected catalog, a sub-section titled 'default catalog : Staged' is shown. At the bottom, a navigation bar includes CATALOG VERSION, CONTENT, CATALOG VERSIONS, BMECAT, PERMISSIONS (selected), and ADMINISTRATION. The PERMISSIONS tab is expanded, showing two sections: 'User accounts with write permission' containing 'Administrator [admin]' and '[employeegroup]', and 'User accounts with read permission' containing 'Administrator [admin]' and '[employeegroup]'. The entire 'PERMISSIONS' section is highlighted with a yellow border.

For more information, see [Visibility Mechanisms \(Checklist\)](#).

Bundling Multiple Catalog Versions in a Catalog

A catalog version represents the collection of products that you offer at a certain point in time. To enable updating your collection without modifying the propagated catalog, you usually use two or more catalog versions bundled in an SAP Commerce object called a catalog. Based on such a catalog/catalog version structure, you can set up a simple maintenance process.

For example, you can have one catalog version for editing the content (the Staged version) and another catalog version for propagation as a webshop (the Online version). Thus, a catalog is not only a container for holding and structuring your products but it also a structure for basic maintenance and propagation processes.

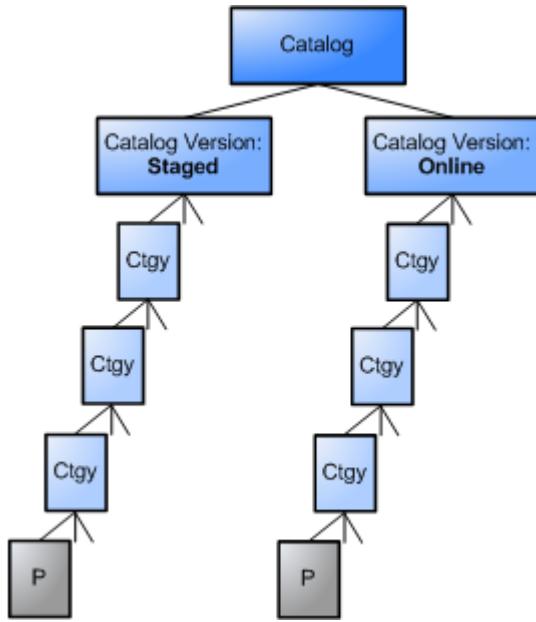


Figure: Typical structure of a catalog including a Staged catalog version and an Online catalog version.

You can organize multiple catalog versions in a catalog, each used for a specific process step. Among different catalog versions, you can synchronize the content. That way, you can establish separate catalog versions for use cases like the following:

- Drafting
- Review and approval
- Seasonal product catalogs, for example summer and winter collections
- Distributed import and export scenarios to be performed on separate catalog versions
- Archiving

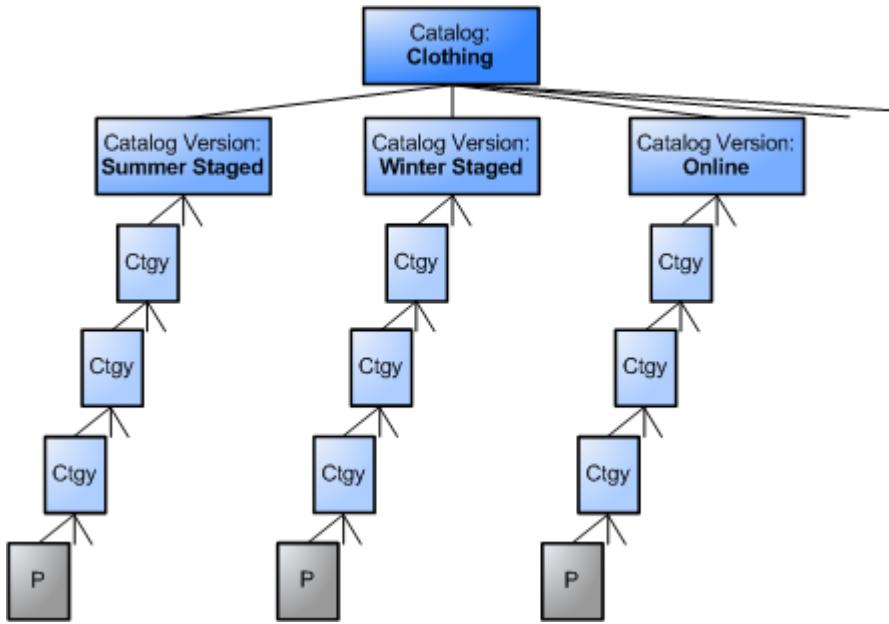


Figure: Sample structure of a catalog with multiple catalog versions.

Only one catalog version within a catalog can be active at a time, which in a web front end typically provides the propagated online version. Consequently, this concept is applicable only for sequential catalog maintenance processes that propagate only one catalog version. For more information, see [Synchronize Catalog Versions](#).

If you set up more complex catalog maintenance processes that allow propagating multiple web front ends simultaneously, you must use multiple catalogs. For more information, see [Managing Multiple Product Catalogs in a Catalog System](#).

To learn how to add a catalog version to a catalog, see [Adding a Catalog Version to a Catalog in Backoffice](#).

Adding a Category to a Catalog Version

Backoffice allows you to add a category to a catalog version.

Context

In this tutorial:

- we use the **Staged** version of the **Default** catalog that is available in SAP Commerce by default
- we use the **myCategory001** category that we created earlier

To add a category to catalog version, log in to Backoffice, and follow these steps:

Procedure

1. In the Backoffice navigation tree, click **Catalog > Catalog Versions**.

The Catalog Versions collection browser opens up with the Staged catalog version listed.

Catalog	Staged	is active catalog version
default catalog	false	
default catalog	true	

2. Click the Staged catalog version.

The **Catalog Versions** collection browser displays on the **CATALOG VERSION** tab.

3. Click **CONTENT** to switch to the **CONTENT** tab.

In the **CONTENT** tab you have access to the **Create a new category** function available from the **Top level categories** field.

Filter Tree entries

SEARCH

Catalog

Catalog... is active catalog version

default catalog Staged false

default catalog Online true

0 ITEMS SELECTED

default catalog : Staged

CATALOG VERSION CONTENT CATALOG VERSIONS BMECAT PERMISSIONS ADM

Catalog

default catalog

Catalog Version

Staged

CATEGORIES

Top level categories

+ Create new Category

4. Click **Create a new category**.

The **Create a new category** wizard opens up.

5. Complete the **Catalog version** and **Identifier** fields and click **DONE**.

Create New Category



PROVIDE ALL MANDATORY FIELDS

Catalog version:

default catalog : Staged

Identifier:

myCategory001

Time created:



CANCEL

DONE

6. Click **SAVE** in the collection browser to save changes in the **Staged** catalog version.

Results

You added a category to a catalog version.

Adding a Catalog Version to a Catalog in Backoffice

Backoffice allows you to add a catalog version to a catalog.

Context

In this tutorial:

- we use the **Default** catalog that is available in SAP Commerce by default
- we create a new catalog version and add it to the **Default** catalog

To add a catalog version to a catalog, log in to Backoffice, and follow these steps:

Procedure

1. In the Backoffice navigation tree, click **Catalog > Catalogs**.

The **Catalogs** collection browser opens up with the **Default** catalog.

Filter Tree entries

SEARCH

1 items

ID	Name
<input checked="" type="checkbox"/> Default	default catalog

0 ITEMS SELECTED

No items selected

2. Click the Default catalog.

The default catalog editor area shows up on the COMMON tab. Notice the Create new Catalog version field.

Filter Tree entries

SEARCH

1 items

ID	Name
<input checked="" type="checkbox"/> Default	default catalog

0 ITEMS SELECTED

default catalog

REFRESH SAVE

COMMON ADMINISTRATION

VERSIONS OF THIS CATALOG

Catalog Versions
default catalog : Online
default catalog : Staged
+ Create new Catalog version

3. Click Create new Catalog version.

The Create new Catalog version wizard opens up.

Create New Catalog version



PROVIDE ALL MANDATORY FIELDS

incl. assurance:

 True False

Catalog:

incl. duty:

 True False

incl. packing:

 True False

is active catalog version:

 True False

Catalog Version:

incl. freight:

 True False

Time created:

CANCEL

DONE

4. Complete the required fields in the wizard.

a. Click ... in the Catalog field and choose your catalog.

Reference Search X

Global Operator: And

Include subtypes

Attribute	Comparator	Value
ID	Starts With	

1 items

ID	Name
Default	default catalog

CANCEL
SELECT (1)

b. Complete the **Catalog Version** field.

c. Click **Done**.

5. Click **SAVE** in the collection browser to save the changes in your catalog.

Results

You added a catalog version to a catalog. You should be able to see your new catalog version listed with the other two:

The screenshot shows the SAP Commerce Catalog Management interface. On the left, a sidebar navigation includes Home, Inbox, System, Catalog (selected), Catalogs, Catalog Versions, Categories, Products, Product Variant Types, Units, Keywords, Classification Systems, Multimedia, User, Order, Price Settings, Internationalization, and Marketing. Below this is a SAVED QUERIES section. The main area has a search bar and a table with columns ID and Name, showing one item: Default (selected) with name default catalog. Below the table, it says 0 ITEMS SELECTED. A modal window titled 'default catalog' is open, showing tabs for COMMON (selected) and ADMINISTRATION. Under COMMON, it lists 'VERSIONS OF THIS CATALOG' with entries: default catalog : Online, default catalog : Staged (selected), and default catalog : Staged2 (highlighted with a yellow box). There is also a '+ Create new Catalog version' button. The ADMINISTRATION tab is visible but empty.

Managing Multiple Product Catalogs in a Catalog System

A catalog system is a group of SAP Commerce catalogs. It represents a framework for managing multiple output catalogs enabling you to maintain the varying content of several product catalogs. In addition, a catalog system can handle multiple input catalogs. A catalog system is set up around a master catalog holding the leading versions of product data. The different catalog contents are synchronized among defined catalog versions, thus reflecting your catalog maintenance process.

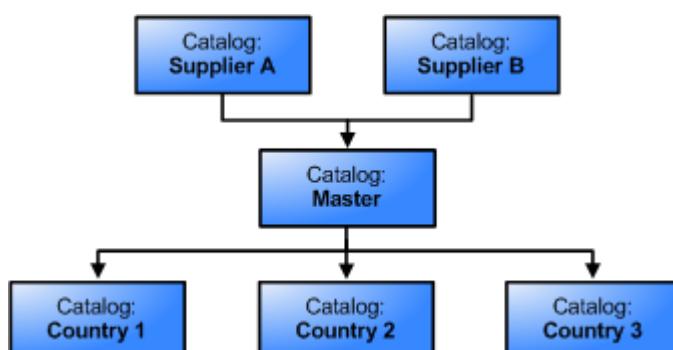


Figure: Sample catalog system consisting of two input catalogs before a master catalog and three output catalogs afterwards.

Within a SAP Commerce catalog, only one catalog version can be active, which in a web front end typically provides the propagated online version. Consequently, this concept is applicable only for sequential catalog maintenance processes that propagate at most one catalog version. If you wish to set up more complex catalog maintenance processes that allow propagating multiple web front ends simultaneously, you have to use multiple catalogs. The same applies for working with multiple input catalogs.

Defining Your Catalog Maintenance Process

Before setting up a catalog system, SAP Commerce recommends that you define your catalog maintenance process. This is important because a catalog system should reflect your organizational structures and responsibilities.

For example, you may want to include a country-specific web page owned by a foreign subsidiary. Typically, you manage such a page in a separate catalog, which receives the basic collection data from a master catalog to be edited and approved locally. Such a country-specific catalog can have its own staged and online versions for editing and propagation. Specific access rights ensure writing permissions for the local team, however several headquarter's representatives may have reading or writing access for supporting tasks.

When sketching your catalog system, SAP Commerce recommends that you define the following elements:

- A master catalog.
- An arbitrary number of separate catalogs that each contain more or less varying content that is actively managed, for example:
 - A catalog is used for separating multiple import sources: If you receive product data from different sources you can use a catalog for each source. For example, you can maintain separate input catalogs for different collections, suppliers, content types, or external source systems like Enterprise Resource Planning (ERP) systems, PIM systems, or file import. That way you can separate the data and corresponding process steps before merging in a master catalog.
 - At least one output product catalog that receives its content from a master catalog: Typically, each active product catalog is based on a separate output catalog to cover different languages, countries, brands, customer groups, output channels, or other criteria.
 - The catalog represents a process step because its content is actively managed, for example by modification of content or synchronization with other catalogs.
- Catalog versions for subordinate process steps that can be organized within a catalog because only one of the catalog versions needs to be active. For example, you can organize year-specific product catalogs in catalog versions. For more information, see [Catalogs and Catalog Versions](#).
- Synchronizations between two catalog versions, indicating the data flow among catalogs.

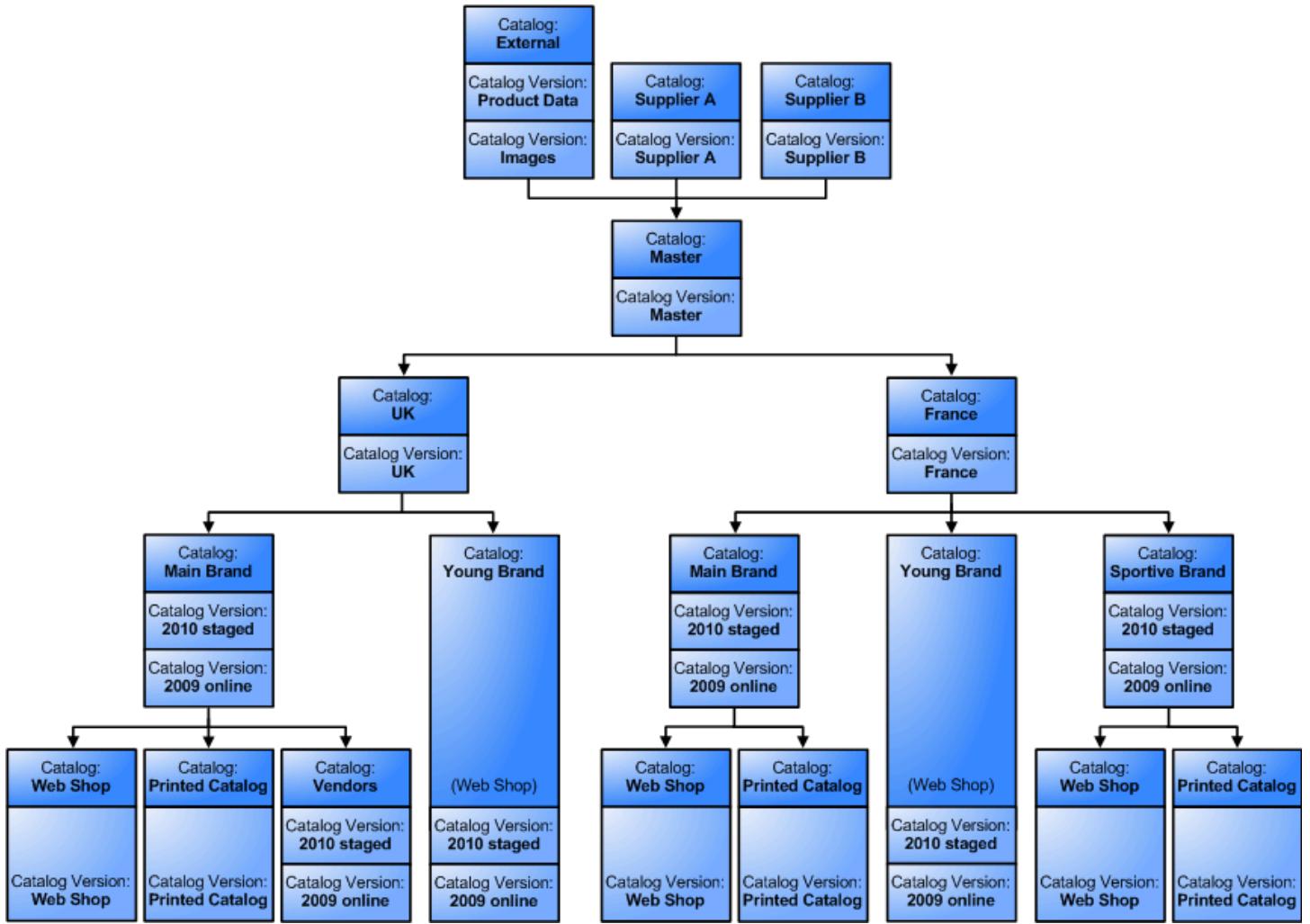


Figure: Example of a complex catalog system. It consists of multiple input catalogs (top), a master catalog, and two country-specific catalog subsystems (for United Kingdom and France). The example assumes that most output catalogs (bottom) are actively managed by Brand Managers. However, the **Vendors** output catalog has an additional process step for audience-specific modifications. Arrows indicate synchronizations between the subordinate catalog versions.

Setting Up a Catalog System

To set up a catalog system, you simply create the required catalogs and connect them by synchronizing jobs between catalog versions. You organize the catalogs involved in such a way so as to establish a synchronization process from input to output. Synchronization operations among catalogs are realized between the contained catalog versions.

Synchronizing Catalog Versions

Catalog versions can be synchronized in a one-directional way for updating a catalog version according to another. This way you can transfer catalog content between the catalog versions which represent steps in your catalog maintenance process. You can synchronize catalog versions, even if they are of the same or different catalogs within your SAP Commerce. Synchronizing operations are executed between catalog versions.

About Synchronizing

A synchronizing operation copies specified content from a source catalog version to a target catalog version, for example a **staged** to an **online** catalog version. That means, the items referenced by the target catalog versions are overwritten to match the items referenced by the source catalog version.

Example of a product price in different catalog versions:

Time:	Before synchronizing	Before synchronizing	After synchronizing
Catalog version:	Source	Target	Target
Value:	20.50 EUR	17.99 EUR	20.50 EUR

A synchronizing operation can be applied to an entire catalog version, but also to selected categories or products. It only works in a one-directional manner, however you can apply two synchronizing operations for establishing bi-directional updating procedures.

By default, only items referenced by the source catalog version are synchronized, and those items of the target catalog version that are not referenced are not affected. For example, let's say the target catalog version consists of the contents of two other catalogs, **clothes** and **hardware**. If the **hardware** catalog is synchronized, the **clothes** catalog-related part of the target catalog version remains unchanged.

Example of product prices for a hardware product and a clothes product in different catalog versions:

Time:	Before synchronizing	Before synchronizing	After synchronizing
Catalog version:	Source	Target	Target
Value Hardware:	20.50 EUR	17.99 EUR	20.50 EUR
Value Clothes:	139.00 EUR	109.00 EUR	109.00 EUR

In contrast, you can configure synchronization settings for removing existing products from the target catalog version that do not exist in the source catalog version. Such synchronization settings can define rules on how product data is copied to the target catalog version. For example, you can determine whether to remove existing products in the target catalog version that do not exist in the source catalog version. Alternatively, only products that exist in both the source and the target catalog version are synchronized. Another option of synchronization settings refers to the languages of products attributed to be synchronized.

A synchronizing operation is held in a synchronization object that references the source and the target object. You can launch a synchronization manually or automatically at a given time through a [The Cronjob Service](#).

To access a **Synchronization** object, go either to its source or target catalog version. On the **Versions** tab, you find all related **Synchronizations** listed below the **Basic Settings** group. Right-click one and open it by choosing an **Edit** command on the context menu. For more information, see [Synchronizing Catalogs](#).

Synchronizing Operations in Catalog Versions within a Catalog

There are specific use cases for synchronizing catalog versions that are part of a single catalog.

Seasonal Switch

For example, you can use this for switching from one catalog version to another catalog version: Let's say, a catalog for clothes has five catalog versions:

- Spring
- Summer
- Fall
- Winter

- Online

The **Online** catalog version is in use for the web shop front end. By synchronizing from **Summer** to **Online**, you can switch the product line in the shop frontend to the summer edition. Synchronizing from **Fall** to **Online** replaces the summer product line with the fall product line, and so on.

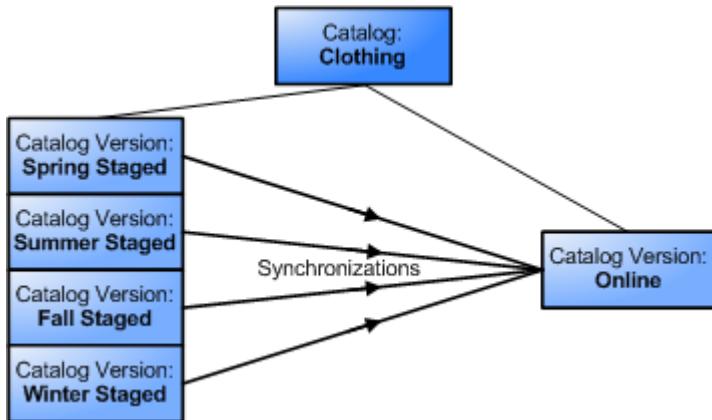


Figure: Example of a catalog with a synchronizing operation for switching seasonal collections.

Separated Input Catalog Versions for Different Content Types

To realize distributed import scenarios, you can use separate catalog versions, for example for different content types. Let's say, you receive your product data from different sources:

- Product names and descriptions are loaded from CSV files provided by a supplier.
- Product images are loaded from a file system maintained by an advertising agency.
- Product prices are retrieved from an Enterprise Resource Planning (ERP) system.

Each source provides a type of product data with each data set related to a defined product. Each product data type is loaded into a different catalog version to enable different loading and clearing routines. These different catalog versions are consolidated by synchronizing operations to a master catalog representing a single source of truth.

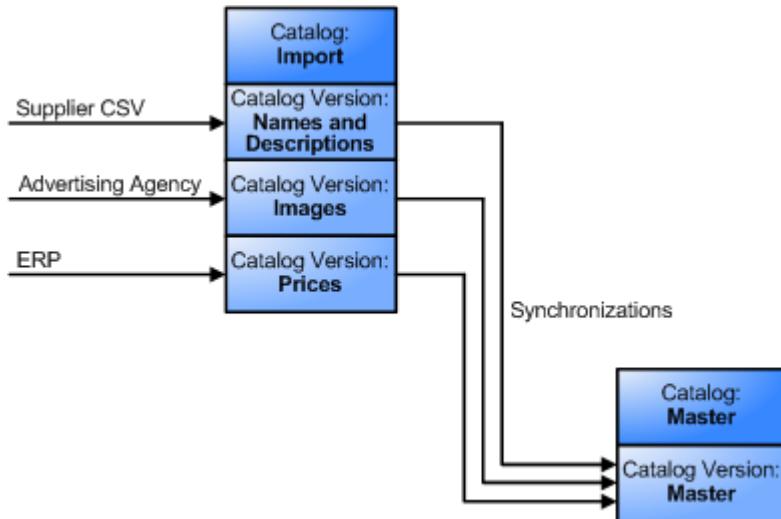


Figure: Example of an import catalog from three source providing different content types.

Defining Product Attributes

Product attributes specify properties of products. They are used to describe a product and to hold related information. A product attribute has a name and a value. For example, the value of the attribute **color** is **blue**. The SAP Commerce provides different modeling methods for defining product attributes: typing and classification.

Selecting the Method

When starting a new SAP Commerce PCM (Product Content Management) or eCommerce project, a consultant or developer investigates the existing product attributes and sketches a modeling strategy for defining product attributes in SAP Commerce. Most attributes can be defined the same as in an existing customer system. For some specific or new attributes, the consultant decides on the best method of attribute definition with regard to the customer requirements and the business context.

The attribute modeling method has no direct impact on the end user of a web shop front end. But, it has important consequences for relating any business logic, data base loading, and the ease of set-up and modification of attributes. Thus, the definition method for an attribute can be worth a careful evaluation.

Comparison of Typing and Classification

This table lists important criteria for comparing methods of product attribute definition.

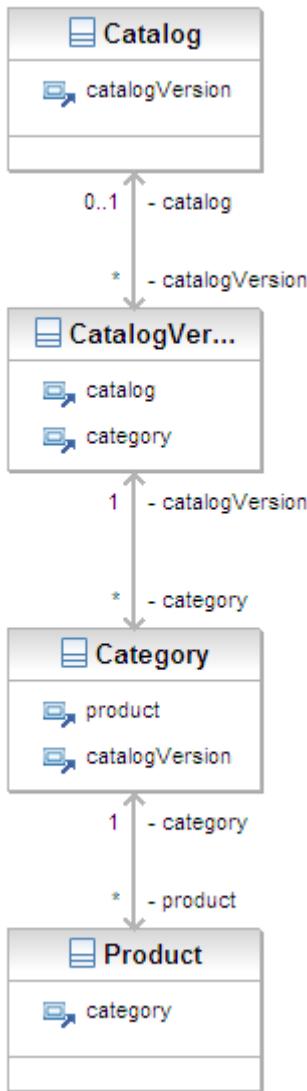
Criteria	Typing	Classification
Character	Basic method	Advanced method
Use Case	Recommended for static product attributes that are used in business logic like calculations. Frequently used for PCM projects.	Recommended for changeable product attributes that are not to be used in business logic. Frequently used for eCommerce projects.
Examples	Article number is unchangeable. Sales unit to be used for calculations of shipping costs.	Resolution is merely a data sheet information gathered from a supplier to be transferred for display. Microphone : The existence of a microphone being part of a product.
Applicable to	All data types.	Products only.
Scope	All instances of a type, for example for all instances of a subtype.	Easy and dynamic application for selected products or categories, for example.
Requirement	None	1. A category tree defining a product structure. 2. A classification system providing available product attributes that can be assigned to categories and products.
Performance Implication	Less load on database.	More load on database.
Typical User	Developer	Product/Purchasing Manager Marketing Manager
Modification	Complex adaptation to new requirements by coding.	Easy adaptation in the platformbackoffice extension, or by CSVfile import.

Criteria	Typing	Classification
Outcome	Type attribute	Category feature, also referred to as classification attribute.
Display in Backoffice	Any position on Product tabs can be configured.	By default on the Product Attribute tab. Modification of display more complex.
Technique	A type attribute typically is defined either by runtime types (not recommended!) or in an extension items.xml file.	A category feature is defined in Backoffice as part of a classifying category. Such a classifying category can be assigned to several products. The assignment to a product category with the consequence that all products of the category and its subcategories hold the category feature is more efficient.
Restriction	No multi-inheritance is possible. Each type attribute has to be configured separately.	Multiple assignment is possible. Through classifying categories, you can assign a category feature to one or multiple product categories.
Read More	The Type System : See section <i>Adding New Attributes to a Type</i> .	Classification

Catalog Guide

The catalog extension provides catalog functionality for holding, structuring, and managing products and product information.

Type Overview



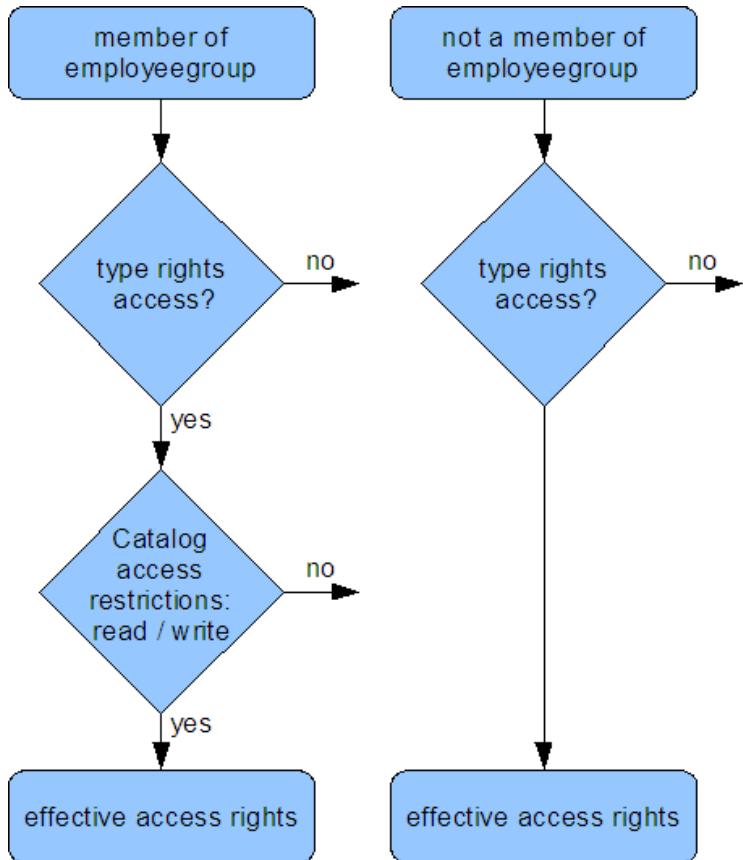
SAP Commerce represents catalogs by a type structure as shown in the following diagram: a catalog can contain a number of catalog version objects, each of which can contain a number of category objects. Category objects can contain catalog objects, product objects or classification objects.

SAP Commerce Type	Typical	Description
Catalog	<ul style="list-style-type: none"> • Clothes catalog • Hardware catalog 	General grouping of products.
CatalogVersion	<ul style="list-style-type: none"> • Summer season 2008 • Summer season 2007 • Winter season 2008 • Winter season 2007 	Narrowing down of the catalog scope.
Category	<ul style="list-style-type: none"> • Jeans • Jackets • Men's fashion • Women's fashion • CPU 	Rather close-up grouping of products.

SAP Commerce Type	Typical	Description
	<ul style="list-style-type: none"> • Graphics cards • Harddisks 	

Catalog Visibility in Backoffice

Catalog View and Editing Rights



In addition to the platform type system access rights, the catalog extension runs a second evaluation of restrictions on catalog access rights. These restrictions do not interfere with the access of catalogs or catalog versions. They are meant to restrict employee rights to manipulate a catalog version content, such as products, categories, prices, others. When you initialize the system and have it create sample data, these rights are created automatically for the **employeegroup** usergroup . You can check the restrictions by looking up the usergroup. If you add new employees to the system, they will be put into the **employeegroup** by default.

The catalog extension uses personalization rules (search restrictions) to evaluate catalog-editing rights. Since only the **employeegroup** usergroup has these restrictions by default, a user must be in that usergroup or one of its subgroups to have those rights.

Thus, catalog and catalog version specific data (such as synchronizations or active version settings) as seen in the Catalog Editor or Catalog Version Editor is usually governed by the settings provided in the access restrictions of the relevant types (here: catalog and catalog version). In contrast, the accessibility of specific catalog version contents in terms of read/write access differentiation is administrated through (direct or indirect) membership in **employeegroup**.

i Note

A user who is not in the employee group or one of its subgroups can bypass this restriction system. A user who bypasses the restrictions, however, may have access rights to catalog versions in the web application that the user is not intended to have. In this way, you can allow users to browse catalogs you do not want them to see.

Catalog Visibility in a Web Application

This section discusses how to set catalogs to be visible in a web application.

Visibility of Products for Customers

A catalog is a list of products that are available, along with product prices (among other attributes). The catalog extension also allows you to block specified customers from viewing specific products, even if the products are generally available. This functionality is called visibility.

Whether the product is visible to customers or not is determined in five independent steps. If any of the steps fails, the product becomes not visible to that respective customer-- even if all other steps are successful. The following graphic shows the five steps:

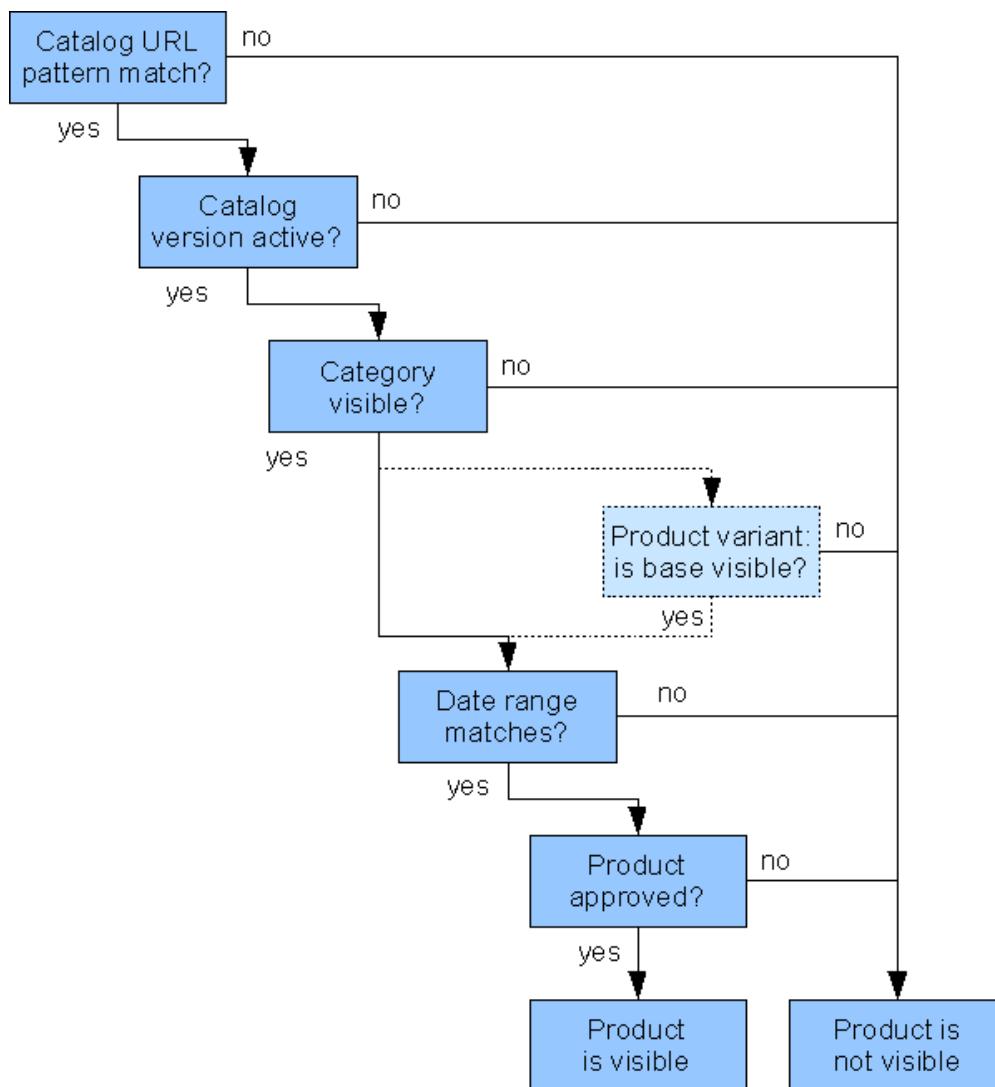


Fig. Visibility Checks for Product

The steps are:

1. Does the catalog URL pattern match?

2. Is the catalog version active?

3. Is the category visible?

Products are usually put into categories. If a category is not visible to a customer or customer group, the products contained in the category will also not be visible.

4. Does the date range match?

If a product is visible or not can also depend on the current date. For example, selling ice hockey equipment in June may not be very successful. Therefore, you can limit a product visibility to a certain date range. The product is not visible before the start of that date range or after its end.

5. Is the product approved?

A product is always in one of three possible states: check, unapproved, and approved. An approved product is visible if all other checks pass, a product set to check or unapproved is not.

i Note

Steps 1 and 2 are only checked at the very start of the session. The three remaining steps are checked on each customer mouse click. Another difference is that the catalog extension itself deals with steps 1 and 2, while the other steps use Platform restrictions.

Matching Catalog URL Patterns

Using the SAP Commerce, you can have several shops running on one server. To keep the shops apart, you will probably use different URLs (like `/computer`, `/skateboard`, etc). URLs can use regular expressions according to the [POSIX Basic Regular expression syntax](#). If the shop's URL pattern does not match the catalog URL pattern, the customer is trying to access the wrong shop and, consequently, will not see that catalog products.

The easiest URL pattern you can set up is `.*`; This pattern matches any URL (even with protocols that do not exist).

This example of a URL pattern `https?://+/bluestore($|/.*)` matches all URLs that conform to all the following conditions:

- Have `http://` or `https://` as their protocol (`http` with an optional (?) `s`, followed by `://`)
- Have at least one character but the slash after the protocol (+; the circumflex ^ negates the term, the plus sign + indicates at least one occurrence)
- Contain the term `/bluestore`
- End with any of the following terms
 - End of line (indicated by the dollar symbol \$; for example: `/bluestore`)
 - `/` plus any string (. *) (such as `/bluestore/mypage.html`)

You may set up several URL patterns for one catalog. That way, you can set up one catalog to be used by several web shops at different URLs, for example.

Catalog Version Active

You may have several catalog versions available in your platform, but only one may be active at a time. The other ones are stored in the platform and can be edited, but they are not currently active. Such staging of catalogs (that is, having one or more inactive versions) is useful if your product portfolio changes over the year, such as a winter and summer collection.

Category Visibility

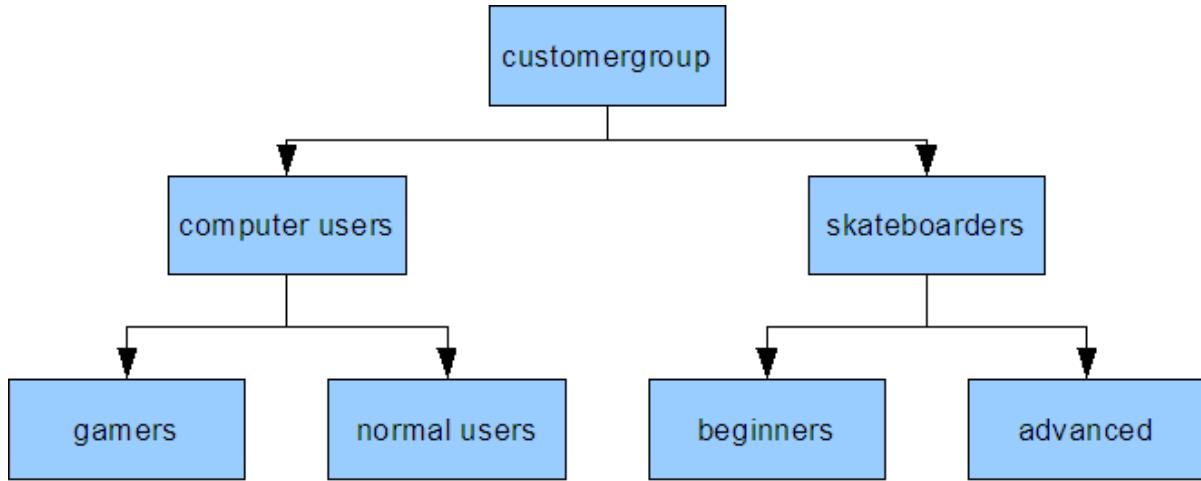
If a product category is visible to a specific customer in a catalog depends on whether the category itself is visible to the customer and the user group the customer belongs to. If the customer or the group the customer belongs to is not explicitly allowed to view a specific category in the platform, the category will not be visible for the customer or the customer's group in the shop.

The screenshot shows the SAP Commerce Category Management interface. On the left, a sidebar navigation includes Home, Inbox, System, Catalog (selected), Catalogs, Catalog Versions, Categories (selected), Products, Product Variant Types, Units, Keywords, Classification Systems, Multimedia, User, Companies (selected), and SAVED QUERIES. The main area displays a list of categories with columns for Identifier, Sync status, and Catalog version. The 'root' category is selected, showing a red error icon and the message 'default catalog : Online'. Below this, the 'root' category details are shown under '[root] - default catalog : Online'. The 'GENERAL' tab is selected, showing the Catalog version as 'default catalog : Online'. Under 'CATEGORY VISIBILITY', the 'Visible to' field contains '[customergroup]' and has a yellow border. The 'DESCRIPTION' tab shows fields for Description and Keywords.

Categories inherit access rights to their children. If you set the root category to be visible to a certain customer, that customer will be able to view not only the root category itself, but also all subcategories of it. On the other hand, user groups also inherit. If a usergroup has the rights to view a certain category, any subgroup of that usergroup will be able to view that category as well.

For this reason, SAP Commerce recommends that you put all customers into a customergroup or into one of its subgroups so that you can maintain the visibility rights more easily. If you do not work with usergroups and their visibility, you will have to maintain the visibility rights on a per user basis, which is more work and more complicated.

If all customers belong to a customergroup directly, all customers will have access to all categories that the customergroup has access to. However, if you want to have two sorts of customers with distinct category access, for example, computer users and skateboarders, you must work with subgroups of customergroup. It is not possible to explicitly deny access to users, only to allow it.



In the image above, an example customer group hierarchy is shown. The customergroup is divided into two groups, computer users and skateboarders, each of which is split into two additional subgroups: gamers and normal users, and beginners and advanced, respectively. The following table gives you an overview of who would be able to view a category if that category **Visible to** value were set to the respective value:

Value of Visible to	Category is Visible to Members of
customergroup	all
computer users	computer users, gamers, normal users
skateboarders	skateboarders, beginners, advanced
normal users	normal users
computer users, advanced	computer users, gamers, normal users, advanced
gamers, normal users	gamers, normal users

i Note

There is a difference between adding computer users and adding both gamers and normal users to the **Visible to** value. If there are any customers who belong to computer users, but not to gamers or normal users, these would be included if you add computer users, but not if you add only its subgroups gamers and normal users.

It is also possible to explicitly add customers to the **Visible to** setting, as shown in the following image. This does not have any side effects, but it makes sure that these customers will have access to that category even if the usergroup they belong to is removed from the list (in that case, their access rights to the category would be gone normally).

Identifier	Name	Sync status	Catalog version
✓ root		red	default catalog : Online

0 ITEMS SELECTED

[root] - default catalog : Online

REFRESH SAVE

GENERAL CATEGORY STRUCTURE MULTIMEDIA ADMINISTRATION

Catalog version

default catalog : Online

CATEGORY VISIBILITY

Visible to
[customergroup]
eckstein-brinkmann [eckstein-brinkmann]
kotal [kotal]
schneider [schneider]

In the above image, the customers eckstein-brinkmann, kotal, and schneider would be able to see category root (and all its subcategories) even if customergroup would be removed from the visibility list.

This inheritance implies another thing as well, however. If you set a category to be visible for a customer or a group of customers, all subcategories of that category will be visible to that customer or group of customers as well. In other words, in the above image, all the customers listed explicitly, plus the members of customergroup, would be able to see the root category and all its subcategories, that is, in the case of root: all categories. If you want a certain category to be visible to some customers only, you need to add those customers explicitly to that specific category. If you start setting the visibility of a category above the one you wish to set, those settings will be inherited downwards and may cause categories to be visible to users that you do not want to be visible to them.

Matching DateRange

Product life cycles have a beginning and an end, that is, products start and cease to be available. Often, there are fixed dates at which a product has reached its market maturity and when it will no longer be sold. The catalog extension supports this. You may set a date by which a product will be visible in your shop, and/or a date when it will no longer be visible. A blank field means that the product will be visible until that date (on-line to) or from that date onwards (on-line from), respectively. If you assign a value to both fields, the product will be available only in the time span between both dates.

i Note

Dates start by 0:00:00 hours AM on the respective day, and end by 11:59:59 PM (0:00:00 to 23:59:59 hours, respectively). It is not possible to have a product visibility start or end at some time within that day, like 11:00 AM.

Approving Product

A product that is a part of the catalog is in one of three possible approval states: approved, check, and unapproved.

Only products in the approved state are visible, the two others indicate that the product is not in a state to be synchronized and consequently to be offered for sale.

You can change a product approval state in its editor. Go to **Catalog > Products**, search for your product by its name or browse to it. Click it to open up its editor and change the approval state in the **ATTRIBUTES** tab.

You can also perform bulk approval of your products using the Bulk Edit or Bulk Approval actions.

For more information, [Bulk Editing in Backoffice](#), [Bulk Edit Action](#), and [Enumeration Action](#).

The screenshot shows the SAP Backoffice interface for managing products. On the left, a sidebar navigation menu includes Home, Inbox, System, Catalog (selected), Catalogs, Catalog Versions, Categories, Products (selected), Product Variant Types, Units, Keywords, Classification Systems, Multimedia, User, and Companies. Under SAVED QUERIES, it says 'No queries'.

The main area displays a list of products in the 'Catalog' view. Two products are listed: 'myProduct002' and 'myProduct001'. Both products have their Article Number, Identifier, Status (red and yellow icons), and Catalog version (default catalog : Staged) displayed. Below the list, it says '0 ITEMS SELECTED'.

For 'myProduct002', a detailed view is shown in the bottom half of the screen. The 'ATTRIBUTES' tab is selected. In the 'ESSENTIAL' section, the 'Approval' field is highlighted with a yellow border and contains the value 'check'. Other fields in this section include Article Number (myProduct002), Identifier (myProduct002), and Catalog version (default catalog : Staged).

Below the essential attributes, there is a section for 'CLASSIFICATION ATTRIBUTES' which states 'No classification attributes available'.

Using the Catalog Extension with Web Application

The catalog extension relies on [Restrictions](#) for visibility and access settings. Without set restrictions, the catalog extension does not work properly and users may be allowed access to catalog versions they are not intended to see, and vice versa.

This is not critical with the Backoffice, where all users must log in and have their restrictions set based on their account. However, in a web application, such as a web shop, you need a way of handling restrictions related to the catalog extension without requiring users to log in. You can use Platform filters to do this. They provide flexibility. You can use the following filters to handle catalog version activation:

- **SimpleCatalogVersionActivationFilter**
- **DynamicCatalogVersionActivationFilter**

For more information, see [Platform Filters](#).

The **SimpleCatalogVersionActivationFilter** filter assures that the configured catalog versions are set as session catalog versions. The **DynamicCatalogVersionActivationFilter** filter takes care of activating the catalog versions at runtime. Below is an example that shows how to add the **SimpleCatalogVersionActivationFilter** to the SAP Commerce filters chain used in your Web application. It presents the current implementation in the `springmvcstore` extension as shown in the following examples from the `springmvcstore-web-spring.xml` file:

```
<bean id="springMvcStoreFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterChain"
      <constructor-arg>
        <list>
          <ref bean="log4jFilter"/>
          <ref bean="dynamicTenantActivationFilter"/>
          <ref bean="springMvcStoreRedirectFilter"/>
          <ref bean="sessionFilter"/>
          <ref bean="springMvcStoreDataSourceSwitchingFilter"/>
          <ref bean="springMvcStoreCatalogVersionActivationFilter"/>
        </list>
      </constructor-arg>
    </bean>

<bean id="springMvcStoreCatalogVersionActivationFilter" class="de.hybris.platform.servicelayer.v...
  <property name="catalogVersionService" ref="catalogVersionService"/>
  <!-- <property name="onlySetOnSale" value="true"/> --> <!-- true by default, set to
  <!-- property catalog.versions.default will be read from properties file -->
  <aop:scoped-proxy/>
</bean>

<bean id="springMvcStoreRedirectFilter" class="de.hybris.platform.servicelayer.web.RedirectWhen...
  <constructor-arg>
    <value><!-- nothing - redirect to default maintenance page --></value>
  </constructor-arg>
  <constructor-arg>
    <list>
      <value>login</value>
      <value>static</value>
    </list>
  </constructor-arg>
</bean>

<bean id="springMvcStoreDataSourceSwitchingFilter" class="de.hybris.platform.servicelayer.web.D...
</bean>
```

For more information about extending the SAP Commerce filters used by your Web application, see the **Configuring Existing Filters** section of [Platform Filters](#).

Setting a Default Catalog During System Initialization or Update

During a SAP Commerce initialization or update, the catalog manager ensures that there is a default catalog. The catalog manager `createEssentialData(...)` method retrieves the catalog, which is set as default.

If no catalog is set as default, the catalog manager creates a new catalog called default and creates an on-line (Online) and an off-line (**Staged**) catalog version within that new catalog.

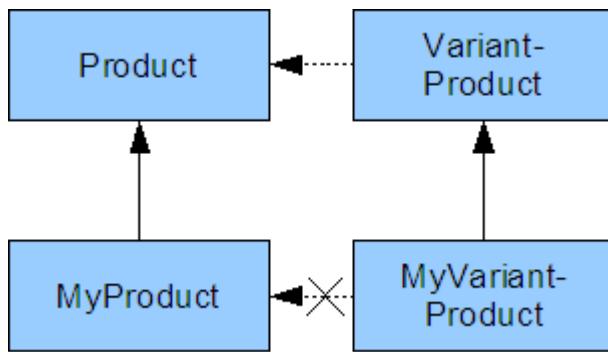
Modeling Product Variants

When working with products, you need to model product variants. SAP Commerce allows several approaches to implement such a model.

Variants are products that differ in some aspect from one another, but are based on the same basic model. An example for variants is color and size for t-shirts. The base product is a t-shirt, the variants are a red t-shirt or a blue t-shirt, a t-shirt in size L, a t-shirt in size XL, and so on.

Using a Subtype of Product and a Subtype of VariantProduct

The intuitive approach of simply deriving from both types directly (as shown in the diagram) does have a major limitation (indicated by the crossed-through line in the diagram).



Since **Product** and **VariantProduct** have a direct line of inheritance (indicated by solid arrows), attributes from **Product** will appear on **MyVariantProduct**. However, since **MyProduct** and **MyVariantProduct** do not have a direct line of inheritance, attributes from **MyProduct** will not be available directly on **MyVariantProduct**. By consequence, you will have to manually ensure that attributes of the **MyProduct** type are available on the **MyVariantProduct**, and vice versa.

There are two workarounds.

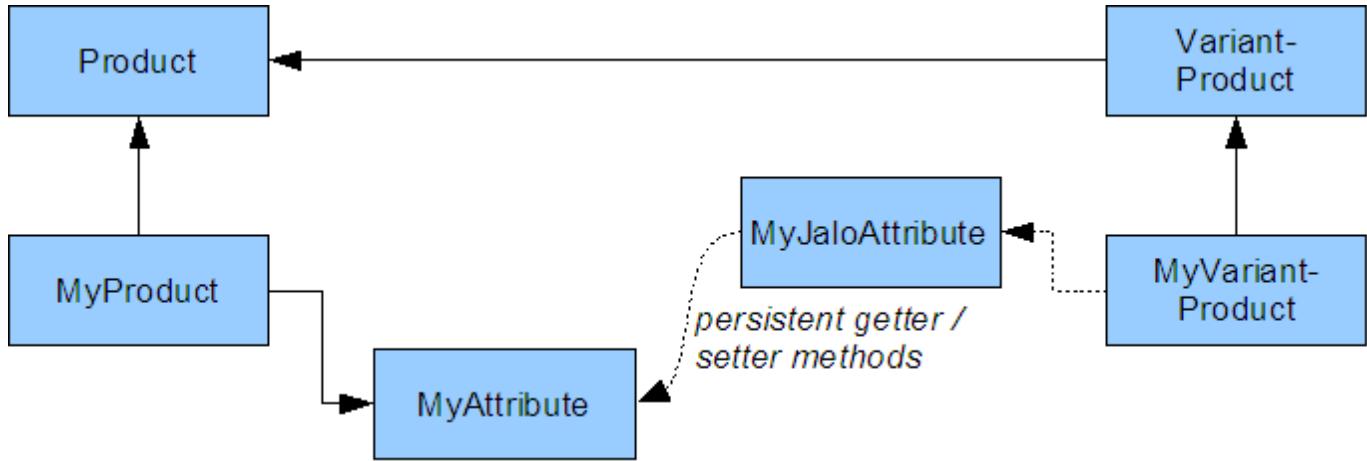
- Define the attributes on the **Product** type itself, for example:

```

<itemtype code="Product" generate="false" autocreate="false">
    <attributes>
        <attribute qualifier="myAttribute" type="java.lang.String">
            <persistence type="property" />
        </attribute>
    </attributes>
</itemtype>
  
```

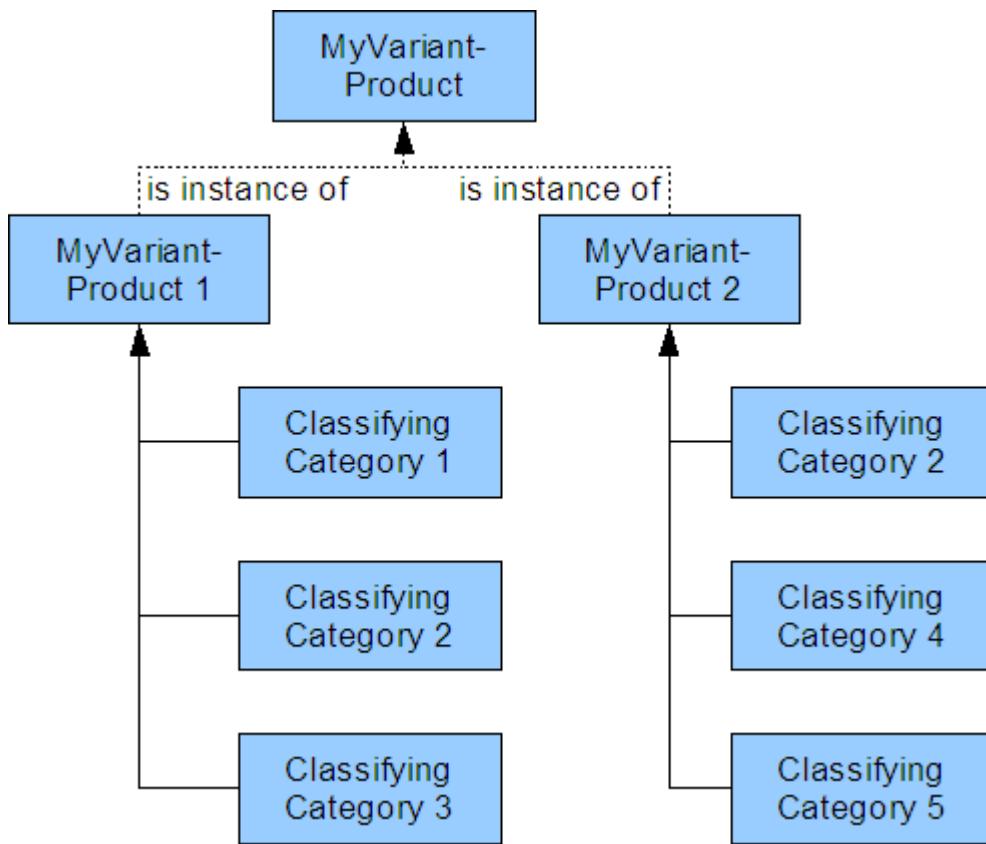
As both **MyProduct** and **MyVariantProduct** inherit from **Product**, you would have the attributes available for both **MyProduct** and **MyVariantProduct**.

- Use jalo-only attributes on either type. By implicitly using the getter and setter methods of the **MyProduct** type, for example, the **MyVariantProduct** type could pretend having the same attributes. Please refer to the [Jalo-only Attributes](#) for details.



Using a Subtype of VariantProduct in Combination with Classification

Another approach to modeling product variants is by using SAP Commerce's classification functionality. This approach uses a subtype of **VariantProduct** (**MyVariantProduct**, in the diagram) that contains only a relation to the variant product type to be used (the relation is indicated by the dotted lines in the diagram). The **VariantProducts** are referenced by the base product to indicate that they are variants of the base product. In addition, the variant products have classification classes assigned to them to make variant-specific attributes available to them.



You will need to use a subtype of **VariantProduct** because **VariantProduct** is an abstract type and cannot be instantiated.

You can set classification attributes to be visible for base products or variant products only. That way, you can make sure that only **MyVariantProduct** has visible classification attributes.

The advantage of this approach is that you can use inheritance of classification classes and therefore can have (classification-based) attributes inherited from the base product.

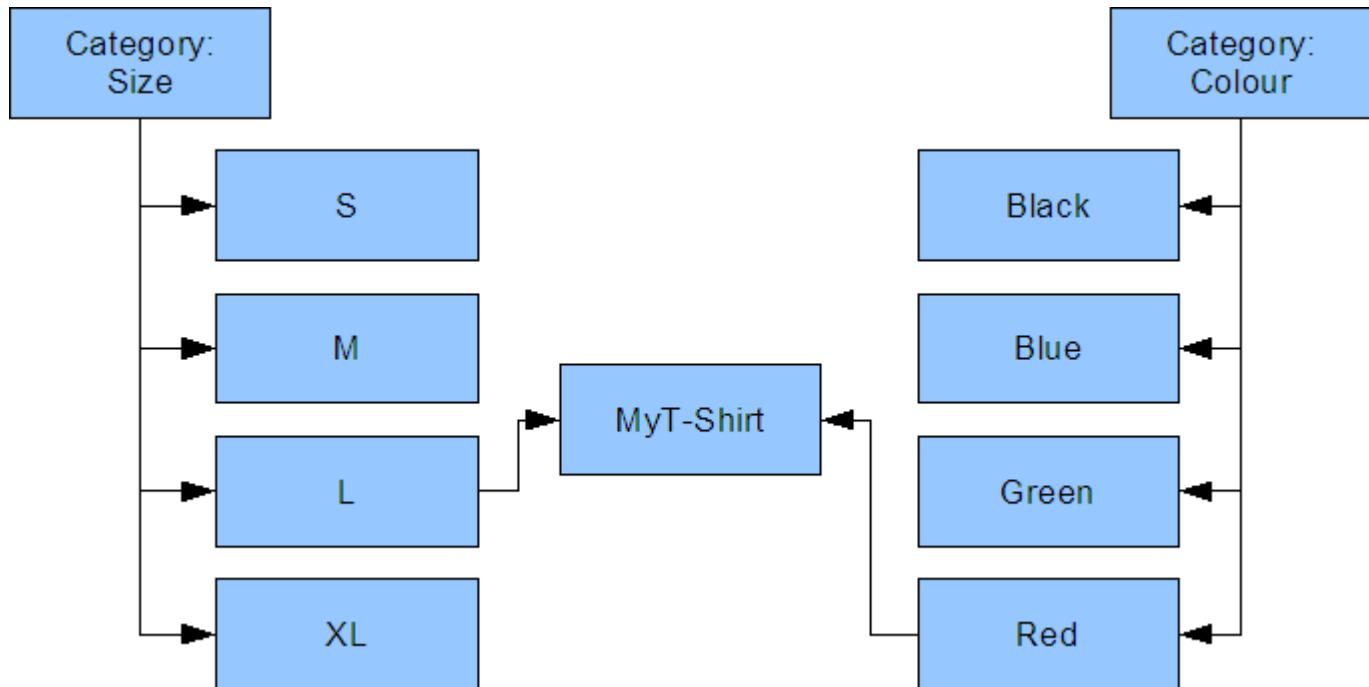
The disadvantage is that you will not have direct getter and setter methods available for the classification-based attributes. This is, however, not a issue specific to this approach but applies to everything that has to do with classification.

i Note

You can use classification-based variants and type system-based variants in any mix ratio.

Building Variants through Categorization

A further approach to building variants is by using SAP Commerce's categorization functionality. Here, the VariantProduct is added to categories for its variations, as in the diagram:



In the diagram, there are the categories Size S, Size M, Size L, and Size XL on the one hand, and the categories Color black, Color blue, Color green, and Color red on the other. By putting MyT-Shirt into the categories Size L and Color red, MyT-Shirt is marked as a red t-shirt in size L.

Implementing Size Conversion Systems for SAP Commerce Catalogs

Clothes sizes are a numerical indication of the fitting size of a piece of garment for a person. Several different size systems for clothes are used today worldwide. In some regions, it is even customary to use different shoe-size systems for different types of shoes (for example: men's, women's, children's, sport or safety shoes). If you are working on a project for a clothing company, you may face such requirements for your system.

Requirements

The requirements for our project were:

- Each product is available in various sizes
- Each size can be different for each sizing system
- Each catalog (which is bound to a country) has one sizing system

- Possibility to change a catalog's sizing system and it should affect all displayed size values "on-the-run"

Example data provided:

Sizes Data (txt file)

1.	2.	3.	4.	5.	
DE	010167	590	6-	4028460585710	1 - Sizing system
DE	010167	600	7	4028460585802	2 - Product code (articleID)
ES	010167	590	6-	4028460585710	3 - Technical size
ES	010167	600	7	4028460585802	4 - SAP size
FR	010167	590	40	4028460585710	5 - Size code (ean)
FR	010167	600	4023	4028460585802	
GB	010167	590	6-	4028460585710	
GB	010167	600	7	4028460585802	
IT	010167	590	6-	4028460585710	
IT	010167	600	7	4028460585802	
US	010167	590	7	4028460585710	
US	010167	600	7-	4028460585802	

Sizing System Data (xls file)

- headers

	A	B	C	D	E	F	G	H	I	J	K	L
1	German sizes		Spanish sizes		French sizes		UK sizes		Italian sizes		US sizes	
2	DE	DE	ES	ES	FR	FR	GB	GB	IT	IT	US	US
3	SAP size	Display size	SAP size	Display size	SAP size	Display size	SAP size	Display size	SAP size	Display size	SAP size	Display size
4	0 0		0 0		0 0		0 0		0 0		0 0	
5	1 1		1 1		1 1		1 1		1 1		1 1	
6	2 2		2 2		2 2		2 2		2 2		2 2	
7	3 3		3 3		3 3		3 3		3 3		3 3	
8	4 4		4 4		4 4		4 4		4 4		4 4	
9	5 5		5 5		5 5		5 5		5 5		5 5	
10	6 6		6 6		6 6		6 6		6 6		6 6	
11	7 7		7 7		7 7		7 7		7 7		7 7	
12	8 8		8 8		8 8		8 8		8 8		8 8	
13	9 9		9 9		9 9		9 9		9 9		9 9	
14	10 10		10 10		10 10		10 10		10 10		10 10	

- more data

166	7-	7.5	50L	50L	38L	38L	3XT	3XT	56L	56L	L/XL	L/XL
167	70A	70A	50S	50S	38S	38S	4-	4.5	56S	56S	LT	LT
168	70B	70B	52L	52L	3XL	3XL	40"	40"	58L	58L	M	M
169	70C	70C	5XL	5XL	3XLL	3XL/L	40"L	40"L	5XL	5XL	M/A	M/A
170	75A	75A	5XLL	5XL/L	3XLT	3XLT	40"S	40"S	5XLL	5XL/L	M/L	M/L
171	75B	75B	5XLT	5XLT	3XT	3XT	42"	42"	5XLT	5XLT	MT	MT
172	75C	75C	5XT	5XT	4-	4.5	44"	44"	5XT	5XT	NS	1 Size
173	8-	8.5	6-	6.5	40L	40L	46"	46"	6-	6.5	OG	1 Size
174	80A	80A	7-	7.5	40S	40S	48"	48"	7-	7.5	OSFA	1 Size
175	80B	80B	8-	8.5	42L	42L	4K	4 K	8-	8.5	OSFB	Babies
176	80C	80C	85A	85A	42S	42S	4S	4S	9-	9.5	OSFC	Kids
177	85A	85A	85B	85B	44L	44L	4XL	4XL	L	L	OSFM	Men
178	85B	85B	85C	85C	44S	44S	4XLL	4XL/L	L/50	L/50	OSFT	Toddlers
179	85C	85C	9-	9.5	46L	46L	4XLT	4XL	L/L	L/L	OSFW	Women
180	9-	9.5	90A	90A	46S	46S	4XT	4XT	L/XL	L/XL	OSFY	Youth
181	L	L	90B	90B	48L	48L	5-	5.5	LT	LT	S	S
182	L/L	L/L	90C	90C	48S	48S	50"	50"	M	M	S/A	S/A
183	L/XL	L/XL	95A	95A	4XL	4XL	5K	5 K	M/48	M/48	S/L	S/L
184	LT	LT	95B	95B	4XLL	4XL/L	5XL	5XL	M/L	M/L	S/M	S/M
185	M	M	95C	95C	4XLT	4XLT	5XLL	5XL/L	MT	MT	ST	ST
186	M/L	M/L	L	L	4XT	4XT	5XLT	5XL	NS	1 Size	XL	XL
187	MT	MT	L/L	L/L	5-	5.5	5XT	5XT	OG	1 Size	XL/A	XL/A
188	NS	1 Size	L/XL	L/XL	50L	50L	6-	6.5	OSFA	1 Size	XL/L	XL/L
189	OG	1 Size	LT	LT	50S	50S	6K	6 K	OSFB	Babies	XLT	XLT
190	OSFA	1 Size	M	M	52L	52L	6K8K	6 K-8.5 K	OSFC	Kids	XS	XS
191	OSFB	Babies	M/L	M/L	54L	54L	6L	6L	OSFM	Men	XS/A	XS/A
192	OSFC	Kids	MT	MT	5XL	5XL	7-	7.5	OSFT	Toddlers	XS/L	XS/L
193	OSFM	Men	NS	1 Size	5XLL	5XL/L	7K	7 K	OSFW	Women	XS/S	XS/S
194	OSFT	Toddlers	OG	1 Size	5XLT	5XLT	8-	8.5	OSFY	Youth		
195	OSFW	Women	OSFA	1 Size	5XT	5XT	8K	8 K	S	S		
196	OSFY	Youth	OSFB	Babies	6-	6.5	8L	8L	S/L	S/L		

Analysis

The most important thing to be aware of is that one size (identified by ean) can have a different SAP size for each sizing system. In the example shown above, the ean 4028460585710 has size 6 for the Spanish sizing system and 7 for the American. This is the SAP size that we have to translate using a proper size conversion system. The requirement is that a catalog must have one, changeable sizing system that will be used during size translation for products. Therefore, create an itemtype called SizingSystem for size-conversion systems. Each instance of the itemtype will be identified by a code attribute (which will be the country's code) and will contain a map of SAP and display values. Next, we have to extend the Catalog itemtype by adding the attribute sizeSystem of SizingSystem type. This will provide a fully configurable mechanism to manage the content of size-conversion systems and connections between themselves and Catalogs. The internal relation between CatalogVersion and products makes it possible to get the proper size-conversion system for each product's size.

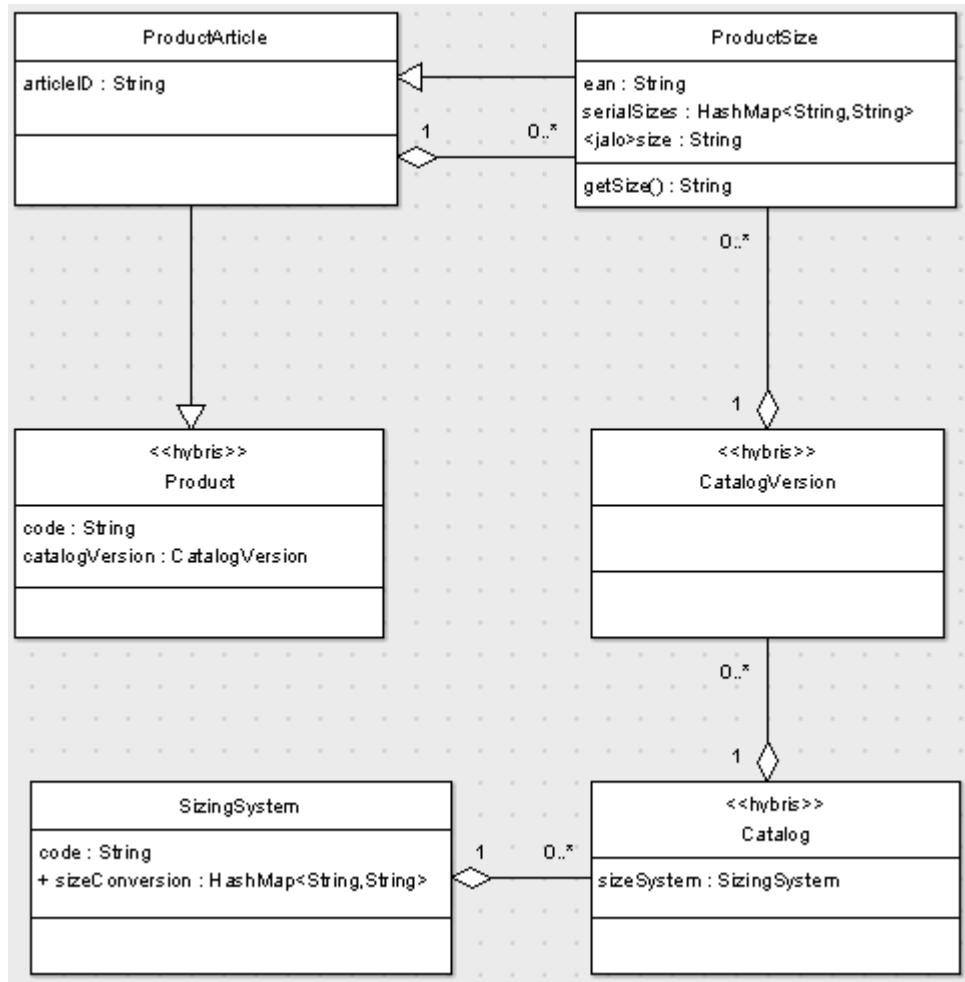
A customer is buying the products size variants but not base products (product is produced in size variants). We decided to create a special itemtype for sizes. The ProductSize type has three basic attributes:

- serialSizes: Contains a map of all of the SAP sizes for each sizing system (identified by country code). Pairs will be like this - FR->7;EN->7;US->7..etc
- size: jalo attribute that returns the translated size value
- ean: Unique identifier for product's size

What we also have to do, is to implement the mechanism that will load all of the data from the size conversion .xls file and will create all of the sizeSystem objects during system setup (how to load data from an .xls file is described in [Importing Data from Multiple Sources](#)). We also have to extend the import process, so that it will load the size's data file and create all of the ProductSize objects.

Class diagram

Illustrative class diagram: ProductSize type extends ProductArticle and as instance of platform based Product type is versionized. This, and the connection between Catalog and SizingSystem gives us the required information which size conversion system we must use when translating the real size. The size attribute is a jalo one so we will implement business logic in its getter. The setter method is not needed.



Implementation

Here is our `getSize(ctx)` method that returns display size value. As we can see the size conversion system is gotten from the product's catalog, so changing the size conversion system for this catalog will automatically affect all of the **ProductSize** objects from its catalogversion. This is the only business logic that we have to implement.

```

public String getSize(SessionContext ctx) throws JaloInvalidParameterException, JaloSecurityException {
    String size = "";
    // get serial size Map from this object
    Map serialSize = (Map)getAttribute(AdidasSizeVariant.SERIALSIZE);
    // if there is no serial size for this object
    if (serialSize == null) {
        String errMsg = "No serial size set for this item - " + this.toString();
        if (isApproved((EnumerationValue)getAttribute(CatalogConstants.TC.ARTICLEAPPROVALSTATI
            throw new JaloSecurityException(errMsg);
        } else {
            log.warn(errMsg);
            return null;
        }
    }
}
  
```

```

        }

        // get sizing system that is connected to active catalog
        SizingSystem sizingSystem = BackendManager.getInstance().getSizeSystem(
            ((CatalogVersion)getAttribute("catalogVersion")).getCatalog());

        // if no sizing system set throw exception
        if (sizingSystem == null) {
            throw new JaloSecurityException("There is no sizing system set");
        }

        // get Code of this sizing system
        String sizeSystemCode = sizingSystem.getCode(ctx);

        // get SAP size form serial size map and then get the display size value from size conversion
        size = (String)sizingSystem.getAllSizeConversion().get( (String)serialSize.get(sizeSystemCode));
    }

    return size;
}

```

Extensibility of Catalog Framework

A catalog system is a group of SAP Commerce catalogs. It represents a framework for managing catalogs enabling you to maintain the content of several product groups.

In addition, each catalog can have versions. This means you can have several versions of the same catalog, where one can be published and shown to the customers, while you can edit and update the other version in order to prepare it for future publications. This way, you can always be online with your product catalog, while updating another version of your content.

Catalog and Catalog Version example:

- Clothing Catalog
 - Summer Season 2011
 - Summer Season 2010
 - Winter Season 2011
 - Winter Season 2010
- Hardware Catalog
 - Summer Season 2011
 - Summer Season 2010
 - Winter Season 2011
 - Winter Season 2010

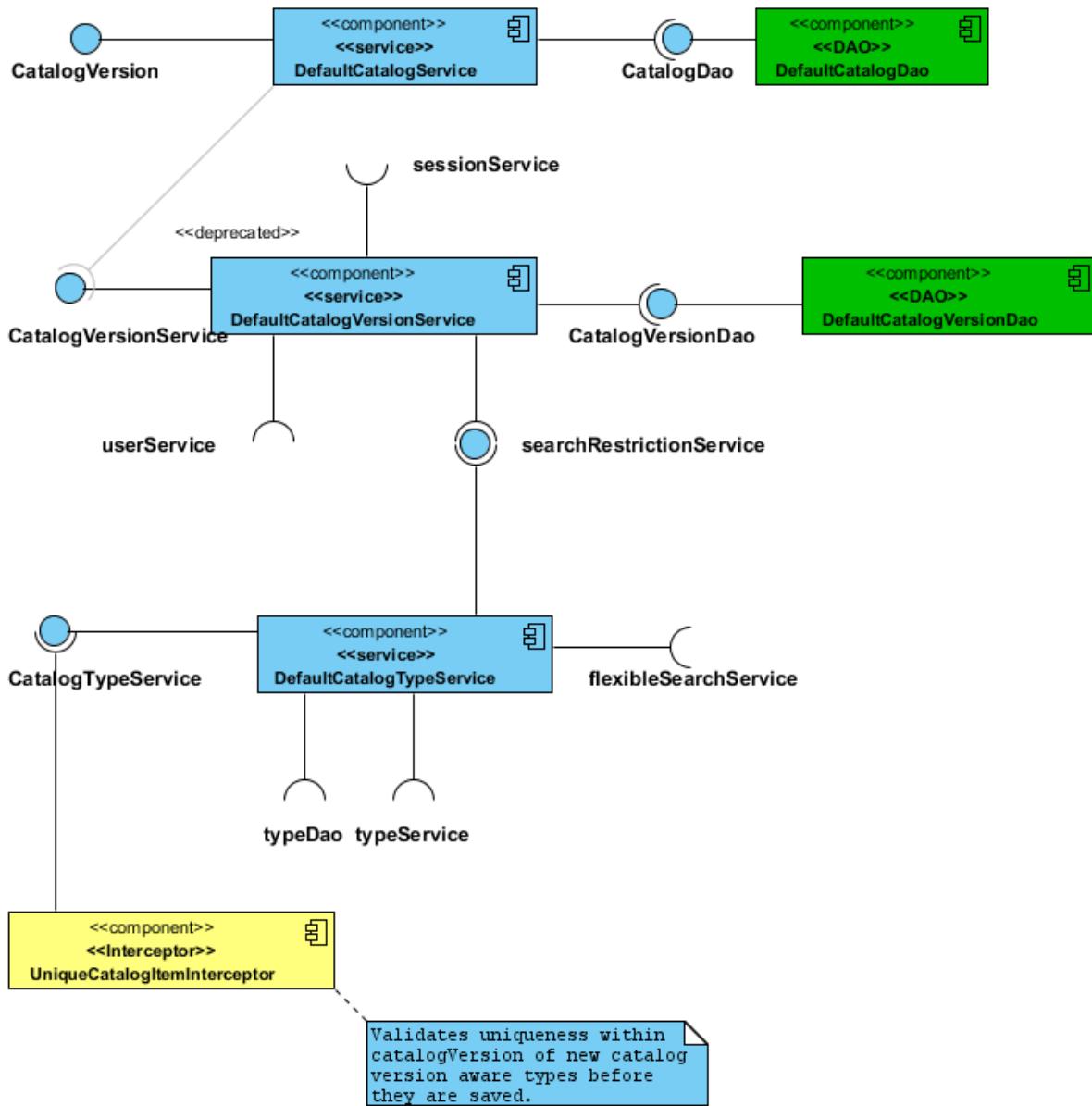
The Catalog framework is designed to manage **catalog** and **catalogVersions**. It also provides functionality for the Catalog aware types. When modeling data in the SAP Commerce you can decide that some types will be **Catalog Version** dependent. That means that the instances of such type are strictly identified with a Catalog version. Moreover, the instances are unique within the Catalog version and usually have corresponding instances in another Catalog version.

The most obvious Catalog version aware types are **product** and **category**. They define the product content of your shop in a particular Catalog version, but also web content oriented types like **CMSpage**, that define look of the on-line shop in a particular Catalog version.

The Catalog framework is organized into the functional components to separate the main topics. The following chapters describe the role of each of them and describe the process of customizing particular parts.

Main Services

The following diagram presents the main components in the Catalog framework.



CatalogService

The purpose of this component is to find:

- Particular catalog by unique code.
- Default catalog in the system.
- All catalogs.

By default it uses dedicated Data Access Object (DAO) object, which can be replaced by customized implementation if needed.

The catalog service is configured with the default catalog DAO:

```

alias alias="catalogService" name="defaultCatalogService"/>
<bean id="defaultCatalogService"
      class="de.hybris.platform.catalog.impl.DefaultCatalogService"
      parent="abstractBusinessService">
    <property name="catalogDao" ref="catalogDao"/>
    <property name="catalogVersionService" ref="catalogVersionService"/>
</bean>
  
```

By implementing the Catalog DAO contract (de.hybris.platform.catalog.daos.CatalogDao) and configuring it with an alias in your spring configuration, you can influence the logic of how the Catalog models are fetched from the data base records. Use parent attribute if you need to extend the default implementation rather than write one from scratch.

```
<alias alias="catalogDao" name="myCatalogDao"/>
<bean id="myCatalogDao" class="de.hybris.platform.catalog.daos.impl.MyCatalogDao"
      parent="defaultCatalogDao" />
```

Catalog Version Service

The focus of the Catalog Version Service are Catalog Versions. Using the methods, you can either fetch Catalog Version Data from the database or manage Catalog Versions' session. For the purpose of the former, it has a DAO object injection, similar to the catalog service. Managing catalog versions' session is achieved using the session service, which is available for the service through its inheritance from the abstractBusinessService. The service has a set of methods that can resolve if the user has the permissions to read from or write to the catalog version.

The service is configured to use the catalogVersionDao, but also to support the userService and searchRestrictionService services. These services are used to resolve principals read/write access rights to the catalog versions.

```
<alias alias="catalogVersionService" name="defaultCatalogVersionService"/>
<bean id="defaultCatalogVersionService" class="de.hybris.platform.catalog.impl.DefaultCatalogVersionService"
      parent="abstractBusinessService" >
    <property name="catalogVersionDao" ref="catalogVersionDao"/>
    <property name="userService" ref="userService"/>
    <property name="searchRestrictionService" ref="searchRestrictionService"/>
</bean>
```

If you need to change the default behavior of the service, you can implement the main interface, de.hybris.platform.catalog.CatalogVersionService, and replace the Spring bean.

```
<alias alias="catalogVersionService" name="myCatalogVersionService"/>
<bean id="myCatalogVersionService" class="de.hybris.platform.catalog.impl.MyCatalogVersionService"
      parent="defaultCatalogVersionService" >
    <!-- use whatever injection that you find useful -->
</bean>
```

In the following example, the customized service extends the default implementation, so the class signature could look like this:

```
public class MyCatalogVersionService extends DefaultCatalogVersionService implements CatalogVersionService {
    //overwrite the default methods
}
```

If you only need to replace one of the default service's injections, then follow the pattern described in the [CatalogService](#) topic.

Catalog Type Service

The catalog type service is used to handle catalog version aware types. It does the following:

- Informs you if a ComposedType is catalog version aware by a model instance, composed type instance, or composed typeCode.
- Identifies unique key attributes of a catalog version-aware type by verifying the unique attribute set, such as UID, code, and so on.
- Identifies the catalog version container attribute, to verify the type's attribute that refers to the corresponding catalog version.

- Returns the model instance of the catalog version aware type identified by a unique set of attributes and the catalog version attribute; for example: to determine the product instance with code "product123" and catalogVersion "Clothes:Spring".
- Returns all types that are catalog version aware.

The service is configured in Spring by dependent injections:

```
<alias alias="catalogTypeService" name="defaultCatalogTypeService"/>
<bean id="defaultCatalogTypeService"
      class="de.hybris.platform.catalog.impl.DefaultCatalogTypeService"
      parent="abstractBusinessService" >
    <property name="typeDao" ref="typeDao"/>
    <property name="typeService" ref="typeService"/>
    <property name="flexibleSearchService" ref="flexibleSearchService"/>
    <property name="searchRestrictionService" ref="searchRestrictionService"/>
</bean>
```

To change the default behavior, you can extend it and override the required methods.

```
<alias alias="catalogTypeService" name="myCatalogTypeService"/>
<bean id="myCatalogTypeService"
      class="de.hybris.platform.catalog.impl.MyCatalogTypeService"
      parent="defaultCatalogTypeService" >
    <!-- use whatever injections you find useful in your implementation -->
</bean>
```

The class signature would be as shown in the following example:

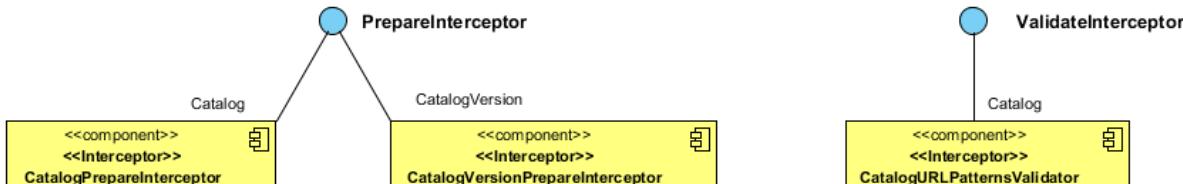
```
public class MyCatalogTypeService extends DefaultCatalogTypeService implements CatalogTypeService
{
    //overwrite the default methods
}
```

Service Layer Framework Integration - Catalog Oriented Interceptors

Similar to other types, catalog and catalog version types can have interceptors that hook in the various moments of the model life cycle. If you decide to implement your own interceptor for the catalog framework, see the [Interceptors](#) document for more information on how to create and register custom interceptors.

Validators and Preparers

When a catalog or catalog version are created, specific interceptors come into the table to validate or populate initial data into the models before they are saved in the database. The following diagram shows the ValidateInterceptor and PrepareInterceptor that are configured by default for catalog and catalog version models.



Catalog Prepare Interceptor

The catalog prepare interceptor sets an ACTIVE flag on the catalog versions when the ACTIVECATALOGVERSION is modified in the catalog model. Additionally, whenever the DEFAULT flag is set on the catalog model, the catalog prepare interceptor resets the flag on the previous default catalog.

To change the behavior, replace the interceptor bean in the following interceptor mapping. To disable the feature, comment out the interceptor mapping.

```
<bean id="catalogPrepareInterceptor" class="de.hybris.platform.catalog.interceptors.CatalogPrepare]>
    <property name="catalogService" ref="catalogService"/>
</bean>

<bean id="prepareCatalogMapping" class="de.hybris.platform.servicelayer.interceptor.impl.Intercepto]>
    <property name="interceptor" ref="catalogPrepareInterceptor"/>
    <property name="typeCode" value="Catalog"/>
</bean>
```

Catalog Version Prepare Interceptor

The catalog prepare interceptor sets the catalog version as the catalog's ACTIVECATALOGVERSION whenever the ACTIVE flag is set to TRUE on the catalog version.

```
<bean id="catalogVersionPrepareInterceptor"
      class="de.hybris.platform.catalog.interceptors.CatalogVersionPrepareIntercepto]>
    <property name="catalogService" ref="catalogService"/>
</bean>

<bean id="prepareCatalogVersionsMapping" class="de.hybris.platform.servicelayer.interceptor.impl.Ir]>
    <property name="interceptor" ref="catalogVersionPrepareInterceptor"/>
    <property name="typeCode" value="CatalogVersion"/>
</bean>
```

If you need to customize this feature, you can either remove the interceptor mapping or replace the interceptor bean with your own bean definition.

Catalog URL Patterns Validator

The catalog URL patterns validator (catalogURLPatternsValidator) verifies if the catalog's URLPATTERNS are compilable with the regexp patterns.

```
<bean id="catalogURLPatternsValidator"
      class="de.hybris.platform.catalog.interceptors.CatalogURLPatternsValidator">
    <property name="catalogService" ref="catalogService"/>
</bean>

<bean id="catalogURLPatternsValidatorMapping" class="de.hybris.platform.servicelayer.interceptor.impl.I]>
    <property name="interceptor" ref="catalogURLPatternsValidator"/>
    <property name="typeCode" value="Catalog"/>
</bean>
```

To customize this feature, you can remove the interceptor mapping or replace the interceptor bean with your own bean definition.

i Note

You can register your own interceptors for catalogs and catalog versions. If you need to assure proper intercepting order, then use the `order` property in the mapping definition.

```

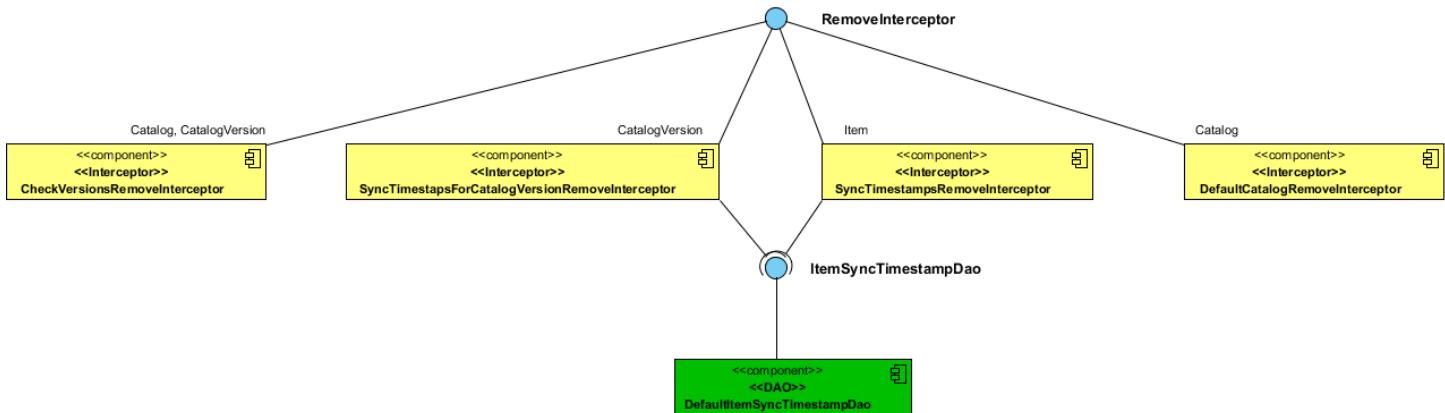
<bean id="exemplaryMappingForCatalog1"
      class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMappi
      >
    <property name="interceptor" ref="interceptor1"/>
    <property name="typeCode" value="Catalog"/>
    <property name="order" value="1"/>
</bean>

<bean id="exemplaryMappingForCatalog2"
      class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMappi
      >
    <property name="interceptor" ref="interceptor2"/>
    <property name="typeCode" value="Catalog"/>
    <property name="order" value="2"/>
</bean>

```

Remove Interceptor

The RemoveInterceptor can be used to prevent a given model from removing, for example, because of some important existing data dependencies. Another reason can be to trigger a cascade deletion of dependent data.



Check Versions Remove Interceptor

The Check Versions Remove Interceptor (checkVersionsRemoveInterceptor) verifies if a catalog version is removable. To determine this, the interceptor uses the DAO object injection to verify if the catalog still contains products, categories, keyword, or medias. To use this approach, users must first delete catalog version content, before the actual catalog version is removed. The interceptor is mapped for catalog and the catalog Version. In case of the catalog, the interceptor verifies the condition of each of its catalog versions.

```

<bean id="checkVersionsRemoveInterceptor"
      class="de.hybris.platform.catalog.interceptors.CheckVersionsRemoveIntercep
      >
    <property name="catalogVersionDao" ref="catalogVersionDao"/>
    <property name="l10nService" ref="l10nService"/>
</bean>

<bean id="checkVersionsRemoveInterceptorMapping"
      class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping'
      >
    <property name="interceptor" ref="checkVersionsRemoveInterceptor"/>
    <property name="typeCode" value="Catalog"/>
    <property name="order" value="1"/>
</bean>

<bean id="checkVersionsRemoveInterceptorCatalogVersionMapping"
      class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping'
      >
    <property name="interceptor" ref="checkVersionsRemoveInterceptor"/>

```

```
<property name="typeCode" value="CatalogVersion"/>
</bean>
```

You can modify the mapping definition in order to customize the feature behavior for the catalog and/or the catalog version types. You can also remove the mapping and create a custom mapping that better suits your business model.

Default Catalog Remove Interceptor

The default catalog remove interceptor prevents the removal of a catalog that is marked with the DEFAULT flag. If you do not require this or if you want to define different removal constraints, you can either remove the corresponding mapping or change the interceptor bean definition in it.

```
<bean id="defaultCatalogRemoveInterceptor"
      class="de.hybris.platform.catalog.interceptors.DefaultCatalogRemoveInterceptor"
      >
    <property name="l10nService" ref="l10nService"/>
</bean>

<bean id="defaultCatalogRemoveInterceptorMapping"
      class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping"
      >
    <property name="interceptor" ref="defaultCatalogRemoveInterceptor"/>
    <property name="typeCode" value="Catalog"/>
    <property name="order" value="0"/>
</bean>
```

SyncTimestampsRemoveInterceptor and SyncTimestampsForCatalogVersionRemoveInterceptor

Each time a synchronization of an item is performed between two catalog versions (each time the product is synchronized from the staged version to the online version), a historical record is created for the operation. The record is called the syncTimestamp.

The following interceptors perform cascade removal of all synchronization timestamps:

- syncTimestampsRemoveInterceptor: Removes all synchronization timestamps connected with an item (catalog version aware) that is being removed
- syncTimestampsForCatalogRemoveInterceptor: Removes the timestamps related with the catalog version that is being removed

Both interceptors use the same DAO object, that fetches synchronization timestamps, the itemSyncTimestampDao.

```
<bean id="syncTimestampsRemoveInterceptor"
      class="de.hybris.platform.catalog.interceptors.SyncTimestampsRemoveInterceptor"
      >
    <property name="itemSyncTimestampDao" ref="itemSyncTimestampDao"/>
</bean>

<bean id="syncTimestampsRemoveInterceptorMapping"
      class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping"
      >
    <property name="interceptor" ref="syncTimestampsRemoveInterceptor"/>
    <property name="typeCode" value="Item"/>
</bean>

<bean id="syncTimestampsForCatalogVersionRemoveInterceptor"
      class="de.hybris.platform.catalog.interceptors.SyncTimestampsForCatalogVersionRemoveInterceptor"
      >
    <property name="itemSyncTimestampDao" ref="itemSyncTimestampDao"/>
</bean>
```

```
<bean id="syncTimestampsForCatalogRemoveInterceptorMapping"
      class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping"
      >
    <property name="interceptor" ref="syncTimestampsForCatalogRemoveInterceptor"/>
    <property name="typeCode" value="CatalogVersion"/>
</bean>
```

In this case you can:

- Remove the mapping.
- Replace the interceptor bean definition.
- Use different dedicated DAO object bean.

i Note

You can introduce your own RemoveInterceptor related to the catalog framework. To do this, implement the `de.hybris.platform.servicelayer.interceptor.RemoveInterceptor` interface and register the bean in a valid type mapping in your extension spring configuration.

Synchronizing Catalogs

Catalog versions can be synchronized in a one-directional way for updating a catalog version according to another. This way you can transfer catalog content between the catalog versions which represent steps in your catalog maintenance process. You can synchronize catalog versions, even if they are of the same or different catalogs within SAP Commerce.

The catalog that is online, and therefore visible to customers, is often not the only catalog that your platform contains; usually, you have several non-visible catalogs as well that you can work with. This has the advantage in that you can assemble several catalogs in advance so you can use them when needed, for example, when seasons change and your product portfolio needs to match that.

For transferring the differing content between two catalogs, you use synchronizing operations among subordinate catalogs versions.

i Note

From a technical perspective, there is no difference between catalogs or catalog versions. For example, the online catalog is different from the catalog that is hidden only because of access rights.

How Does Synchronizing Work?

The single catalog that is visible to your customers is called **online**, and the catalogs that are not visible to customers are **staged**. A standard synchronizing operation is the act of updating the online catalog with at least one of the staged catalogs. If you synchronize a staged catalog, the online catalog will then contain all values from that staged catalog.

A synchronizing operation can be applied to an entire catalog version, but it can also be applied to selected categories or products. Synchronization works in a unidirectional manner only, from staged to online. However, you can apply two synchronizing operations to establish bi-directional update procedures.

i Note

You can also change access rights of the staged catalog to be the online catalog. However, SAP Commerce does not recommend this solution because you will not have a backup of your online catalog. Synchronizing catalogs is easy and can be fully automated.

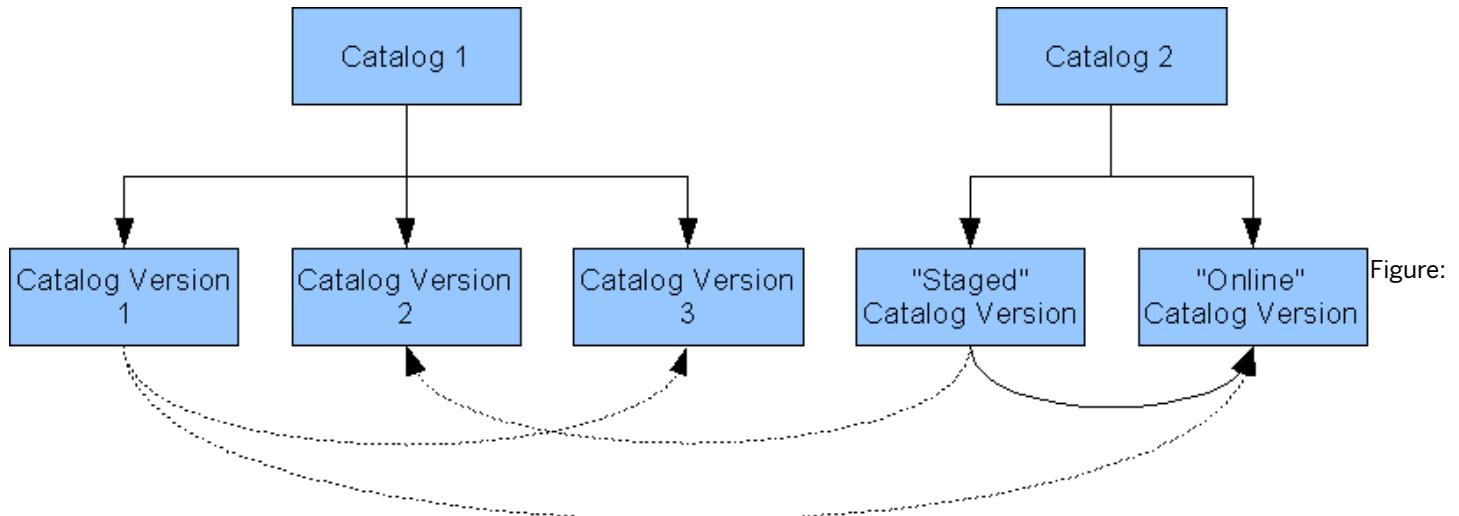


Figure:

This illustration shows a sample catalog system consisting of **Catalog 1** with three versions, and **Catalog 2** with two versions. You may want such a setup if, for example, you have several suppliers who have product catalogs of their own. The dotted lines represent customized synchronizations, and the solid line represents the default synchronization: **Default:Staged** ➔ **Default:Online** ➔.

Synchronization Statuses

When you search for items in Backoffice, for example products, they display with a graphic symbol next to each product name. Each such symbol indicates a current product synchronization status.

- : Indicates that a source item is synchronized with the target item.
- : Indicates that a source item has been changed and needs synchronization with the target item.
- : Indicates that an initial synchronization is needed. The source catalog version has never been synchronized with the target catalog version. Since synchronizing single items is disabled, you need to synchronize an entire catalog version.

Each synchronization execution produces synchronization timestamps. The logic evaluating the **Initial synchronization needed** status checks to see if any timestamps exist for the source catalog version. In other words, it checks if the catalog has ever been synchronized.

It is possible to change the logic by introducing the following configuration entry:

```
catalog.synchronization.initialinit.check.timestamps=false
```

In this case, the status will be evaluated based on the executions of synchronizations as Cronjobs.

i Note

If you set the `catalog.synchronization.initialinit.check.timestamps=false` and clear the historical synchronization Cronjob execution items from your system, you will see the status for your catalog version.

See the following topic for more information: *No Changes in a Synchronized Catalog Version After the Synchronization Was Modified?*

About Synchronizations

A synchronizing operation is held in a **synchronization** object referencing the source and the target object. Thus, synchronizations are definitions of how products and their data such as names, descriptions, and prices will be copied into the target catalog.

You can create, edit, and delete a synchronization using Backoffice. Each synchronization is bound to the catalog version that is used as the synchronization source. For more information, see [Editing Synchronization Rules](#).

i Note

To support systems with a large number of products, SAP Commerce supports multithreaded catalog synchronization operations. For details, see [Performing Multithreaded Catalog Synchronization in Backoffice](#).

i Note

The synchronization logic is designed to manage deadlocks resulting from synchronization being a highly-concurrent process.

Performing Synchronization

Backoffice enables you to:

- edit synchronization rules
- synchronize complete catalog versions or chosen items
- check the impact of synchronization

You can find more explanation in separate synchronization topics.

No Changes in a Synchronized Catalog Version After the Synchronization Was Modified?

After you have become familiar with how synchronization works, you will probably want to experiment with synchronizations. If so, you may run into the following situation: You have synchronized a catalog version, modified the synchronizations, and after launching the synchronizations again, the target catalog version is the same as before running the edited synchronizations. For example, the original synchronizations updated the product codes, whereas the new synchronizations updated their prices. However, the target catalog version still only contains the codes. This seems to be a bug, but, in fact, it is not. It has to do with the way the catalog selects the products to synchronize. Whether a product is considered to be new and therefore, synchronized or not, depends on the last point of time when the product was changed. The platform itself calls this the modification timestamp.

Once a product is synchronized with a certain synchronization , that timestamp is stored with that rule. Consequently, the product is considered up-to-date in the target catalog. Unless the product's last change date changes, that product will not be synchronized any more with that rule, even if the rule was edited. The idea behind this is that the platform can avoid synchronizing up-to-date products, which speeds up the synchronizing operation tremendously. In the case that the synchronization has changed, however, this commonly useful mechanism fails. It is technically possible to find out if a synchronization has changed since its last run, but due to all of the checks and database accesses that are necessary, performance would be greatly impacted.

If you want to try an edited synchronization effect with a previously synchronized product, you need to manually edit that product. For example, change the product description, and then save the changed product. The platform will then set the modification timestamp to the current date and synchronize this product again on the synchronization's next run. After this edit and change, you can directly re-edit your product to its previous state. For example, add a letter to the description, save it and then remove that letter from the description again and save. That way, the timestamp will be updated, while your product stays the same. You might think that simply opening the product's editor and saving it will change the timestamp, but it does not because the system recognizes that the product has not changed. It does not save the product and therefore, the timestamp does not change either.

Synchronization of Translated Collection Elements

A warning message appears if an element cannot be translated, but the synchronization continues. Additionally, there is a new flag to control the default behavior `synchronization.allow.partial.collection.translation`.

This property defines what happens with the synchronization of a collection of references if the sync process cannot translate all the references in the source collection.

`synchronization.allow.partial.collection.translation=true` (default behavior) means that if the synchronization fails to translate any of the source references, it skips translation of that reference with a warning message and continues translation of the root item. The result is the target collection only contains those target references that were successfully translated.

`synchronization.allow.partial.collection.translation=false` means that if the synchronization fails to translate any of the source references, it aborts synchronization of the root item with an error message and any pending changes to the root item are rolled back.

The `partiallyTranslatable` in `SyncAttributeDescriptorConfig` allows you to set up a synchronization job in ImpEx and override the global behavior for a given synchronization job.

Configuring Synchronization as CronJobs

A CronJob is a task that runs automatically at a certain point of time for a given number of times. For example, CronJobs may create backups every day and store them on remote drives on a certain day of the week, or rebuild indexes for the search engine every other day at a certain time.

Synchronizations are CronJobs as well, so you may have the respective catalog versions being synchronized automatically at a certain time. The idea behind this is that these synchronizations may cause heavy load on your servers and you should run them at times when the system load is usually low, such as early morning or on Sundays. SAP Commerce enables running synchronization processes independently, without a related CronJob. For details on CronJobs, refer to [The Cronjob Service](#).

Configuring Synchronization for Catalog Systems

A catalog system is group of SAP Commerce catalogs connected by synchronizing operations that are realized between the contained catalog versions. You organize the involved catalogs in a way to establish a synchronization process from input to output. Thus, a catalog system represents a framework for managing multiple output catalogs enabling you to maintain the varying content of several product catalogs. In addition, a catalog system can handle multiple input catalogs. Typically, a catalog system is set up around a master catalog holding the leading versions of product data.

See also: [Managing Multiple Product Catalogs in a Catalog System](#)

Example of a catalog system:

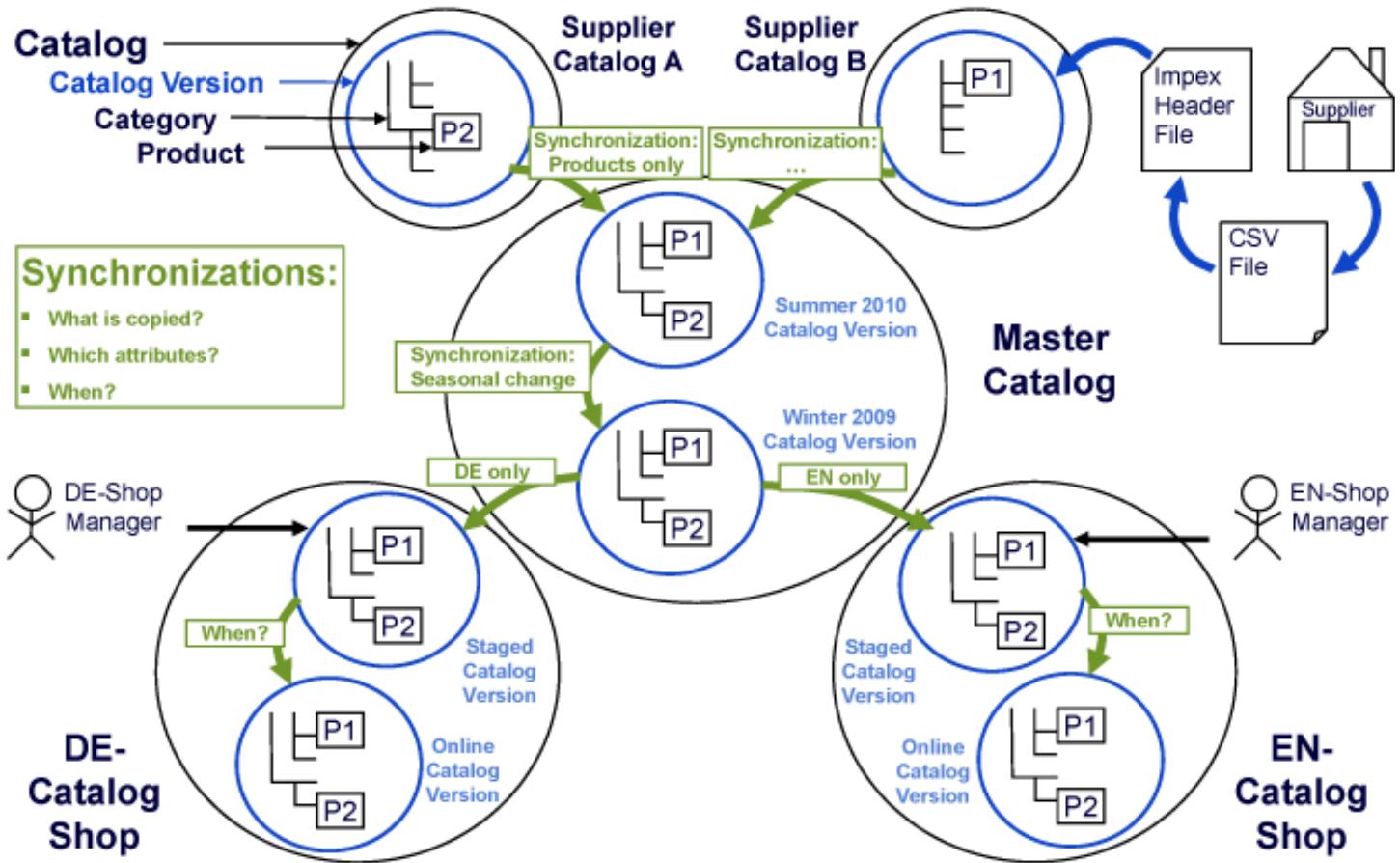


Figure: This sample catalog system consists of two supplier catalogs (A and B), a **Master Catalog**, and two language-specific output catalogs; **DE-Catalog Shop** and **EN-Catalog Shop**. The master catalog and the output catalogs each consist of two catalog versions to distinguish between editing stages and final stages. While the current catalog version is Winter 2009, the next issue, Summer 2010 is prepared in its own catalog version. In the output catalogs, the contained catalog versions are used to separate country-specific editing and approval activities from the propagated final **Online** catalog version. The catalog versions in this catalog system are connected by synchronizations, indicated by green arrows that configure the content transfer between catalog versions.

Synchronization with ServiceLayer

You can use the ServiceLayer adapter for synchronization jobs. The immediate benefit is that you can use the ServiceLayer infrastructure such as interceptors and validators. To enable the ServiceLayer mode, place the following property in your `local.properties` file:

```
synchronization.legacy.mode=false
```

You can also change this dynamically in SAP Commerce Administration Console. Go to the **Platform tab** **Configuration**.

For more information, see the following:

- [ServiceLayer](#)
- [Configuring the Behavior of SAP Commerce](#)
- [Administration Console](#)

Editing Synchronization Rules

Backoffice allows you to edit synchronization rules by using synchronization editor.

Context

To edit synchronization:

Procedure

1. Log into Backoffice and navigate to **System > Multithreaded Synchronization**.

The result view displays available synchronization jobs.

2. Choose the synchronization job you wish to modify to expand a synchronization editor.

The screenshot shows the Multithreaded Synchronization editor. On the left is a navigation sidebar with a dark theme, containing sections like System, Types, and Audit Report Config. The main area has a light theme. At the top, there's a search bar with a yellow 'SEARCH' button. Below it, a toolbar with icons for delete, add, and edit. A table header row shows columns for 'Code', 'Synchronization source', and 'Synchronization target'. A row for 'Sync Default:Staged -> Default:Online' is selected, highlighted with a yellow border. The status bar at the bottom of this row indicates 'default catalog : Staged' for source and 'default catalog : Online' for target. Below the table, a message says '1 ITEMS SELECTED'. Underneath, a specific synchronization job is detailed: 'Sync Default:Staged -> Default:Online'. It has tabs for 'CONFIGURATION', 'CRONJOBS', 'RESTRICTIONS', 'ADVANCED ATTRIBUTES', and 'ADMINISTRATION'. The 'CONFIGURATION' tab is active. In the 'ESSENTIAL' section, the 'Code' field contains 'Sync Default:Staged -> Default:Online', and the 'Synchronization source' and 'Synchronization target' fields both show 'default catalog : Staged'.

3. On the **CONFIGURATION** tab in the editor area, click **Edit Configuration** button.

The screenshot shows the configuration editor for the 'Sync Default:Staged -> Default:Online' job. The top part is identical to the previous screenshot. The 'CONFIGURATION' tab is active. The 'ESSENTIAL' section shows the 'Code' field with 'Sync Default:Staged -> Default:Online', 'Synchronization source' with 'default catalog : Staged', and 'Synchronization target' with 'default catalog : Online'. Below this is a section titled 'SYNCHRONIZATION ATTRIBUTES CONFIGURATION'. It includes a table for 'Synchronization attributes configuration' with two columns: 'Included Properties' (31 Types, 224 Attributes) and 'Excluded Properties' (0 Types, 0 Attributes). A yellow box highlights the 'Edit Configuration' button at the bottom left of this section.

Browse the tree to expand available properties. Any selected property is updated in the target catalog, and any unselected properties keep their values. This enables you to merge content from several catalog versions into a single one, and split them into individual catalogs again.

The screenshot shows the 'Edit Synchronization Attributes Configuration' dialog. On the left, a tree view lists various synchronization items: 'Item [Item]' (selected), 'Comments [comments]', 'Documents [allDocuments]', 'Owner [owner]', 'AbstractMedia [AbstractMedia]', 'Category [Category]', 'Category Feature [ClassAttributeAssignment]', 'Configured product [AbstractConfiguratorSetting]' (selected), '[configurationCategoryPOS]', 'Configurable category [configurationCategory]', 'Configurator type [configuratorType]', 'Identifier [id]', 'Qualifier [qualifier]', and 'Feature Descriptor [ClassificationAttribute]'. On the right, a 'Details' panel shows checkboxes for 'Synchronize', 'Copy by value', 'Untranslatable value', and 'Partially translatable'. At the bottom, there are status indicators: 'Attribute' (grey), 'Completely Included' (blue checked), 'Partially Included' (grey), and 'Not Included' (red crossed-out). There are also 'CANCEL' and 'SAVE' buttons.

4. On the **ADMINISTRATION** tab, set the synchronization rules of how product data is copied to the target catalog version:

- o **Basic Setting:** You can decide whether missing objects in the target catalog version should be created. Furthermore, you can decide that existing products in the target catalog version that do not exist in the source catalog version should be removed. Alternatively, only products that exist in both the source and the target catalog versions are synchronized.
- o **Language Settings:** Define the languages of products attributed to be synchronized.
- o **Synchronization User/Groups:** Define users and groups that are permitted to initiate such a process although they don't have such write permissions.
If the option, **Respect sync. permissions only** is checked, only the defined sync. users are allowed to synchronize. If it is not checked, then a user with granted write permissions for the target catalog version is allowed to do so, too.
- o **Type Settings:** Set the root types that are analyzed in case of complete synchronizations of catalog versions. Such synchronizing procedures searches for new, changed, updated, and removed items of the listed types.

Checking the Impact of Synchronization

Backoffice allows you to look up differences in the content of your catalog versions. The **Catalog Version Diff** action displays items that need to be synchronized. You may want to use this function to see the differences before synchronization and a synchronization result afterwards.

Context

A list of differences between two versions of a catalog is generated based on a synchronization job that binds the two versions of the catalog. The synchronization job configuration determines the items that are going to be synchronized. As a result, only the items provided in the configuration are taken into consideration when you generate the list of differences between your catalog versions.

In case given items have been already synchronized, generating a list of differences returns no result.

To better understand the context of this topic, get familiar with the synchronization example and its prerequisites described in [Synchronizing Catalog Versions in Backoffice](#).

To display a list of differences between catalog versions, log in to Backoffice and follow these steps:

Procedure

1. In the Backoffice navigation tree, click **Catalog > Catalog Versions**.

A tab with a list of available catalog versions shows up.

2. Click a catalog version.

In our case, it is the **Staged** version.

Information about the catalog version displays, with action icons available for that catalog version, including the **Catalog Version Diff** action icon.

Catalog	Catalog Version	is active catalog version
<input checked="" type="checkbox"/> default catalog	Staged	false
<input type="checkbox"/> default catalog	Online	true

0 ITEMS SELECTED

default catalog : Staged

CATALOG VERSION **CONTENT** **CATALOG VERSIONS** **BMECAT** **PERMISSIONS** **ADMINISTRATION**

ESSENTIAL

Catalog	Catalog Version
default catalog	Staged

3. Click the **Catalog Version Diff** icon.

The **Catalog Version Diff** window opens.

Notice that for the **Staged** catalog version, a synchronization job is already chosen. It is **Sync Default:Staged -> Default:Online**.

4. **Optional:** Choose a particular synchronization job.

5. Click **Generate**.

A list displays with items that haven't been synchronized.

Catalog Version Diff



Select synchronization job:

Sync Default:Staged -> Defa

GENERATE

Type	Source	Last modification of source	Last synchronization
Product	001 [product001] - default catalog : Staged	January 3, 2018 3:08:12 PM CET	
Product	002 [product002] - default catalog : Staged	January 3, 2018 3:08:36 PM CET	
Product	003 [product003] - default catalog : Staged	January 3, 2018 3:09:00 PM CET	

Results

You have displayed a list of items that require to be synchronized.

Synchronizing Catalog Versions in Backoffice

Backoffice allows you to synchronize entire catalog versions. To synchronize your catalogs, use the **Catalog synchronization** action.

Context

The example synchronization procedure we perform below is based on a few prerequisites:

- We perform a full synchronization between the **Staged** and **Online** versions of the **default** catalog. The catalog is provided with SAP Commerce by default.
- Since both versions of the catalog are empty, we created a few products (**product001**, **product002**, **product003**) that are the items that we synchronize. The products are created for the **Staged** version of the catalog.
- Our synchronization example uses the **Sync Default:Staged -> Default:Online** job that binds both versions of the catalog. The job defines the **Staged** version as the source catalog, and the **Online** version as the dependent (target) catalog. This relation determines the direction of synchronization - items are synchronized from the source to the target. The job is provided with SAP Commerce by default..

Before you start synchronization, you may want to see the difference in the content of the catalogs you want to synchronize. For more information, see [Checking the Impact of Synchronization](#).

To synchronize catalogs, log in to Backoffice and follow these steps:

Procedure

1. In the Backoffice navigation tree, click **Catalog > Catalog Versions**.

A tab with a list of available catalog versions shows up.

2. Click the source version of the catalog you want to synchronize.

In our case, it is the **Staged** version of **default catalog**.

A tab shows up with information about the catalog as well as available action icons, including the **Catalog synchronization** icon.

The screenshot shows the SAP Fiori Catalog Management interface. At the top, there is a search bar with a magnifying glass icon and a yellow 'SEARCH' button. Below the search bar are standard navigation icons: a trash can, a plus sign, a dropdown arrow, a refresh symbol, and a double arrow. To the right, there are icons for list view, edit, and grid view, followed by the text '2 items'. The main area displays a table with two rows:

Catalog	Catalog Version	is active catalog version
<input checked="" type="checkbox"/> default catalog	Staged	false
<input checked="" type="checkbox"/> default catalog	Online	true

Below the table, it says '0 ITEMS SELECTED'. Underneath the table, the text 'default catalog : Staged' is displayed. At the bottom of the screen, there is a navigation bar with tabs: CATALOG VERSION (which is highlighted in blue), CONTENT, CATALOG VERSIONS, BMECAT, PERMISSIONS, and ADMINISTRATION.

ESSENTIAL**Catalog**

default catalog

Catalog Version

Staged

CATALOG VERSION**Generation date****is active catalog version**

True



False

Languages

German [de]

English [en]

Territories

...

3. Click the **Catalog synchronization** icon.

The **Catalog synchronization** wizard opens up.

4. In the **Sync job selection** tab of the wizard, choose a synchronization job .

In our case, it is **Sync Default:Staged -> Default:Online**.



Catalog synchronization

CATALOG SYNCHRONIZATION
Sync Job selection

CATALOG SYNCHRONIZATION
Advanced configuration

CATALOG SYNCHRONIZATION
Results

Synchronization Job:

Sync Default:Staged -> Default:Online



CANCEL

NEXT

5. Click **Next** to move to the **Advanced configuration** tab and adjust your configuration if required.

We don't change anything at this step.

6. Click **Start** in the **Advanced configuration** tab to perform the synchronization.

Synchronization results display when the process has finished.

7. Click **Done** to close the wizard window.

Results

You have performed synchronization.

Now you can check the synchronization result. For more information, see [Checking the Impact of Synchronization](#). As another option, you can check the status of the items you synchronized. For more information, see the [Synchronization Statuses](#) topic in [Synchronizing Catalogs](#).

Synchronizing Items

Backoffice enables you to synchronize selected items. To synchronize your items, use **Synchronize Action** icon.

Context

The following example synchronization procedure is based on a few prerequisites:

- We perform a synchronization of items from the **Staged** to the **Online** version of the default catalog.
- We created three products (`product001`, `product002`, `product003`) and we synchronize two of them (`product001`, `product002`). The products are assigned to the **Staged** version of the catalog.

For more information about creating a product, see section [Creating a Product](#) in [Editing Tax and Discount Values of an Order in Backoffice](#).

- Our synchronization example uses the Sync Default:Staged -> Default:Online job that binds both versions of the catalog. The job defines the Staged version as the source catalog, and the Online version as the dependent (target) catalog. This relation determines the direction of synchronization - items are synchronized from the source to the target. The job is provided with SAP Commerce by default.

To synchronize items, log in to Backoffice and follow these steps:

Procedure

- In the Backoffice navigation tree, click **Catalog > Products**.

A tab with a list of available products shows up.

Article Num...	Identifier	Status	Catalog version
<input type="checkbox"/> product001	product001	● △ ○	default catalog : Staged
<input type="checkbox"/> product002	product002	● △ ○	default catalog : Staged
<input type="checkbox"/> product003	product003	● △ ○	default catalog : Staged

- Choose the products you want to synchronize.

In our case, they are **product001** and **product002**.

Article Num...	Identifier	Status	Catalog version
<input checked="" type="checkbox"/> product001	product001	● △ ○	default catalog : Staged
<input checked="" type="checkbox"/> product002	product002	● △ ○	default catalog : Staged
<input type="checkbox"/> product003	product003	● △ ○	default catalog : Staged

A tab shows up with information about the product as well as available action icons, including the **Synchronize Action** icon.

Article Num...	Identifier	Status	Catalog version
<input checked="" type="checkbox"/> product001	product001	● △ ○	default catalog : Staged
<input checked="" type="checkbox"/> product002	product002	● △ ○	default catalog : Staged
<input type="checkbox"/> product003	product003	● △ ○	default catalog : Staged

2 ITEMS SELECTED

product002 [product002] - default catalog : Staged

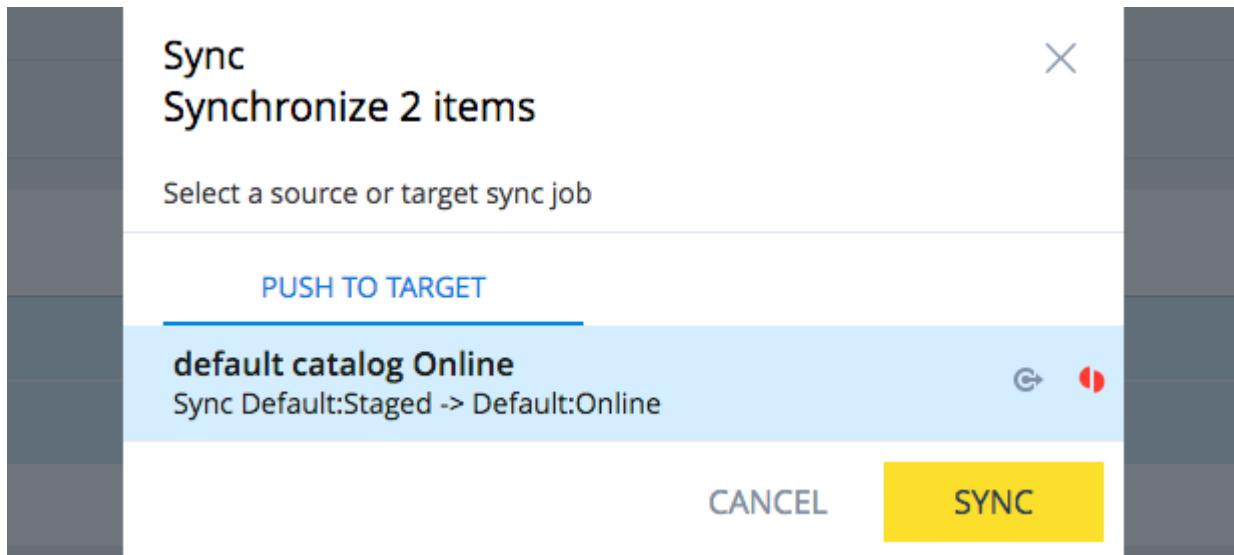
Synchronize Save

- Click the **Synchronize Action** icon.

The **Synchronize Action** wizard opens up.

- In the **Synchronize Action** wizard, choose a synchronization job and click **SYNC**.

In our case, it is **Sync Default:Staged -> Default:Online**.



The synchronization process is completed. You can look up the items in the collection browser and see their synchronization statuses.

Collection Browser			
Filter Tree entries		SEARCH	
			5 items
	Article Num... Identifier	Status	Catalog version
	product001 product001	● ▲ ○	default catalog : Staged
	product001 product001	● ▲ ○	default catalog : Online
	product002 product002	● ▲ ○	default catalog : Staged
	product002 product002	● ▲ ○	default catalog : Online
	product003 product003	● ▲ ○	default catalog : Staged

Results

You have performed the synchronization of your items.

For more information, see [Checking the Impact of Synchronization](#). As another option, you can check the status of the items you synchronized. For more information, see the Synchronization Statuses section in [Synchronizing Catalogs](#).

Dependent Synchronization

Dependent synchronization avoids creation of cross-references between items from **Staged** and **Online** catalog versions that are a part of different catalogs.

Overview of Dependent Synchronization

It is recommended to use dependent synchronization in the following situations:

- You have multiple catalogs with Staged and Online catalog versions
- These catalogs have items that refer to each other
- These catalogs have synchronization jobs to synchronize their catalog versions

The following diagram illustrates an example with two different catalogs:

- **Categories Catalog:** It contains the category structure for the Web shop. There are two catalog versions:
 - **Categories Catalog Staged Version:** Here you create the structure of catalogs. Then you decide which product is shown in which category, that is to say, you match a category to a product. In your catalogs structure products are held in the Product Catalog. These changes need to be synchronized with the **Categories Catalog Online Version** to make the changes visible in the Web shop.

- o **Categories Catalog Online Version:** After synchronization this catalog version should reflect data from the **Categories Catalog Staged Version:**

- **Products Catalog:** It is fed by the SAP system. There are two catalog versions:

- o **Products Catalog Staged Version:** It contains all available products with all updated attributes, for example price. You decide which catalog should be available online and for how long. From this point you can also refer a product to a category. All changes are synchronized with the **Products Catalog Online Version.**

- o **Products Catalog Online Version:**

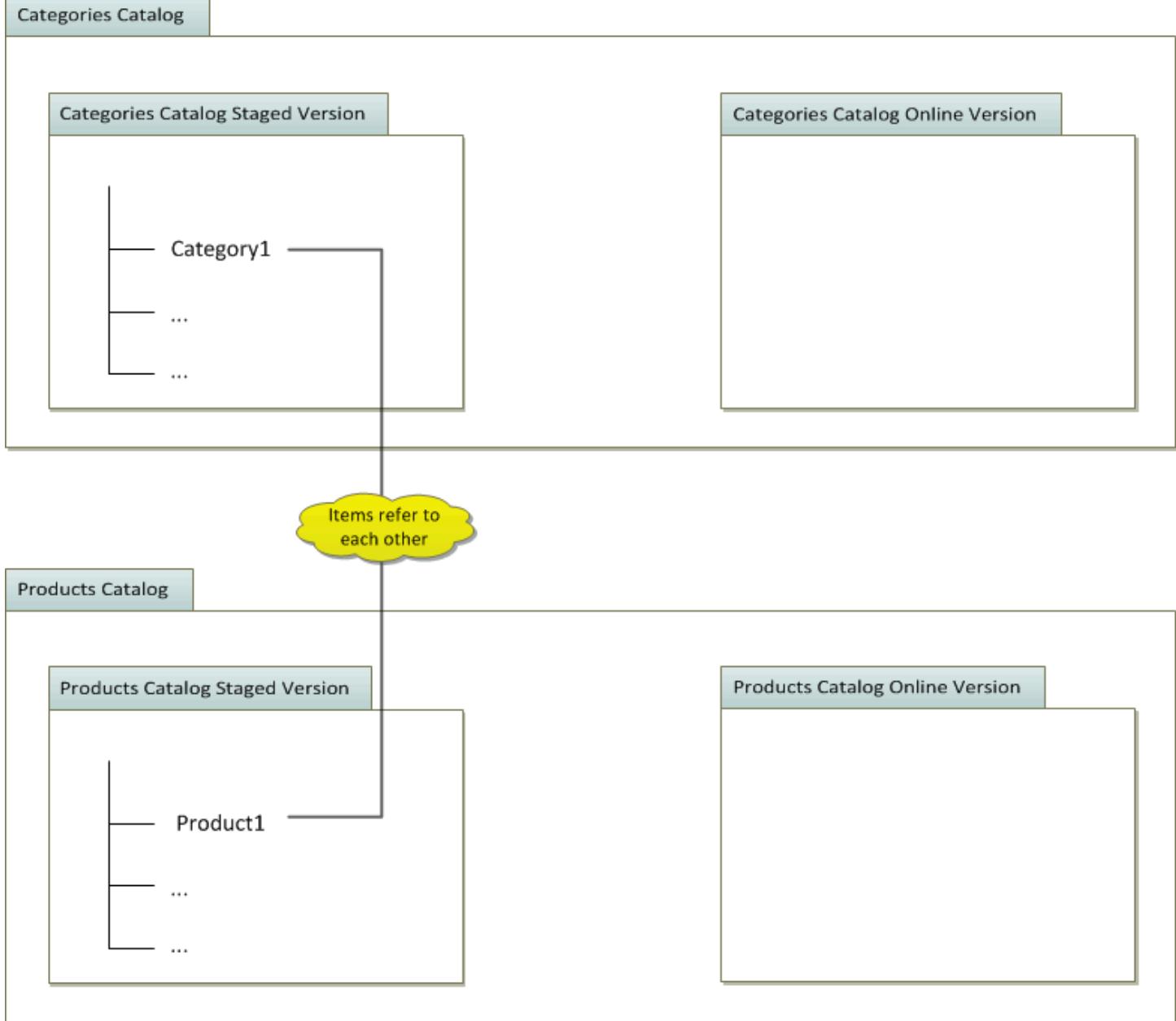


Figure: Catalogs before synchronization.

Synchronization without Dependency

If synchronization jobs are not aware of each other, then cross-references between items from different catalogs are created.

The following diagram provides an example with the same catalogs shown in *About Dependent Synchronization* section after running an independent synchronization:

1. **Categories Catalog Synchronization:** Category1, from the Categories Catalog Staged Version, is copied within the same Categories Catalog to the Categories Catalog Online Version. Category1 contains a reference to Product1 in the Products Catalog Staged Version, which is a part of the Product Catalog. Because of the relation, this reference is also copied. As a result, after the synchronization, Product1 refers to two categories: stages and online.

2. Products Catalog Synchronization: Product1 from the Products Catalog Staged Version is copied within the same Products Catalog to the Products Catalog Online Version. As Product1 already had two references to two different categories (staged and online), those references are also copied.

After both synchronization jobs are finished, items from online versions of catalogs do not only hold references to each other, but also hold cross-references to the staged catalog versions. The situation would look exactly the same if synchronization jobs ran in a different order.

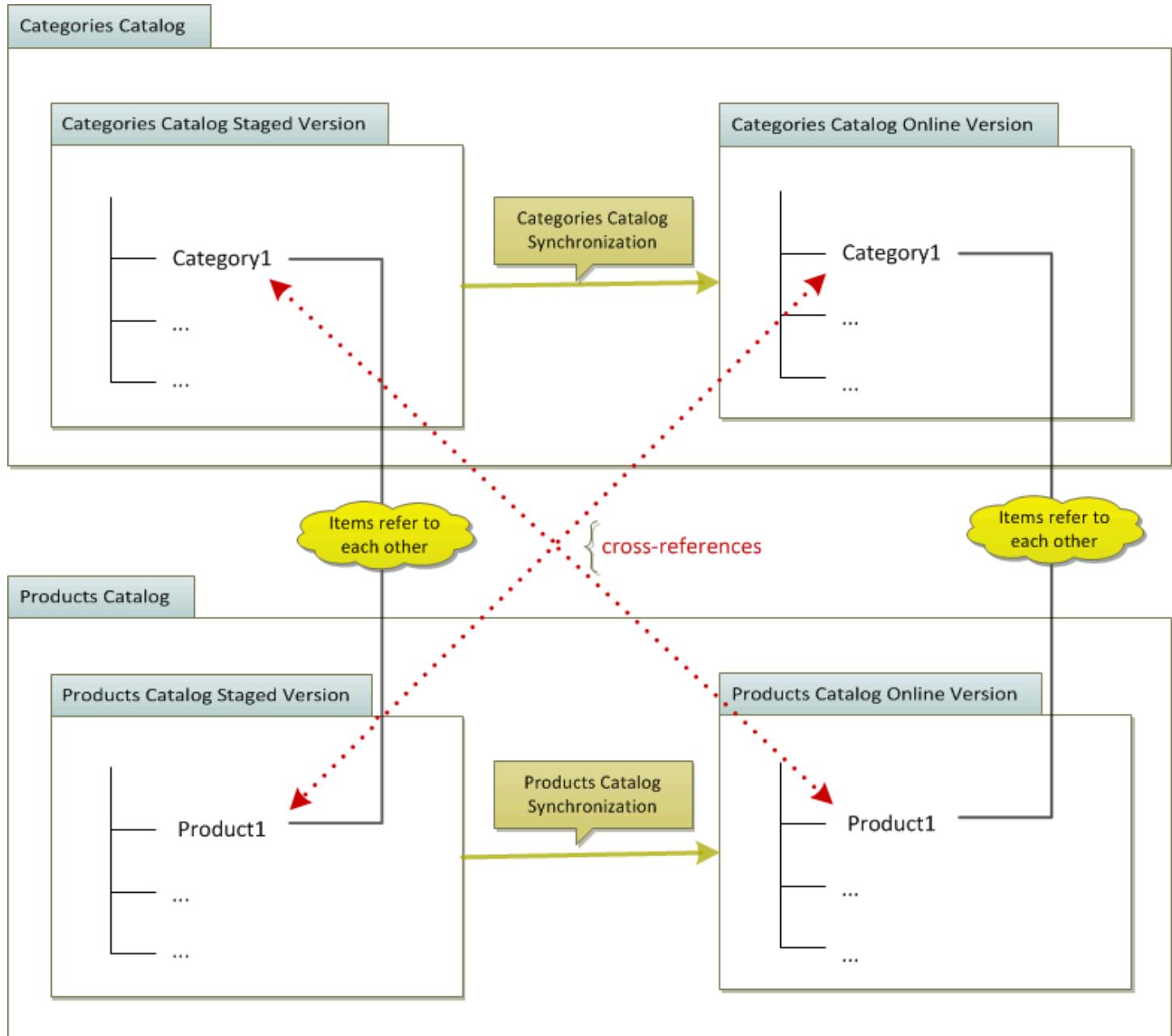


Figure: Catalogs after independent synchronization.

Synchronization with Dependency

If synchronization jobs are dependent on each other, cross-references between items from different catalogs are not created. Setting a dependency between synchronization jobs does not mean that these synchronizations jobs trigger each other.

The diagram below illustrates an example with the same catalogs shown in the Dependend Synchronization section, after running the dependent synchronization:

1. Categories Catalog Synchronization:

- Category1, from the Categories Catalog Staged Version, is copied within the same Categories Catalog to the Categories Catalog Online Version.

- b. As Category1 contains a reference to Product1, from the Products Catalog Staged Version, which is a part of the Product Catalog, it is checked if this synchronization job is dependent or depends on synchronization job from the Product Catalog.
- c. The dependency is configured, the Products Catalog Online Version is checked for the online version of Product1. As the Products Catalog Synchronization was not triggered yet, then the online version of Product1 does not exist. Cross-reference to staged version is not created.

2. Products Catalog Synchronization:

- a. Product1 from the Products Catalog Staged Version is copied within the same Products Catalog to the Products Catalog Online Version.
- b. As Product1 contains a reference to Category1 from the Categories Catalog Staged Version, which is a part of the Category Catalog, it is checked if this synchronization job is dependent or depends on synchronization job from the Category Catalog.
- c. The dependency is configured, the Categories Catalog Online Version is checked for the online version of Category1. As the Categories Catalog Synchronization has already been triggered, the online version of Category1 exists. The reference between online version of the items from different catalogs is created.

After both synchronization jobs are finished, items from online versions of catalogs hold only the references to each other. The situation would look exactly the same if synchronization jobs ran in a different order.

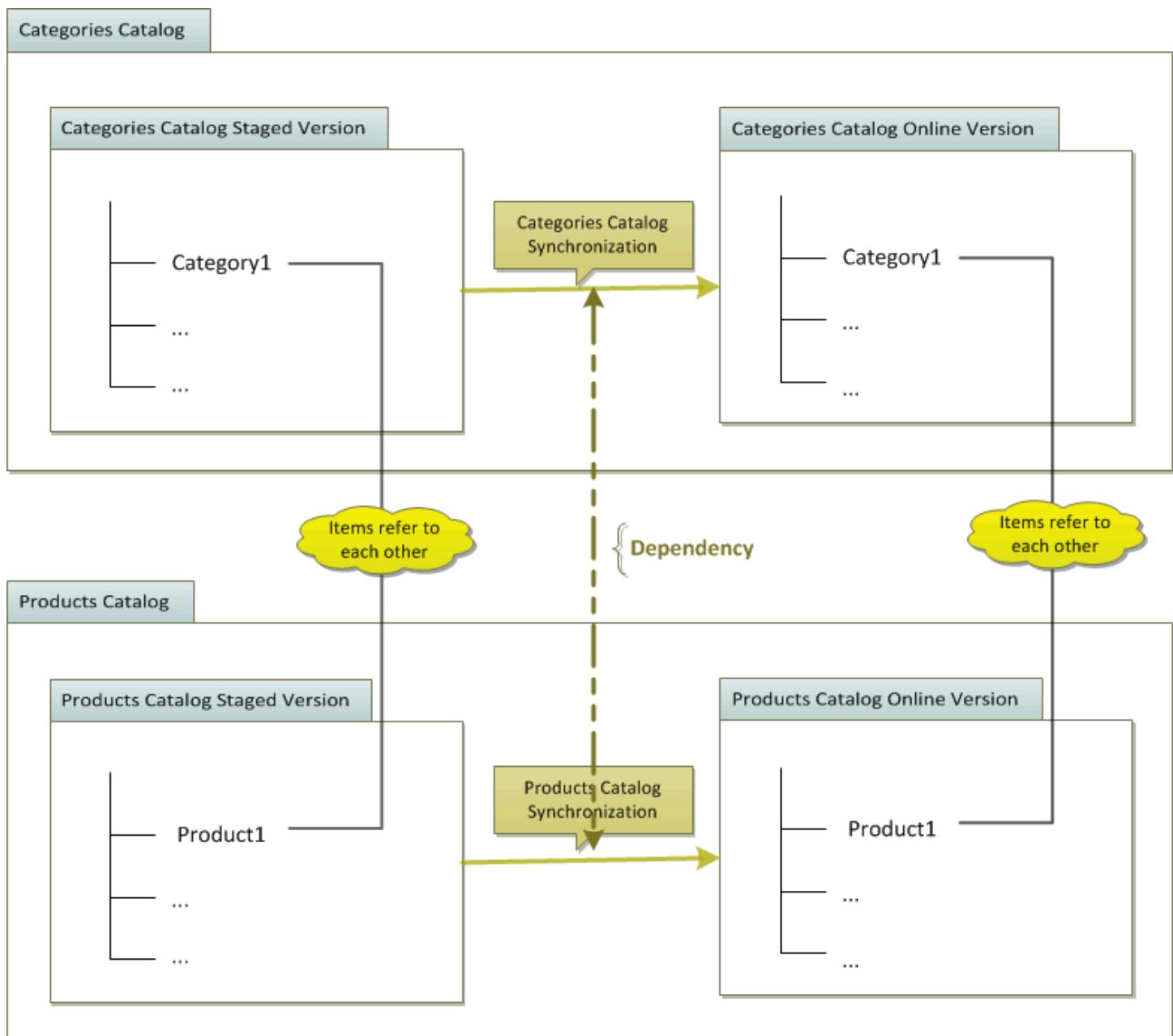


Figure: Catalogs after dependent synchronization.

Configure Dependency in Code

To set a dependency, you need to use one of these attributes: `dependsOnSyncJob` and `dependentSyncJob`. From the code perspective it does not matter which attribute is used because they use many-to-many relations, so dependency works in the same way.

Configure CatalogVersionSyncJobs with Dependency

1. Create synchronization jobs. Creates synchronization jobs for the given catalogs from the staged catalog version to the online catalog version.

```
final CatalogVersionSyncJob category_job = createSyncJob("categoryCatalog", "category_staged"
final CatalogVersionSyncJob product_job = createSyncJob("productCatalog", "product_staged", "
```

2. Set a dependency. Make sure that a rerun of the synchronization always uses a new cron job.

i Note

Every execution of the `CatalogVersionSyncJob` requires a new cron job.

```
//Method that creates for a given synchronization job a new CronJob and configures it automat
private CatalogVersionSyncCronJob createFullSyncCronJob(final CatalogVersionSyncJob job)
{
    final CatalogVersionSyncCronJob cj = (CatalogVersionSyncCronJob) job.newExecution();
    job.configureFullVersionSync(cj);
    return cj;
}

//Now you get your CronJobs for the given jobs
CatalogVersionSyncCronJob category_cj = createFullSyncCronJob(category_job);
CatalogVersionSyncCronJob product_cj = createFullSyncCronJob(product_job);

//Set the dependencies between jobs
final Set<CatalogVersionSyncJob> dependantJobSet = new HashSet<CatalogVersionSyncJob>();
dependantJobSet.add(product_job);

category_job.setDependentSyncJobs(dependantJobSet);
//Alternative use: product_job.setDependsOnSyncJob(dependantJobSet);
```

3. Execute cron jobs asynchronously.

```
category_job.perform(category_cj, false);
product_job.perform(product_cj, false);
```

It is worth noting that after the execution of cron jobs, synchronization jobs of one or both cron jobs may still be running or has not even started yet. Therefore, you need to check if they have finished. See details in *Waiting for Synchronization Jobs* section.

Checking Results of Synchronization and Rerun

It is possible that one of dependent synchronization jobs ends with an error. This is normal, because one job relies on an item which is created by the other job. If this item is not there, this job reports an error. In the second run, this job will be successfully finished, because the item exists. Therefore, after both synchronizations are finished, check the result and do another rerun if needed.

```
EnumerationValue fail = category_cj.getFailureResult();
if (category_cj.getResult().equals(fail) || product_cj.getResult().equals(fail) ) //one of them failed
{
    //You need to create a new CronJob for the rerun
    CatalogVersionSyncCronJob category_cj = createFullSyncCronJob(category_job);
    CatalogVersionSyncCronJob product_cj = createFullSyncCronJob(product_job);
```

```
//Execute them asynchronously
category_job.perform(category_cj, false);
product_job.perform(product_cj, false);
}
```

If, after the second run, the synchronization job still finishes with an error, then it is a real error and it should be investigated.

Waiting for Synchronization Jobs

1. The waiting is divided into two parts:

- Waiting if both synchronization jobs started in a given timeout
- Waiting if both synchronization jobs finished in a given timeout

```
private void checkExecutionOfAsynchronousJobs(final long startTimeOutInMillis, final long
{
    assertAllStarted(startTimeOutInMillis, cronjobs);
    assertAllFinished(executionTimeOutInMillis, cronjobs);
}
```

2. Set waiting if both synchronization jobs started in a given timeout.

```
private void assertAllStarted(final long startTimeOutInMillis, final CatalogVersionSyncCronJob
                            throws InterruptedException
{
    final long timeoutTime = System.currentTimeMillis() + startTimeOutInMillis;
    //check if the cronjob had started
    while (!allStartedOrFinished(cronjobs) && System.currentTimeMillis() < timeoutTime)
    {
        Thread.sleep(500);
    }
    if(!allStartedOrFinished(cronjobs)) throw new RuntimeException("At least one of the given J
}

private boolean allStartedOrFinished(final CatalogVersionSyncCronJob... cronjobs)
{
    for (final CatalogVersionSyncCronJob cronjob : cronjobs)
    {
        if (!(cronjob.isRunning() || cronjob.isFinished()))
        {
            return false;
        }
    }
    return true;
}
```

3. Set waiting if both synchronization jobs finished in a given timeout.

```
private void assertAllFinished(final long endTimeOutInMillis, final CatalogVersionSyncCronJob.
{
    final long timeoutTime = System.currentTimeMillis() + endTimeOutInMillis;
    //check if the cronjob had started
    while (!areAllOfThemFinished(cronjobs) && System.currentTimeMillis() < timeoutTime)
    {
        Thread.sleep(500);
    }
    if(!areAllOfThemFinished(cronjobs)) throw new RuntimeException("At least one of the Jobs di
}

private boolean areAllOfThemFinished(final CatalogVersionSyncCronJob... cronjobs)
{
    for (final CatalogVersionSyncCronJob cronjob : cronjobs)
    {
        if (!cronjob.isFinished())
        {
            return false;
        }
    }
}
```

```
        return true;
    }
```

4. To wait for the above synchronization jobs, they need to be executed in the following manner:

```
//Execute them asynchronously
category_job.perform(category_cj, false);
product_job.perform(product_cj, false);
checkExecutionOfAsynchronousJobs(2000, 5000, category_job, product_job); //waiting at least 2
```

Running CatalogVersionSyncJobs Synchronously

If synchronization jobs are started asynchronously, this means that both jobs run in the same thread, one after one.

To run jobs synchronously do the following:

```
category_job.perform(category_cj, true); //code execution stops here for some seconds till job is finished
product_job.perform(product_cj, true); //code execution stops here for some seconds till job is finished

//here, the jobs could still have an error result, so another single rerun is necessary
//But you do not need to poll the jobs if they are already finished or not.
```

Performing Multithreaded Catalog Synchronization in Backoffice

To support systems with large numbers of products in a multi-catalog scenario, SAP Commerce offers multithreaded catalog synchronization. You can perform it in Backoffice.

Context

Multithreaded catalog synchronization enables you to use as many cores as you like to speed up catalog synchronization operations. You can configure the number of cores for each synchronization job separately. Our tests on 16 core machines have demonstrated an almost linear scaling if the database can handle the load. Follow the steps to configure the multithreaded synchronization.

Procedure

1. To run a multithreaded catalog synchronization, log into Backoffice and navigate to **System** **Multithreaded Synchronization**.
2. Use the button to add a new synchronization configuration.

3. Configure the multithreaded synchronization as regular synchronizations.

4. Once the configuration is ready, navigate to **Administration > Unbound** to adjust the advanced settings.

For synchronization, you can set the following:

- o **Cache size** in entries: Keep the default.
- o **Usage of transactions**: By default set to false.
- o **Number of scheduler threads**: How many threads are used for the scheduling if the medias. Factory default is 1, meaning no multithreading.
- o **The Number of used threads**: Enter the number of the used threads. Factory default is 1, meaning no multithreading.

5. Save and execute the synchronization as usual.

Synchronization of Custom Item Types

SAP Commerce allows you to synchronize catalog versions into another catalog version by defining synchronizations.

[Item Type Requirements](#)

Item types which should be included in the synchronizing process need to meet some basic requirements before it is possible to add them to a synchronization item.

[Marking a Type as Synchronizing Capable](#)

If all requirements are met, the item type can be **marked** as synchronizable.

[Synchronization Strategies](#)

Once our item type has been marked as synchronizable, it will be automatically recognized as such by any synchronizing operation without a need to configure anything else. This means that source version items of this type are translated into their existing target version counterpart items.

[Synchronizing a Single Item](#)

Apart from synchronizing the entire catalog, you can also choose to synchronize single items.

[Troubleshooting](#)

Even though it's much easier to include own item types in the synchronization operation there are still some points to be remembered in case of any issues.

Item Type Requirements

Item types which should be included in the synchronizing process need to meet some basic requirements before it is possible to add them to a synchronization item.

The required settings can be done either manually in Backoffice or declared inside the **items.xml** file.

Catalog Version Home Attribute

First of all, item types must provide a catalog version home attribute typed to **CatalogVersion**. This way, each item of this type knows where it belongs to. This attribute should be initial, mandatory, and read-only if possible. For details on setting **CatalogVersion** as the home attribute, see [Setting CatalogVersion as the Home Attribute for a Type](#).

Presence of a Unique Key Attribute

An item must have one or more unique key attributes. A synchronizing capable item has to be distinguishable within its enclosing catalog version. For example, for the type **Unit** the unique key attribute is **code**.

Setting CatalogVersion as the Home Attribute for a Type

Follow these steps to learn how to set **CatalogVersion** as a home attribute for a type.

Context

In our example, we set **CatalogVersion** as a home attribute for the **Unit** type. If possible, the attribute must be set to initial, mandatory, or read-only.

Once you followed the steps, you will end up with the following configuration:

```
<attribute qualifier="catalogVersion" type="CatalogVersion">
  <modifiers read="true" write="false" search="true" initial="true" optional="false"/>
  <persistence type="property"/>
</attribute>
```

You may decide to make those changes directly in **items.xml** if you don't want to use Backoffice.

Procedure

1. In the Backoffice tree, go to **System > Types** and select a type, for example Unit.
2. On the **Properties** tab, click **Create new Attribute**

code	java.lang.String [java.lang.true]	true	false	true
conversion	java.lang.Double [java.lan]true	true	true	true
name	[localized:java.lang.String]true	true	true	true
unitType	java.lang.String [java.lang.true]	true	false	true

+ Create new Attribute

3. Choose **CatalogVersion** as **Feature** type and edit a **Qualifier** by entering for example **catalogVersion**.
4. Save and the attribute is listed as property of the type.

Marking a Type as Synchronizing Capable

If all requirements are met, the item type can be **marked** as synchronizable.

Context

You can do it manually in Backoffice or this can be declared inside the `items.xml` file.

Procedure

1. In Backoffice go to the **Extended** tab of the type editor.
2. In the **Catalog** group, set the **Is repository capable** to **True**.
3. In the **Repository attribute** drop-down list box, select the catalog version home attribute, for example **catalogVersion**.
4. To the **Repository unique key attributes** list, add the qualifier code of the attribute, for example **code** for **Unit**

The screenshot shows the SAP Fiori interface for managing item types. The 'Unit [Unit]' item is selected. The 'EXTENDED' tab is active. In the 'CATALOG' section, the 'is repository capable' field is set to 'True'. The 'Repository attribute' field contains 'Unit [Unit] -> [CatalogVersion]'. The 'Repository unique key attributes' field contains 'Unit [Unit] -> Identifier [code]'. A yellow box highlights this section.

Alternatively, this can also be done by editing the `items.xml` model file. The following example refers to the default catalog item **Keyword**. It shows the shortened **Keyword** item type declaration, which illustrates how to use the `<custom-properties>` tag to mark the type, define the catalog version, and unique key attributes:

```

<itemtype code="Keyword" ...>
  <custom-properties>
    <!-- marking the type as synchronizing capable here: -->
    <property name="catalogItemType">
      <value>java.lang.Boolean.TRUE</value>
    </property>

    <!-- define catalog version attribute here: -->
    <property name="catalogVersionAttributeQualifier">
      <value>"catalogVersion"</value>
    </property>

    <!-- define unique key attributes here; separate multiple attribute qualifiers by comma -->
    <property name="uniqueKeyAttributeQualifier">
      <value>"keyword"</value>
    </property>
  </custom-properties>
  <attributes>
    <attribute qualifier="keyword" type="java.lang.String">
      <modifiers read="true" write="true" search="true" optional="false"/>
      <persistence type="property"/>
    </attribute>
    ...
    <attribute qualifier="catalogVersion" type="CatalogVersion">
      <modifiers read="true" write="true" search="true" optional="false"/>
      <persistence type="property"/>
    </attribute>
  </attributes>
  ...
</itemtype>

```

Synchronization Strategies

Once our item type has been marked as synchronizable, it will be automatically recognized as such by any synchronizing operation without a need to configure anything else. This means that source version items of this type are translated into their existing target version counterpart items.

Often this is not enough, because a item is not automatically created inside the target version, unless it's part of another item which is copied to the target version.

There are two ways to ensure that items are created inside the target version:

- Using **part-of** mechanism to create/copy them together with enclosing or referring items
- Making it the **root type** of the synchronization to create/copy them without any enclosing or referring item required

Synchronizing with Enclosing or Referring Items Using Part-of Logic

To use the part-of mechanism means that each occurrence of the item type inside an attribute of other synchronized item types (like Product) must be marked as part-of within the synchronization.

Context

Now, items of our type are always treated like part-of items which means they are always created if they do not exist within the target catalog version. To achieve this, use Backoffice to configure synchronization attributes configuration for Multithreaded Synchronization as described in the following steps:

Procedure

1. Log into Backoffice and navigate to **System > Multithreaded Synchronization**.

The result view displays available synchronization jobs.

2. Choose the synchronization job you wish to modify to expand a synchronization editor.

3. On the **CONFIGURATION** tab in the editor area, click **Edit Configuration** button.

Sync Default:Staged -> Default:Online

CONFIGURATION CRONJOBS RESTRICTIONS ADVANCED ATTRIBUTES ADMINISTRATION

ESSENTIAL

Code	Synchronization source	Synchronization target
Sync Default:Staged -> Default:Online	default catalog : Staged	default catalog : Online

SYNCHRONIZATION ATTRIBUTES CONFIGURATION

Synchronization attributes configuration

Included Properties	Excluded Properties
31 Types 224 Attributes	0 Types 0 Attributes

Edit Configuration

4. Expand the tree to find all attributes containing our item type. You can also use the search field to find a specific attribute.

Edit Synchronization Attributes Configuration

Search

Item [Item]

- Assigned Cockpit Item Templates [assignedCockpitItemTemplates]
- Comments [comments]
- Documents [allDocuments]
- Owner [owner]

Abstract Configuration [AbstractAsConfiguration]

- Catalog Version [catalogVersion]
- Unique identifier [uid]
- Boost Item Configuration [AbstractAsBoostItemConfiguration]
- Boost Rule [AsBoostRule]
- Facet Configuration [AbstractAsFacetConfiguration]
- Facet Value Configuration [AbstractAsFacetValueConfiguration]

Details

Synchronize

Copy by value

Untranslatable value

Partially translatable

Attribute Completely Included Partially Included Not Included

CANCEL

5. Enable **Copy by value**.

Because these settings cannot be declared inside items.xml, you should put some code inside the `createEssentialData` method of your extension manager, which performs these changes.

```
public void createEssentialData(Map params, JspContext jspc) throws Exception
{
    ...

    CatalogManager cm = CatalogManager.getInstance();

    // get source and target version of synchronization
    CatalogVersion src = cm.getCatalog( "hwcatalog" ).getCatalogVersion( "staged" );
    CatalogVersion tgt = cm.getCatalog( "hwcatalog" ).getCatalogVersion( "online" );

    // find rule by source and target (provided there is just one!)
    SyncItemJob syncJob = cm.getSyncJob( src, tgt );
```

```

// get attribute which should be treated as part-of
AttributeDescriptor ad = TypeManager.getInstance().getComposedType(
    Product.class
).getDeclaredAttributeDescriptor( "keywords" );

// get or create attribute synchronization config item
SyncAttributeDescriptorConfig cfg = syncJob.getConfigFor(
    ad,
    true /* create on demand */
);

// set part-of behavior to true
cfg.setCopyByValue( true );

...
}

}

```

Synchronizing as Root Type

If our item type describes rather central or standalone items without much connection to other synchronized item types you should consider to add it to the root types of the synchronization instead of tweaking attributes as described before.

Context

All items of synchronization root types are evaluated at the beginning of a synchronizing operation and scheduled for synchronization if:

- No counterpart item exists within the target catalog version, unless this has been disabled.
- The source item modification time stamp is newer than last synchronizing time.
- An item exists within the target catalog version but no longer in the source catalog version (by default this is disabled).

You can set add the new item to synchronization root types in the Multithreaded Synchronization in Backoffice.

Procedure

1. Navigate to **System > Multithreaded Synchronization**.
2. Select the synchronization from the list.
3. Go to **Administration > Unbound**
4. Add the new type to the Root types list.

The screenshot shows the SAP Fiori Sync interface with the 'Sync Default:Staged > Default:Online' title bar. The 'ADMINISTRATION' tab is active. On the left, there's a list of items: Configurable Search Configuration [Abstract], Unit of Measurement [ClassificationAttribute], Mini Cart Component [MiniCartComponent], Account Navigation Component [NavigationComponent], and Product [Product] -> Approval [approvalStatus]. Below this is a button to 'Add new item'. In the center, there's a 'Request abort step' section with three radio buttons: 'True' (unchecked), 'False' (checked), and 'N/A'. To the right, a yellow-bordered box contains a 'Root types' section with five entries: Category [Category], Product [Product], Media [Media], Keyword [Keyword], and Tax row [TaxRow]. A '...' button is at the bottom right of this box.

5. Of course this may also be put in code, preferably inside the `createEssentialData` method of your extension manager.

```
public void createEssentialData(Map params, JspContext jspc) throws Exception
{
    ...
    CatalogManager cm = CatalogManager.getInstance();

    // find source and target version
    CatalogVersion src = cm.getCatalog("hwcatalog").getCatalogVersion("staged");
    CatalogVersion tgt = cm.getCatalog("hwcatalog").getCatalogVersion("online");

    // find synchronization by source and target version
    SyncItemJob syncJob = cm.getSyncJob(src, tgt);

    // get modifiable list of root types
    List<ComposedType> rootTypes = new ArrayList<ComposedType>( syncJob.getRootTypes() );

    // insert our type at first position
    rootTypes.add(
        0,
        TypeManager.getInstance().getComposedType("Keyword")
    );

    // dont forget to save them again
    syncJob.setRootTypes( rootTypes );
    ...
}
```

Synchronizing a Single Item

Apart from synchronizing the entire catalog, you can also choose to synchronize single items.

It is always possible to trigger synchronizations for single items if:

- A synchronization exists for the selected item's catalog version.
- The current user has got the permission to change the target catalog version.

For details on synchronizing single items, see [Synchronizing Items](#).

For details on synchronizing catalogs, see [Synchronizing Catalogs](#).

Troubleshooting

Even though it's much easier to include own item types in the synchronization operation there are still some points to be remembered in case of any issues.

Bear in mind the following notes while working with synchronization of custom items:

- Dependencies: Real lifetime dependencies (for example between `Order` and `OrderEntry`) must be obeyed when configuring synchronization root types: the parent item type must be placed before its part-of item type.
- Unique key behavior differences: Sometimes unique key constraints are put down into code and cannot be changed (for example for `Language.isoCode`), no matter what we're trying to configure at its composed type; synchronization just checks configured constraints but item creation will most likely throw errors.
- Cyclic part-of configuration: When using the synchronization attribute configuration to mark some attributes as part-of this may collide with other (real or configured) part-of attributes; synchronization will fail whenever a part-of item tries to copy its parent (or any of its parents) as part-of too.
- Root type list contains super and subtypes: Make sure each new synchronization root type is not subtype of any existing root type; otherwise synchronization will at least perform slow or may fail completely.
- Modification time issues:
 - Keep in mind that changing a part-of item does not mark its owning item modified automatically. To allow synchronization to schedule the owning item correctly you have to mark it modified manually, preferably inside the part-of item's business code.
 - On the other hand relation attributes and plain property backed attributes do mark the enclosing item modified.
 - Since there is just one modification time for each item the synchronizing operation may only notice that something has changed, but not which attributes. So even if the synchronization omits several attributes a modified item will always be synchronized regardless, which attributes are actually changed.
- Root types affect full synchronization only: When triggering synchronization for single items (via search result and so on) the synchronizing operation will publish these items only. Of course all synchronization attribute settings are obeyed here as well.

Catalog Synchronization API

You can start catalog synchronization processes and get their status via API.

i Note

The actual setup of a `CatalogSyncJob` is not in scope here.

Starting Full Catalog Synchronization

`CatalogSynchronizationService` offers multiple methods for starting the synchronization process.

Ad-Hoc Full Synchronization

There are methods for creating and starting sync jobs ad hoc. You can use them when there are no pre-existing `CatalogVersionSyncJob` items set up for a given pair of catalog versions:

```
CatalogSynchronizationService css = ...
CatalogVersionModel source = ...
CatalogVersionModel target = ...

// create a job and a cron job and execute synchronously using the default amount of worker threads
css.synchronizeFully( source, target )

// create a job and a cron job and execute synchronously using 16 worker threads
```

```
css.synchronizeFully( source, target, 16 )

// create a job and a cron job and execute in background using the default amount of worker threads
css.synchronizeFullyInBackground(source, target)
```

! Restriction

This is recommended for testing only as sync jobs are heavy weight and shouldn't be created more than once per catalog version pair.

Full Synchronization for an Existing Sync Job

The regular use case of starting synchronization is based on an existing `CatalogVersionSyncJob`, containing all relevant settings (for example number of workers, root types, attributes).

You can pass additional settings via a `SyncConfig` parameter:

```
CatalogSynchronizationService css = ...
CatalogVersionSyncJobModel syncJob = ...

// configuring
SyncConfig cfg = new SyncConfig();
cfg.setSynchronous( false ); // background
cfg.setForceUpdate( true ); // update all even if timestamps are the same

css.synchronize( syncJob, cfg );
```

Starting Partial Synchronization

Simplified Partial Synchronization API

The service offers methods for use cases where only a subset of source catalog items is supposed to be processed:

```
CatalogSynchronizationService css = ...
CatalogVersionSyncJobModel syncJob = ...
SyncConfig cfg = ...

// start partial sync for given items and specific job
List<ItemModel> itemsToSync = ...
css.performSynchronization( itemsToSync, syncJob, cfg );

// start multiple partial sync processes for given items from multiple catalog
List<ItemModel> itemsFromMultipleCatalogs = ...
List<CatalogVersionSyncJobModel> multipleSyncJobs = ...
css.performSynchronization( itemsFromMultipleCatalogs, multipleSyncJobs, cfg );
```

i Note

Filtering of Items

The given items are always **filtered** to match the specified sync jobs. Items that don't match their source or targets catalog versions are silently ignored! This also means that when multiple sync jobs are specified, items may be accepted by one job but

ignored by another.

i Note

Scheduling Removal

In case of items **removed** from their source catalog versions, schedule their **target catalog version counterparts** in order for the sync process to correctly remove them. Since they're recognized as target items automatically, you can simply add them to the item list.

Partial Synchronization via SyncConfig API

Partial synchronization can also be started using SyncConfig:

```
CatalogSynchronizationService css = ...
CatalogVersionSyncJobModel syncJob = ...

List<ItemModel> sourceItemsToSync = ...
List<ItemModel> targetItemsToRemove = ...

SyncConfig cfg = new SyncConfig();
cfg.setSynchronous( false ); // background

// add source items for insert or update
for( ItemModel item : sourceItemsToSync )
{
    cfg.addItemToSync( item.getPk() );
}

// add (orphan) target items for removal
for( ItemModel item : targetItemsToRemove )
{
    cfg.addItemToDelete( item.getPk() );
}

// start it
css.synchronize( syncJob, cfg );
```

Getting Process Information

Most methods actually return SyncResult objects, which allows you to inspect the status of a started sync process:

```
CatalogSynchronizationService css = ...
CatalogVersionSyncJobModel syncJob = ...
SyncConfig cfg = ...

// start it
SyncResult sr = css.synchronize( syncJob, cfg );

// get sync cron job
CatalogVersionSyncCronJob syncCronJob = sr.getCronJob();

// if started synchronous: check result
```

```

if( sr.isFinished() && sr.isSuccessful() )
{
    // great
}

// if started in background: poll for status - note that models need to be refreshed to reflect changes
ModelService modelService = ...

CatalogVersionSyncCronJob syncCronJob = sr.getCronJob();
while( !sr.isFinished() )
{
    Thread.sleep( 1000 );
    modelService.refresh( cronJob() );
}

```

Catalog Item Synchronization Status API

You can look up synchronization statuses for specific items contained within catalog versions.

A `CatalogVersionSyncJob` specifies a pair of `CatalogVersion` items as **source** and **target** catalog versions. You can only obtain synchronization statuses for items contained in either of them.

Such a sync job also defines a list of `ComposedTypes` (the so called **root types**) that determine which type of catalog-version-contained items are actually subject to being synchronized by this job. Therefore, only items that are of one of the root types of a given sync job can have synchronization statuses.

Synchronization Statuses

The statuses that items within the source and target versions (using `SyncItemStatus` codes) can have, include:

Source catalog version	Target catalog version	Status	Comment
Item X exists.	No copy of item X exists.	COUNTERPART_MISSING	The next sync run will create a copy of X within the target catalog version.
Item X exists.	A copy of item X exists with the same content as X.	IN_SYNC	Items are in sync - nothing to do.
Item X exists.	A copy of item X exists with content older than X.	NOT_SYNC	The source item has been changed since last sync - next sync will update the copy of X.
Item X was removed.	A copy of item X still exists	COUNTERPART_MISSING	The next sync run will apply the source item removal and will remove the copy of X as well.

Looking up Statuses for Items

To look up an individual status information for a given item and a sync job, use `SynchronizationStatusService`:

```

// given
SynchronizationStatusService sss = ...
ItemModel myItem = ...
CatalogVersionSyncJobModel syncJob = ...

// get info
SyncItemInfo info = sss.getSyncInfo( myItem, syncJob);

switch( info.setSyncStatus() )
{
    case COUNTERPART_MISSING:
        if( info.getFromSource() ){
            // new item to be copied
        }else{
            // leftover target item without source
        }
        break;
    case NOT_SYNC:
        // item exists on both sides but need update
}

```

For any pair of items being synchronized, an `ItemSyncTimestamp` item is created for tracking their status. You can look up that item using `SyncItemStatus`:

```

// given
SyncItemInfo info = sss.getSyncInfo( myItem, syncJob);
ModelService modelService = ...

// get last sync pending attributes
PK tsPK = info.getSyncTimestampPk();
if( tsPK != null )
{
    ItemSyncTimestampModel ts = modelService.get(tsPK);
    for( AttributeDescriptorModel pending : ts.getPendingAttributes() )
    {
        // got it
    }
}

```

To look up multiple statuses for lists of items or for multiple sync jobs, use bulk methods:

```

// given
SynchronizationStatusService sss = ...

// bulk get for one item with multiple sync jobs set up
ItemModel item = ...
List<ItemSyncJobModel> syncJobs = ...
List<SyncItemStatus> states = sss.getSyncInfo( item, syncJobs );

// bulk get for multiple items with multiple sync jobs set up
ItemSyncJobModel syncJob = ...
List<ItemModel> items = ...
List<SyncItemStatus> states = sss.getSyncInfo( items, syncJob );

```

i Note

Note that the bulk methods check whether the given items are actually applicable to the given sync job or not. If they're not, information with `SyncItemStatus.NOT_APPLICABLE` is returned.

Looking up Sync Jobs for Items

Sometimes you may want to find out which sync jobs are actually responsible for a given item, either using it as source or target. For that purpose, the service offers these methods:

```
// given
SynchronizationStatusService sss = ...
ItemModel myItem = ...

// where do we sync this to?
List<SyncItemJobModel> syncFromHereJobs = sss.getOutboundSynchronizations();

// where do we get data synced from?
List<SyncItemJobModel> syncToHereJobs = sss.getInboundSynchronizations();
```

Parallel Catalog Synchronization

Full catalog synchronizations cannot run in parallel. It is possible, however, to start multiple partial sync CronJobs on condition the items you want to synchronize don't overlap.

You can pick one or more products from a catalog and synchronize them using a chosen catalog version sync job. The `CatalogVersionSyncCronJob` gets the `fullSync` attribute. This attribute is set by service during its creation, and contains information whether a given cron job performs a partial or a full synchronization.

There are some points to consider when starting an A catalog synchronization job:

- if another B. `fullSync == true` sync job is running --> failure ("Cannot perform partial sync cron job when there is other full sync cron job running")
- if A. `fullSync == true`, and any other sync is running --> failure ("Cannot perform full sync cron job when others are running")
- if A. `fullSync == false`, and no other sync is running --> success
- if A. `fullSync == false`, and there is another sync B with B. `fullSync == false` is running -> compare schedules:
 - overlap --> failure ("Cannot perform partial sync when there are items overlapping from other running cron job")
 - The number of items exceed the limit (`catalog.sync.partial.max.items` default value set to 500) --> failure ("Cannot perform partial sync when number of items to sync exceed the limit")
 - no overlap --> success

If you set the `abortOnCollidingSync` attribute to `true`, it sets a cron job status to `ABORTED` when the given cron job fails to pass the mentioned validation, otherwise the status stays as `UNKNOWN` for backward compatibility.

When checking for overlapping items with other running cron jobs, the items are taken into account as of the beginning of the cron job. So even if a cron job did synchronize most of its items, the overlap will be checked for all of them.

Synchronization Properties

These are some of the properties that you can use to configure your synchronization, or that are related to the synchronization process.

Property Name	Default Value	Definition
catalog.sync.turns.without.locking	5	Sets the number of turns without locking.
synchronization.dumpfile.tempdir	<\${HYBRIS_TEMP_DIR}>/sync	Defines a directory to store dump files. If the property value is empty, the synchronization will use the system's temporary directory.
synchronization.within.deployment	true	Allows you to define how synchronization is performed during deployment. This means that objects for synchronization are optimized for deployment, such as synchronized taking into account the current state of the system.
synchronization.itemcopycreator.stacktraces	false	Logs stacktraces from the de.hybris.platform.catalog.CatalogItemCopyCreator.create method on INFO level.

Classification

Classification enables you to define product attributes in a way different to the typing method.

Classification-based attributes are called **category features**, sometimes also referred to as **classification attributes**.

Through classification, you can flexibly allocate category features that may change frequently. Defining and modifying them is easy because they are managed independently of the product type.

Available category features can be organized independently from product catalog structures in separate **classification systems**, where they can be structured into **classifying categories**. Using classifying categories, you can assign a category feature to one or multiple product categories used in catalogs. That means that all category features of the classifying categories are available within all products included in the assigned product categories. In addition, each category feature assigned to a category of a catalog structure is inherited by all subcategories.

Classification Basics

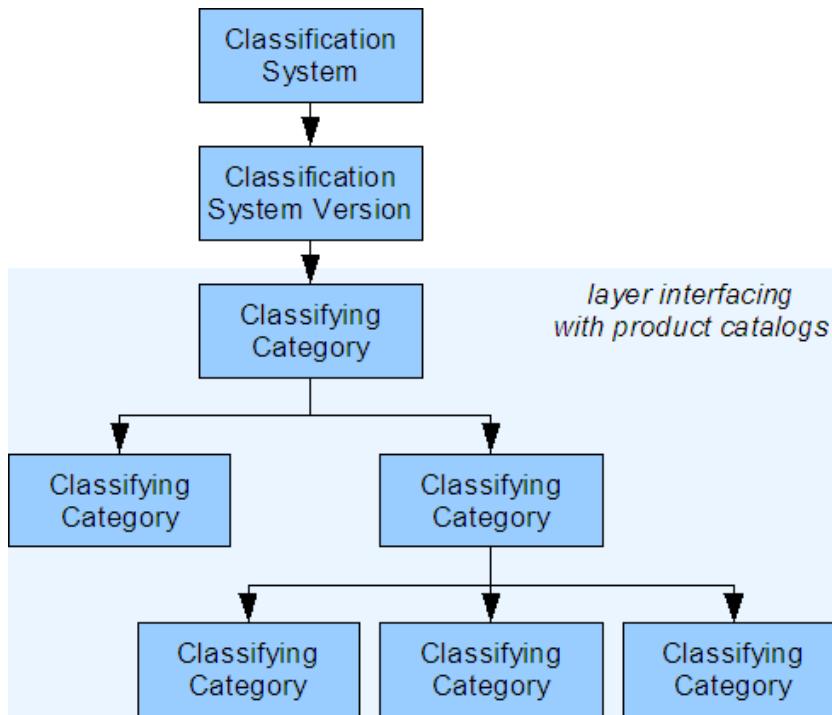
Classification helps simplify the management of product data independently of product catalogs or product definitions. To that aim, a classification system preferably takes the form of a hierarchical structure characterized by an appropriate number of levels of detail according to the range of product data to be covered. This hierarchical structure is achieved by classifying categories similar to product categories of a catalog. A classification system's independence from product catalogs, combined with the ability to inherit features from higher levels within its classification structure, makes it easy to assign product-related attributes.

Such classification attributes, called category features in platformbackoffice, function as a kind of container that eventually holds the product-specific value of the corresponding attribute. For example, this means that while the category feature defines the weight (and maybe also kg as a relevant unit of weight), the actual weight of a product is never part of the classification system itself; but it resides with the product(s) in the product catalog. Therefore the actual product attributes (as displayed in product catalogs) exist only by allocating category features of the classification system to product categories or individual products. The benefit of such an allocation is that the category feature - product attribute construct is and remains dynamic throughout the product catalog's life cycle.

Structuring Classification Systems

In the `platformbackoffice` extension, a classification system takes a form similar to that of ordinary product catalogs, that is, a classification system has one or more classification system versions that themselves contain any number of classifying categories organized in a tree-like structure. Classifying categories also represent the layer where classification systems and product catalogs interface.

The following diagram shows how the SAP Commerce catalogs organize classification data:



Classifying categories hold the product information. The position of a classifying category within its tree structure indicates the level of detail of product information it has; more general data is provided closer to the root and more detailed data is provided further down the tree.

Example of a single branch of a simple classifying category structure:

Category Level	Assigned Category Features	Applies to
Classifying Category Level 1	General: weight, measurement	All products
Classifying Category Level 2	TV/Video related: standard (PAL etc.)	All TVs, all DVD players/recorders
Classifying Category Level 3	DVD Players/Recorders related: region code	All DVD players/recorders (but not to TVs)
Classifying Category Level 4	DVD Recorder related: hard disk capacity	All DVD recorders (but not players)

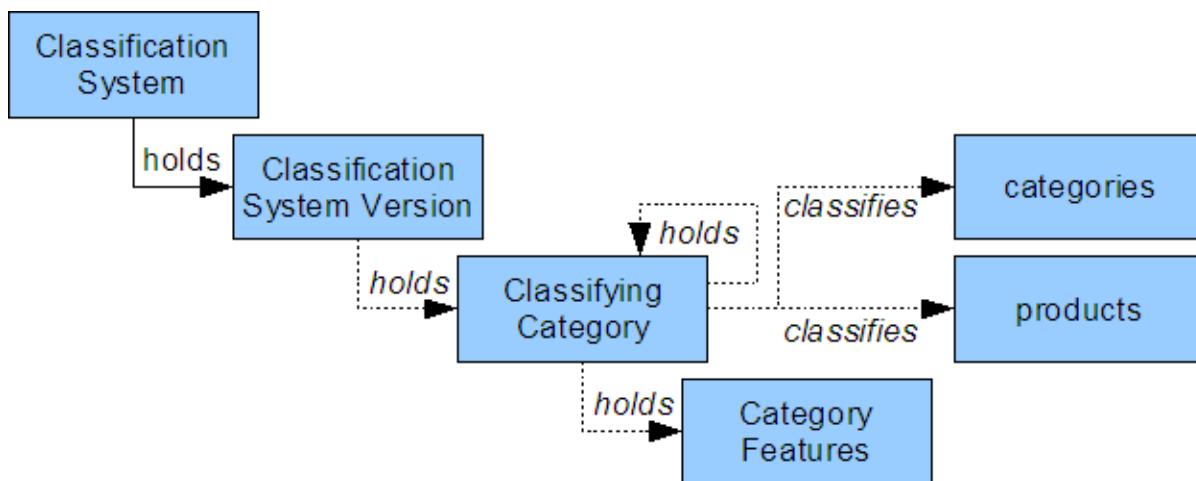
Not provided in our example are the possible side branches that would might exist in such a structure, for example: audio-related data at Classifying Category Level 2, TV only related data on Classifying Category Level 3 or DVD Players related data on Classifying Category Level 4.

SAP Commerce Classification Overview

This section provides a more detailed description of the various components of a classification as they are represented in SAP Commerce. A classification attribute, referred to as category feature in `platformbackoffice`, is part of a classification system and is a loose combination of the following:

- An attribute label, referred to as feature descriptor.
- A feature descriptor type, which specifies the kind of value.
- Optionally, pre-selected values allowed for the classification attribute, referred to as feature values.

Category features are grouped in classifying categories, which are grouped in classification systems through the classification system versions.



Classification System

A classification system must contain at least one classification system version in order to hold classifying categories. In the case of great variety within the product spectrum to be classified and/or more than one product catalog existing, it can be useful to create more than one classification system. For details on how to create a classification system, see [Creating Classification Systems](#).

Classification System Version

A classification system version can contain any number of classifying categories. The ACTIVE flag on the version makes it possible to define which version's assigned category features are displayed in the product catalog. Only one classification system version can be active for a classification system at a time. For details on how to create a classification system version, see [Creating Classification System Versions](#).

Classifying Category

A classifying category can contain one or more classifying categories and/or category features. Similar to product categories, the classifying categories can be organized in a tree-like structure in order to organize the category features that they carry in a meaningful hierarchical manner. The classifying category tree structure forms the heart of the whole classification and it should therefore be well designed for maximum benefit in its interaction with the product catalog. The category features defined within the classifying category are displayed as product attributes once allocated to the product catalog. Category features do not appear at the product categories themselves, only at the products. This means that you usually manage the values of the category features in platformbackoffice through the product view. For details about how to create a classifying category, see the [Creating Classifying Categories](#).

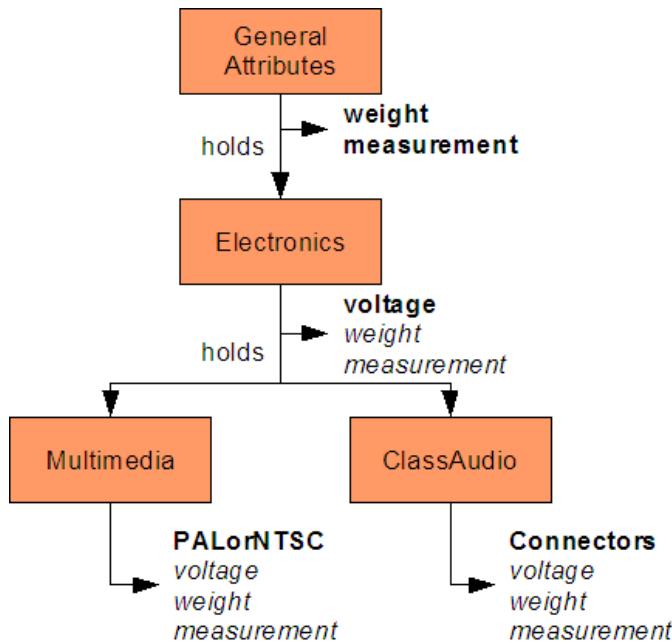
Classifying Category Hierarchies

If you create a classifying categories tree, the category features are passed on downwards (that is, towards the end of the tree), from one classifying category to its children. Each classifying category has all the category features that its direct parents have. This means that the root of the tree only has the category features defined for itself, while the endmost nodes will have the most category features.

The following diagram provides an example of a classifying categories tree:

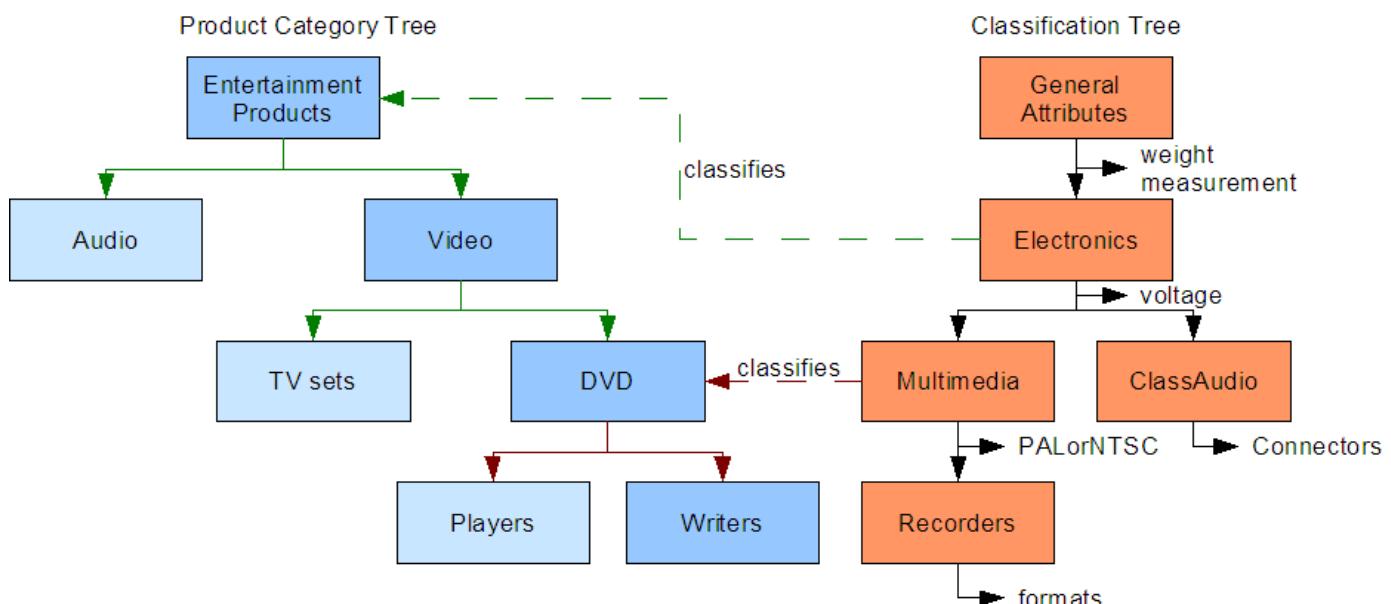
The classification system has a root classifying category called General Attributes that defines **weight** and **measurement** as its category features (shown in bold as they are defined here). General Attributes has a sub-classifying category Electronics that also have **weight** and **measurement** (shown in italics as they are passed on). Electronics also defines a category feature, **voltage**. All products classified as Electronics (whether directly or through a category) have these three category features.

This kind of inheritance only works downwards, not upwards or sideways. The two sub-classifying categories of Electronics (Multimedia and ClassAudio) each define a category feature (PALorNTSC and Connectors, respectively). Multimedia does not inherit the Connectors category feature, nor does ClassAudio inherit the PALorNTSC category feature because inheritance does not work sideways. Electronics does not inherit PALorNTSC or Connectors because inheritance does not work upwards.



Classifying Product Categories

Classification of product catalogs takes place through linking classifying categories (and thus their category features by extension) with product categories. Product categories normally also form a hierarchy. By classifying a product category, any subcategory of that product category is classified in the same way (only downwards again). For example, let's take our previous classification example and add a product category tree:



In the diagram, two product categories are classified: Entertainment Products with Electronics and DVD with Multimedia. Let's look at the Entertainment Products product category first. The fact that it is classified with Electronics means that every product category (and, therefore, every product) that is a subcategory of Entertainment Products is classified as Electronics as well.

Therefore, every such subcategory gains all category features of Electronics (weight, measurement, and voltage) - Audio, Video, DVD, and TV sets, in the example. This classification is indicated by the green connectors. The DVD product category and its subcategories and products is somewhat special as it is classified as Multimedia. If DVD wasn't classified as Multimedia, DVD and its subcategories Players and Writers would be classified as Electronics as well.

The DVD product category is classified as Multimedia. This means that the category features of Multimedia are passed on to DVD (and override the classification as Electronics): the directly defined PALorNTSC and the passed-on ones weight, measurements, and voltage. All of these category features are also available for the subcategories of DVD: Players and Writers. This is indicated by the red connectors.

Passed-down category features do not "add up" if the classifying categories overlap in the classification hierarchy or reside in different branches of the classification tree. Instead, the most specific classifying category applies to a product category - that is, the classifying category in the shortest direct hierarchy within the classification system. For example, if the Writers product category was classified as ClassAudio, it would not receive the category features of the Multimedia classifying category: the more direct classification (ClassAudio) would override the passed-on classifying category (Multimedia) and provide its category features (Connectors, voltage, weight, and measurement) instead.

There are two ways of actually classifying a product category. Both have the same result: a product category is assigned to be a child of a classifying category. The difference is in the way the link is established: one establishes the link from the product category, the other from the classifying category. As both ways have the same result, which you should choose depends on the number of product categories you want to classify.

- Setting the product category to be a subcategory of a classifying category: In Backoffice, open the classifying category and assign the product category to its **Containing Categories** field. This approach is especially useful if you want to classify several product categories by one classifying category.
- Setting the classifying category to be a supercategory of the product category: In Backoffice, open the product category and assign the classifying category to its **Supercategories** field. This approach is especially useful if you want to classify one product category by several classifying categories.

Category Feature

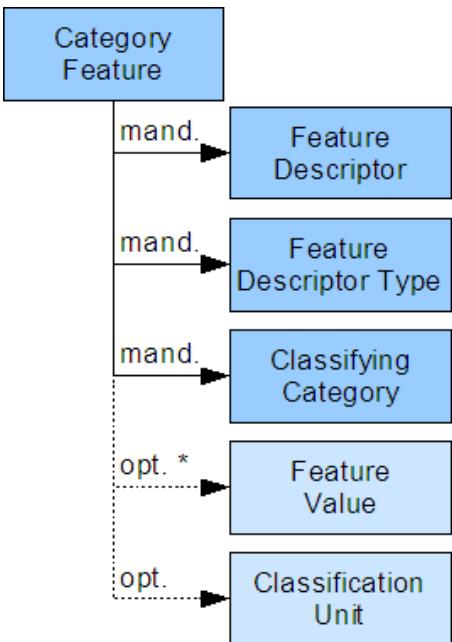
Think of a category feature as a sort of container for the actual value of an attribute as it appears with a product. Even though the linking to the product catalog does occur at the level of the classifying categories, the category features are the only part of the classification system that actually appear in the product catalog and carry product specific information (e.g. the actual amount of kilograms of a product displaying a category feature weight allocated to it).

A category feature is a conglomerate of a number of separate objects, not all of which are necessary in any case:

There are three mandatory objects (marked **mand.** in the diagram). You can only create a category feature if all of these three settings are specified:

- Classifying category: The classifying category holding the category feature
- Feature Descriptor: The label or title for the category feature
- Feature Descriptor Type: Defines what form a product's attribute eventually takes:

Feature Descriptor Type	Allows display of
String	Descriptions and values in text form
Number	Numerical values
Boolean	Boolean values selectable via radio buttons (Yes, No, and n/a)
ValueList	Predefined value(s) selectable via a drop-down menu.



As explained above, normally the actual value pertaining to a category feature is entered and managed in the product catalog (at the product) and not in the classification system. It is however possible to define a set of such values within a category feature through the use of Feature Values. The category feature editor allows the creation and management of a list of feature values (tab Feature Values). Unlike other objects related to a category feature, it is possible to have several feature values (1:m relation, marked with an asterisk (*) in the diagram). All components of that list are selectable at the respectively classified product(s) in the product catalog under the condition that the feature descriptor type for that category feature is set to "Valuelist".

Further settings for category features can be edited as indicated in the sections **Properties/Visibility of the attribute** under the **Common** tab of the category feature editor. For details on how to create a category feature, see [Creating Category Features](#).

Setting	Allowed values	Description / Comment
Classifying Category (mandatory)	A classifying category	The classifying category this category feature belongs to.
Feature Descriptor (mandatory)	A feature descriptor (auto-complete field, "find as you type")	The feature descriptor for this category feature. For more details on feature descriptors, please refer to the next section of this document.
External ID	Text	
Feature Descriptor Type (mandatory)	String, Number, Boolean, ValueList	The kind of value this category feature allows.
Unit	One of the classification units in the classification system the category feature belongs to	
Format definition for numbers		Please also refer to https://docs.oracle.com/javase/8/docs/api/java/text/DecimalFormat.html This is an advanced function.
Mandatory	true (checked) or false (unchecked)	
Multilanguage	true (checked) or false (unchecked)	

Setting	Allowed values	Description / Comment
Multi-valued	true (checked) or false (unchecked)	
Range	true (checked) or false (unchecked)	
Visibility		
Allow search	true (checked) or false (unchecked)	
Show in list view	true (checked) or false (unchecked)	
Show in compare view	true (checked) or false (unchecked)	

Feature Descriptor

A features descriptor defines the label or, in other words, the title of product attributes. The values are localized according to the languages existing in the system. When deciding on the naming, keep in mind that the values of the features descriptor's label represent the attribute titles that are displayed at the product. For details about how to create a features descriptor, see *Creating a Feature Descriptor* section of the [Creating Category Features](#).

Feature Value

A possible value for a **category feature**. There is no dependency between a features descriptor and a feature value per se, it is only via a category feature that a link is established. Feature values are used for supplying one or more default values for product attributes through category features. Note that this is an exception to the normal pattern where the category feature only supplies the container or frame of the attribute, but not its value(s). When feature values are part of a category feature it is mandatory to set the feature descriptor type of that category feature to "Valuelist" as explained above. Only then is a drop-down list of the defined feature values displayed.

Classification Units

The value of a category feature can optionally be assigned a classification unit. This is a specialized, classification system-specific kind of unit that is not related to the units assigned to products directly. Therefore, you cannot assign a product unit to a category feature or vice versa. The classification unit does not relate to a product, but to a category feature. By setting a conversion factor, you can make various classification units become convertible into each other (such as 1 cm is 0.1 dm or 0.01 meters). In other words, a classification unit says "this attribute value is given as a measurement of X".

Using Several Classification Systems at Once

The figure gives an example of how different classification systems (represented by Classification Trees A and B) can be used within the same product catalog: All categories and products by Manufacturer A (which is a category itself actually) are classified as Machines. Electricity defines a category feature **current**, which is inherited by all its sub-classifying categories. As Machines is a subcategory of Electricity, **current** also applies to Machines. The classifying category Software on the other hand is on the same category level as Machines, and therefore does not affect Machines. In the same way, the classifying category Photography of the other classification system does not affect products classified as CPU or the other way round.

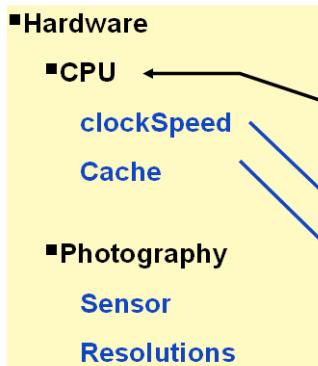
The category Laptops of Manufacturer A is classified as CPU. By consequence, the category features of both the classifying categories CPU and Machines apply to every product that belongs to the category Laptops. Therefore, all category features of both CPU and Machines (and Electricity, through inheritance) apply to Laptops and its contents: **clockSpeed**, **Cache**, **current**, **size**, and **weight**.

Since Product A belongs to Laptops, clockSpeed, Cache, current, size, and weight all appear with Product A.

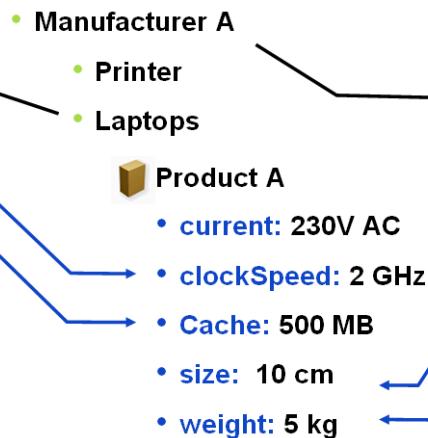
As shown with category Manufacturer A inheritance of category features also works along (sub)categories of product catalogs with more than one classification system. Classified product categories and their contents benefit from all the provided category features.

Multi Classification

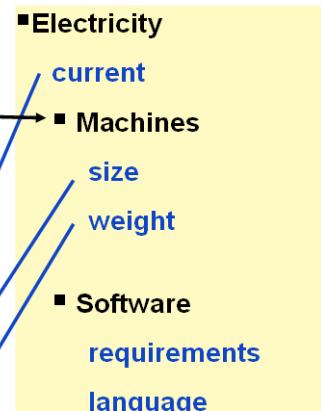
Classification Tree A



Category Tree



Classification Tree B



Classification Attributes can be assigned from several classification trees

As it is possible to classify a product category, it is also possible to classify an individual product by setting the classifying category to be the product's supercategory. As that classification is effective for that single product only, however, this approach is not recommended. By classifying product categories, you are able to classify entire groups of products, which is far more flexible.

Working with Classification

Learn to create various parts of a classification system, from the classification system itself down to a category feature and classifying a product category.

[Adding a Product to a Classifying Category](#)

Backoffice Administration Cockpit enables you to add a product to a classifying category.

[Classifying a Product Category](#)

You can classify a product category in Backoffice Administration Cockpit.

[Creating Category Features](#)

The Classification functionality of SAP Commerce allows you to assign attributes more dynamically than using the SAP Commerce's type system.

[Creating Classification Systems](#)

In SAP Commerce, you can create classification systems, blank classification systems that you can use to create a classification hierarchy, and predefined classification systems.

[Creating Classification System Versions](#)

Creating a classification system version is only possible in the context of a catalog. A classification system version cannot exist outside a catalog version.

[Creating Classifying Categories](#)

You can easily create a new classifying category in Backoffice Administration Cockpit.

Adding a Product to a Classifying Category

Backoffice Administration Cockpit enables you to add a product to a classifying category.

Procedure

1. Log into Backoffice.
2. Go to **Catalog > Classification Systems > Classifying Categories**.
3. Select a category to which you want to add your product.

Identifier	Name	Sync stat...	Classification System Version
5364	Weight & dimensions	●	Powertools Classification : 1.0
5363	Energy management	●	Powertools Classification : 1.0
5362	Technical details	●	Powertools Classification : 1.0
5361	Packaging content	●	Powertools Classification : 1.0
5360	Energy management	●	Powertools Classification : 1.0
5359	Weight & dimensions	●	Powertools Classification : 1.0
5358	Technical details	●	Powertools Classification : 1.0
4969	Technical details	●	Powertools Classification : 1.0
4916	Packaging content	●	Powertools Classification : 1.0
4915	Energy management	●	Powertools Classification : 1.0
4914	Weight & dimensions	●	Powertools Classification : 1.0
4913	Technical details	●	Powertools Classification : 1.0
4850	Technical details	●	Powertools Classification : 1.0
4783	Technical details	●	Powertools Classification : 1.0
4743	Packaging content	●	Powertools Classification : 1.0
4742	Packaging data	●	Powertools Classification : 1.0
4740	Weight & dimensions	●	Powertools Classification : 1.0
4739	Technical details	●	Powertools Classification : 1.0

1 ITEMS SELECTED

4. Go to the **Category Structure** tab.
5. Scroll down to the **Subcategories and Products** section.
6. Click the **... icon under Included Products** to display a list of the products.

7. Select the check mark next to the products you wish to add to the category. You can select more than one product.
8. Click **Select** to close the window and add the products.

9. Click **Save** to save your assignment.

Classifying a Product Category

You can classify a product category in Backoffice Administration Cockpit.

Procedure

1. Log in to Backoffice.
2. Go to **Catalog > Classification Systems > Classification Categories**.
3. Select a classification category from the list, or search for the classification category using the **Search** field.
4. Go to the **Category Structure** tab for the selected classification category.
5. Click the **...** icon next to the empty field under **Supercategories** or **Subcategories and Products** to open the classification categories list.
6. Click the checkmark next to the required category or product by clicking the **Reference Search** window to select one or more categories.
7. Click **Select**.

Creating Category Features

The Classification functionality of SAP Commerce allows you to assign attributes more dynamically than using the SAP Commerce's type system.

[Creating Category Features](#)

You can create new category features in Backoffice Administration Cockpit.

[Creating a Feature Descriptor](#)

You can create new feature descriptors for your category features in Backoffice Administration Cockpit.

Creating Category Features

You can create new category features in Backoffice Administration Cockpit.

Procedure

1. Log into Backoffice.
2. Go to **Catalog > Classification Systems > Classifying Categories**.
3. Select the classifying category in which you'd like to create a category feature from the search result list.
4. Go to the **Class Attributes** tab.
5. Click **Create New Category Feature**.
A pop-up appears.
6. Select a feature descriptor for the new category feature from the available choices in the **Feature Descriptor** menu.
If the required feature descriptor is not available, you can create it. For more information, see [Creating a Feature Descriptor](#).
7. Click **Next** until you reach the final screen, then click **Done**.
8. Click **Save** to save your new category feature.

Creating a Feature Descriptor

You can create new feature descriptors for your category features in Backoffice Administration Cockpit.

Procedure

1. Log in to Backoffice.
2. Go to **Catalog** **Classification Systems**.
3. Click **+** to open the **Create New Feature Descriptor** window.
4. Enter an identifier name for the classification feature into the **Identifier** field.
5. Select the classification system version which is to hold the classification feature from the **Classification System Version** menu.
6. Click **Done**.

You created a new feature descriptor.

Creating Classification Systems

In SAP Commerce, you can create classification systems, blank classification systems that you can use to create a classification hierarchy, and predefined classification systems.

Procedure

1. Log in to Backoffice Administration Cockpit.
2. Go to **Catalog** **Catalogs**.
3. Click the down arrow next to the **+** icon above the list of available catalogs.
4. Click the down arrow next to **Catalog** in the menu and select **Classification system** from the available options.
5. Enter the required details for your new classification system, and set the default status of the system using the radio buttons if required.
6. Click **Done**.

Results

When you refresh the list by clicking **Search**, your new classification system appears in the list.

Creating Classification System Versions

Creating a classification system version is only possible in the context of a catalog. A classification system version cannot exist outside a catalog version.

Procedure

1. Log into Backoffice.
2. Go to **Catalog** **Catalogs**.
3. Select the catalog in which you'd like to create a classification system version from the list of available catalogs and versions.
4. Got to the **Common** tab in the catalog details.

5. Click the arrow next to the **Create new Catalog version** field under **Versions of this Catalog**.
6. Click the arrow next to **Catalog version** to expand the options.
7. Select **Classification system version**.
8. Enter the required details in the **Create New Classification system version** window that appears.
9. Click **Done**.

The new classification system version is created, and applied to the catalog.

Creating Classifying Categories

You can easily create a new classifying category in Backoffice Administration Cockpit.

Procedure

1. Log in to Backoffice.
2. Go to **Catalog** **Classification Systems** **Classifying Categories** .
3. Click **+** to open the **Create New Classifying Category** window.
4. Enter an identifier name for the classification feature into the **Identifier** field.
5. Select the classification system version from the **Classification System Version** menu.
6. Click **Done**.

You created a new classifying category.

Classification Feature Value API

You can use SAP Commerce API to manage classification attributes and feature values defined by classification systems .

The central part of the feature value API is the **FeatureList** POJO. It encapsulates all feature values for one product according to its associated classification classes. This POJO holds the **Feature** instances, which encapsulate all values for one classification attribute. Finally the actual values are wrapped within the **FeatureValue** object, because classification attribute values may consist of multiple components like the actual value, a description and a unit. For all service operations like loading and storing features there is the dedicated **ClassificationService** service. You could also replace it with your own logic. For more details, read the *Customizing ClassificationService* section. Technically, all feature values are written as **ProductFeature** items.

i Note

Ordering a one-to-many relation is burdened with an SQL statement querying for an ordering attribute used to order the items of the **-many** side of the relation, for example **ProductFeature** of **Product2FeatureRelation**. For information how to disable such statements, see section *Tuning Ordered One-to-Many Relations* in [Relations](#).

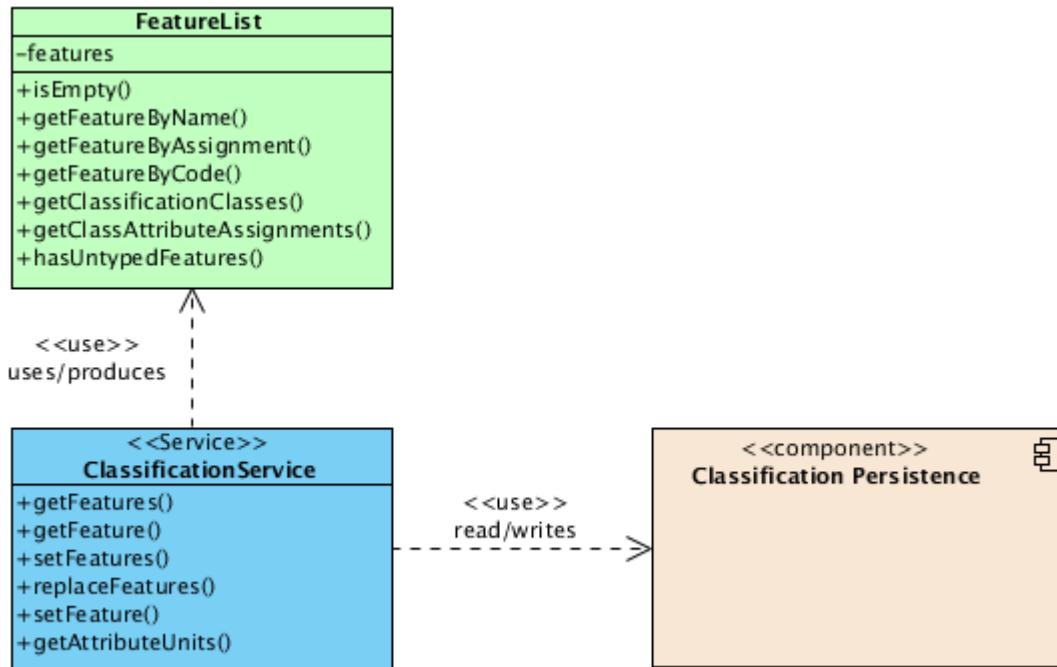
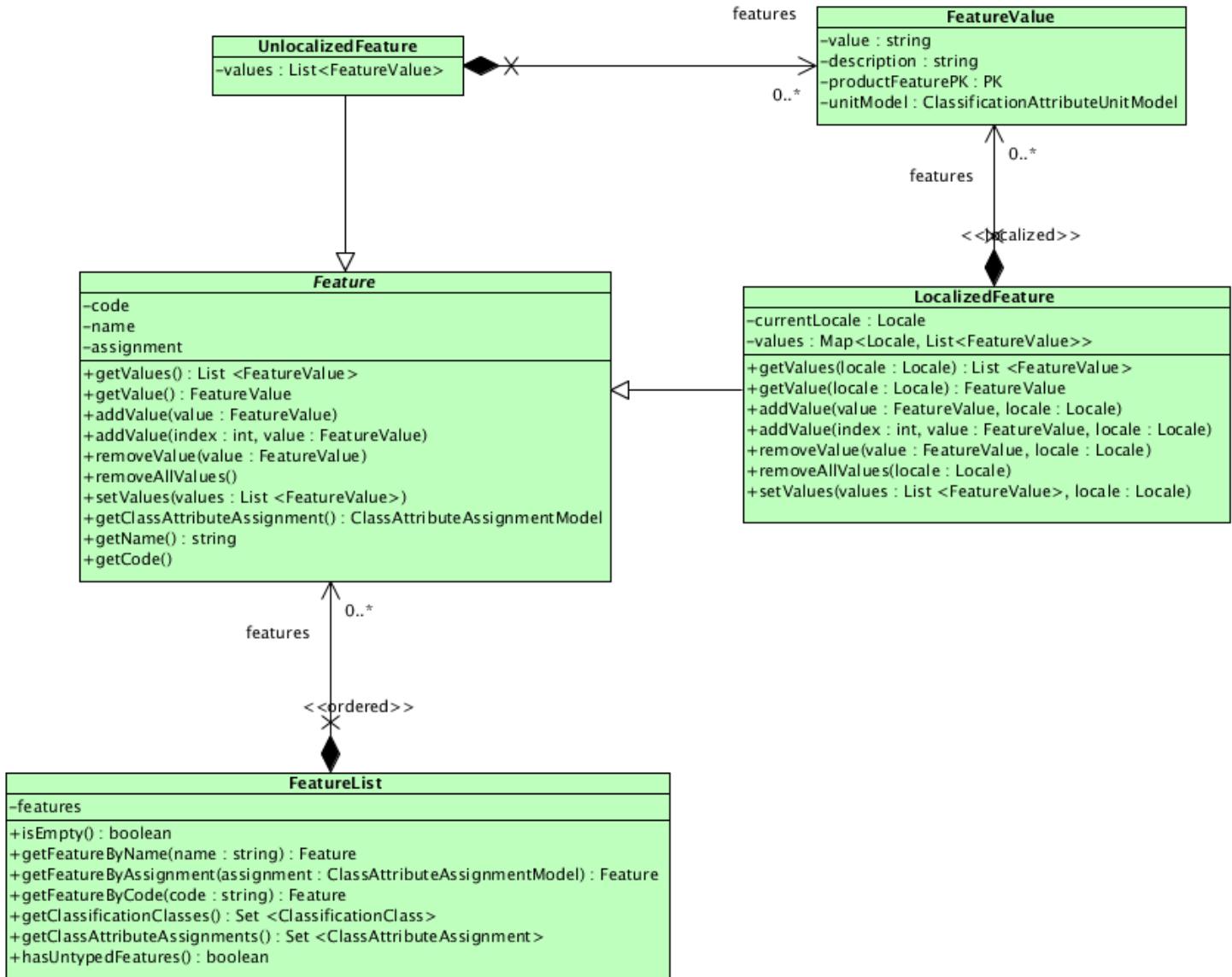


Figure: Above UML model shows general overview of classification feature values.

Figure: Above UML model illustrates a detailed architecture of the **FeatureList** and dependent objects. It is a part of API used to represent feature values that are initially read from the database.

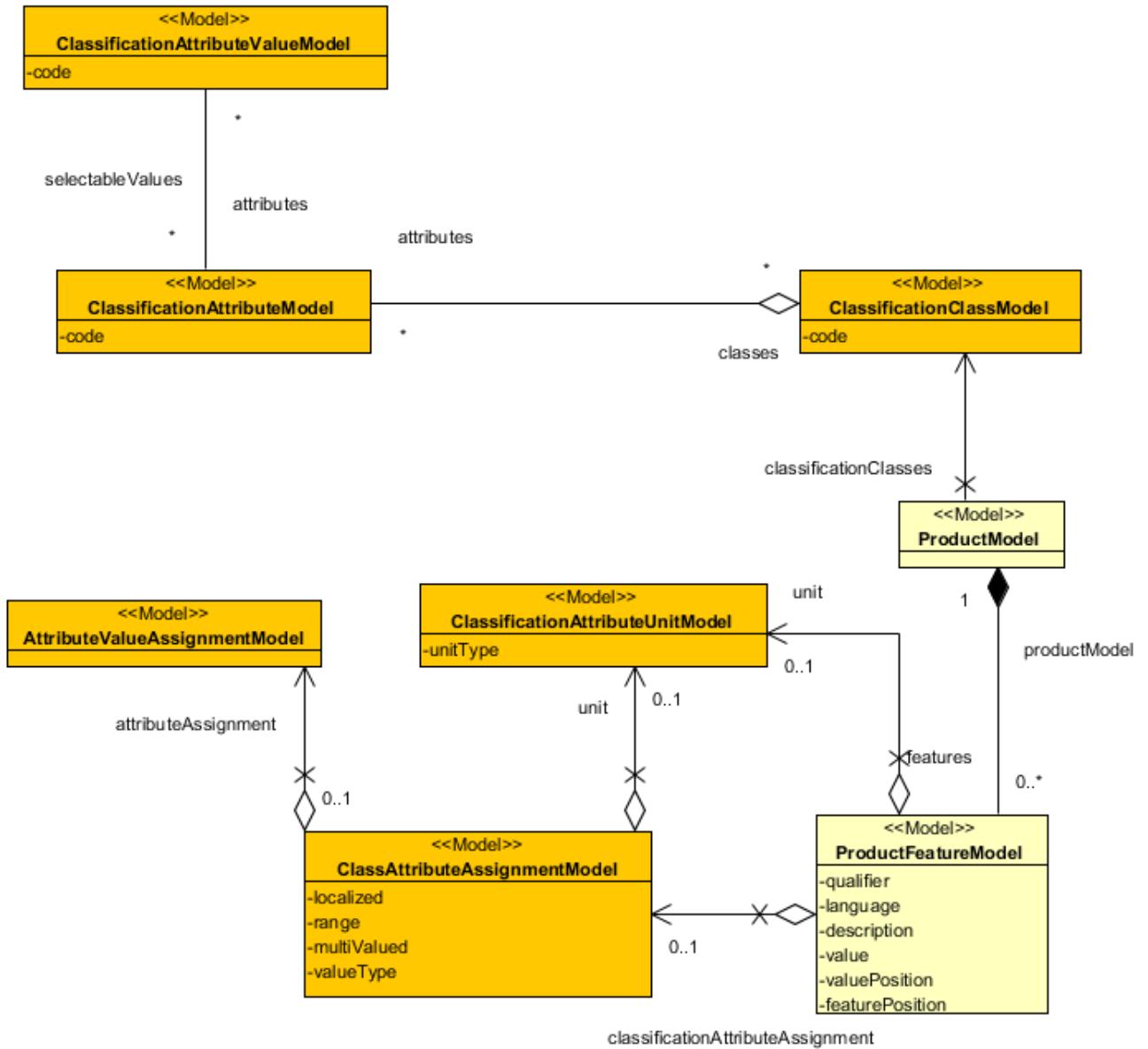


Figure: Above UML model illustrates a detailed architecture of the **Classification Persistence** component shown above in the first UML. It is an part of API used to represent how feature values are persisted in the database.

Accessing Classification Values

The most important difference between normal hybis Model handling and feature value handling is the fact that all modifications are collected within a **FeatureList** POJO before they all are persisted at once, so after each modification it is required to call appropriate method on **ClassificationService** to store **FeatureList** back to database.

Reading

You can read feature values from single `ProductModel` by using `getFeatures` method of `ClassificationService`. You can load all the feature values into `FeatureList` and then read single values from this object.

```
// classificationService is injected by Spring
final FeatureList featureList = classificationService.getFeatures(productModel);
// now get some value by code from FeatureList
final Feature feature = featureList.getFeatureByCode("SampleClassification/1.0/graphics.resolution");
// get FeatureValue from Feature
final FeatureValue featureValue = feature.getValue();
```

`FeatureValue` as object contains three main properties:

- `value`: It holds real value.
- `description`: Optional description of value.
- `unit`: Optional `ClassificationAttributeUnitModel` object.

Below is an example of how to read these properties:

```
final FeatureValue featureValue = feature.getValue();
final Object realValue = featureValue.getValue();
final String description = featureValue.getDescription();
final ClassificationAttributeUnitModel unit = featureValue.getUnit();
```

It is also possible to read value from `LocalizedFeature` like in below example:

```
// classificationService is injected by Spring
final FeatureList featureList = classificationService.getFeatures(productModel);
// now get some value by code from FeatureList
final Feature feature = featureList.getFeatureByCode("SampleClassification/1.0/graphics.resolution");
// get FeatureValue from Feature for specific Locale
if (feature instanceof LocalizedFeature) {
    Locale locale = new Locale("en");
    final FeatureValue featureValue = ((LocalizedFeature) feature).getValue(locale);
}
```

Adding

Once you have obtained a `FeatureList`, you can add new values to it like in below example:

```
// Assume you already have the following objects
ProductModel product;
// load all the features from product
final FeatureList featureList = classificationService.getFeatures(product);
// get feature from the list
final Feature feature = featureList.getFeatureByCode("SampleClassification/1.0/graphics.resolution");
// add value to feature
feature.addValue(new FeatureValue("1680x1050"));
// Store list of features
classificationService.setFeatures(product, featureList);
```

The `FeatureValue` can be instantiated with more than one parameter:

```
// Assume you already have the following objects
ProductModel product;
// load all the features from product
final FeatureList featureList = classificationService.getFeatures(product);
// get feature from the list
```

```

final Feature feature = featureList.getFeatureByCode("SampleClassification/1.0/foo.bar");

// With value and description
final FeatureValue featureValue1 = new FeatureValue("Redistribution and use in source and binary fo
// With value, description and unit (assume that you have already kg object)
ClassificationAttributeUnitModel kg;
final FeatureValue featureValue2 = new FeatureValue(Double.valueOf(777), "Weight of something", kg)

// add value to feature
feature.addValue(featureValue1);
feature.addValue(featureValue2);

// Store list of features
classificationService.setFeatures(product, featureList);

```

Modifying

Existing FeatureValue can be modified. Below example shows how to do it:

```

// Assume you already have the following objects
ProductModel product;

final FeatureList featureList = classificationService.getFeatures(product);
final Feature feature = featureList.getFeatureByCode("SampleClassification/1.0/graphics.resolution

// get first value from the list
final FeatureValue value = feature.getValue();

// change the value
value.setValue("2048x1280");

// store list
classificationService.setFeatures(product, featureList);

```

Removing

Existing FeatureValue can be removed. Below example shows how to completely remove all the values from Feature and then store its state to persistence layer:

```

// Assume you already have the following objects
ProductModel product;

final FeatureList featureList = classificationService.getFeatures(product);
final Feature feature = featureList.getFeatureByCode("SampleClassification/1.0/graphics.resolution

// remove all values
feature.removeAllValues();

// store list
classificationService.replaceFeatures(product, featureList);

```

It is also possible to remove single value from Feature object like in below example:

```

final Feature feature = featureList.getFeatureByCode("SampleClassification/1.0/graphics.resolution
final FeatureValue value = feature.getValue();
feature.removeValue(value);

// store list
classificationService.replaceFeatures(product, featureList);

```

Managing Localized Features

To store localized feature values use `LocalizedFeature` class. It provides methods for storing and reading localized values.

Below example shows how to add new value to locale:

```
// Read Features first
final FeatureList featureList = classificationService.getFeatures(productModel);
final Feature feature = featureList.getFeatureByCode("SampleClassification/1.0/graphics.resolution");

// Now add some new value to locale "en"
if (feature instanceof LocalizedFeature) {
    Locale localeEn = new Locale("en");
    final FeatureValue newValue = new FeatureValue(Double.valueOf(666.777), "really cool description");
    ((LocalizedFeature) feature).addValue(newValue, localeEn);
}
```

Customizing ClassificationService

`ClassificationService` service is dedicated for all service operations like loading and storing features. It internally uses the `ClassificationClassesResolverStrategy` for resolving class attribute assignments and classification classes and the `LoadStoreFeaturesStrategy` for loading and storing features into persistence layer. By the fact of using strategy pattern in `ClassificationService`, you are able to completely replace below strategies:

- `ClassificationClassesResolverStrategy`: It is responsible for resolving classification classes
- `LoadStoreFeaturesStrategy`: It is responsible for loading and storing product features

Search

General

First of all you have to keep in mind that contrary to normal SAP Commerce attributes classification attributes are designed for flexibility when it comes to adding or removing products to or from classes in addition to hold any amount of untyped feature values as well. Therefore the actual values are not stored inside the product's data table but all inside one table holding all feature values. So naturally any search upon classification attributes may be slower or resource consuming than searching upon normal attributes.

Using GenericSearch API

The easiest way of including classification attributes into a search query is using the `FeatureValueCondition` provided by the feature value API.

Getting Started

A simple search upon just one classification attribute using the static convenience methods of `FeatureValueCondition` looks like this:

```
ClassificationClass myClass;
ClassificationAttribute myAttribute;

GenericQuery q = new GenericQuery(
```

```

        ProductModel._TYPECODE,
        FeatureValueCondition.notNull(
            myClass.getAttributeAssignment(myAttribute)));
    )
);
List<ProductModel> product = genericSearchService.search(q).getResults();

```

Mixing with Normal Conditions

It is possible to mix attribute and classification attribute conditions.

```

ClassificationClass myClass;
ClassificationAttribute myAttribute;

GenericQuery q = new GenericQuery(ProductModel._TYPECODE);
q.addCondition(
    FeatureValueCondition.notNull(
        myClass.getAttributeAssignment(myAttribute)
    )
);
q.addCondition(
    GenericCondition.like( ProductModel.CODE, "%blah%" )
);
List<ProductModel> product = genericSearchService.search(q).getResults();

```

Searching Localized Features

When adding conditions for localized classification attributes it must be understood that only values of the current session language are searched. If required otherwise the condition also allows to specify the desired language directly as shown below.

```

ClassificationClass myClass;
ClassificationAttribute myLocAttr;

GenericQuery q = new GenericQuery(ProductModel._TYPECODE);

FeatureValueCondition locCond1 = FeatureValueCondition.contains(
    myClass.getAttributeAssignment(myLocAttr), "some text"
);
locCond1.setLanguage( C2LManager.getInstance().getLanguageByIsoCode("en") );

FeatureValueCondition locCond2 = FeatureValueCondition.contains(
    myClass.getAttributeAssignment(myLocAttr), "hallo"
);
locCond2.setLanguage( C2LManager.getInstance().getLanguageByIsoCode("de") );

// now we combine it within a OR condition
q.addCondition( GenericCondition.or( cond1, cond2 ) );

List<ProductModel> product = genericSearchService.search(q).getResults();

```

Automatic Number Conversion by Unit

One of the main advantage of using `FeatureValueCondition` instead of manual `FlexibleSearch` queries is the built-in unit conversion. This means that if the classification attribute actually defines unit and this unit is convertible into other units the search condition automatically converts values into one search unit. This way it is possible to store values described by different units and be able to compare them.

The following example assumes that you have a classification attribute `weight` containing the unit `kg` which is convertible into `g` and `mg`.

```

ClassificationAttributeUnit kg, mg, g;
ClassificationClass myClass;
ClassificationAttribute weight;

GenericQuery q = new GenericQuery(ProductModel._TYPECODE);
// without specifying the unit always the configured unit is taken ( here kg )
q.addCondition(
    FeatureValueCondition.less(
        myClass.getAttributeAssignment(weight), Double.valueOf( 2.5 )
    )
);
// we also may specify the unit to convert into ( now we search for > 300 mg )
FeatureValueCondition untCond = FeatureValueCondition.greater(
    myClass.getAttributeAssignment(weight), Double.valueOf( 300 )
);
untCond.setUnit( mg );
q.addCondition( untCond );

List<ProductModel> product = genericSearchService.search(q).getResults();

```

Comparison of Selection Values

Also there is the capability of comparing selectable values built-in, which are in fact items and therefore stored as plain PK reference. The condition object generates a query which maps each selectable value to its position with the selectable values list which is configured for the searched classification attribute.

Having a classification attribute `sensorClass` which offers the selectable values [**1 megapixel**, **2 megapixel**, **3 megapixel**, **4 megapixel...**] the user can easily use the `less` or `greater` condition to find better or worse matching products.

```

ClassificationClass digiCam;
ClassificationAttribute sensorClass;
ClassificationAttributeValue 1mp, 2mp, 3mp, ...

GenericQuery q = new GenericQuery(
    ProductModel._TYPECODE,
    // > 2 megapixel ...
    FeatureValueCondition.greater( myClass.getAttributeAssignment(sensorClass), 2mp ),
    // ... and < 5 megapixel
    FeatureValueCondition.less( myClass.getAttributeAssignment(sensorClass), 4mp )
);

List<ProductModel> product = genericSearchService.search(q).getResults();

```

Ordering by Classification Attributes

When searching products including classification attributes it may also be required to sort the result by these attributes. The feature value API provides the `FeatureValueOrderBy` class to meet this requirement.

As described before `number values` are converted into one unit as well as `selectable values` turned into their position within the selectable values list at the queried attribute.

```

ClassificationClass myClass;
ClassificationAttribute myAttr;

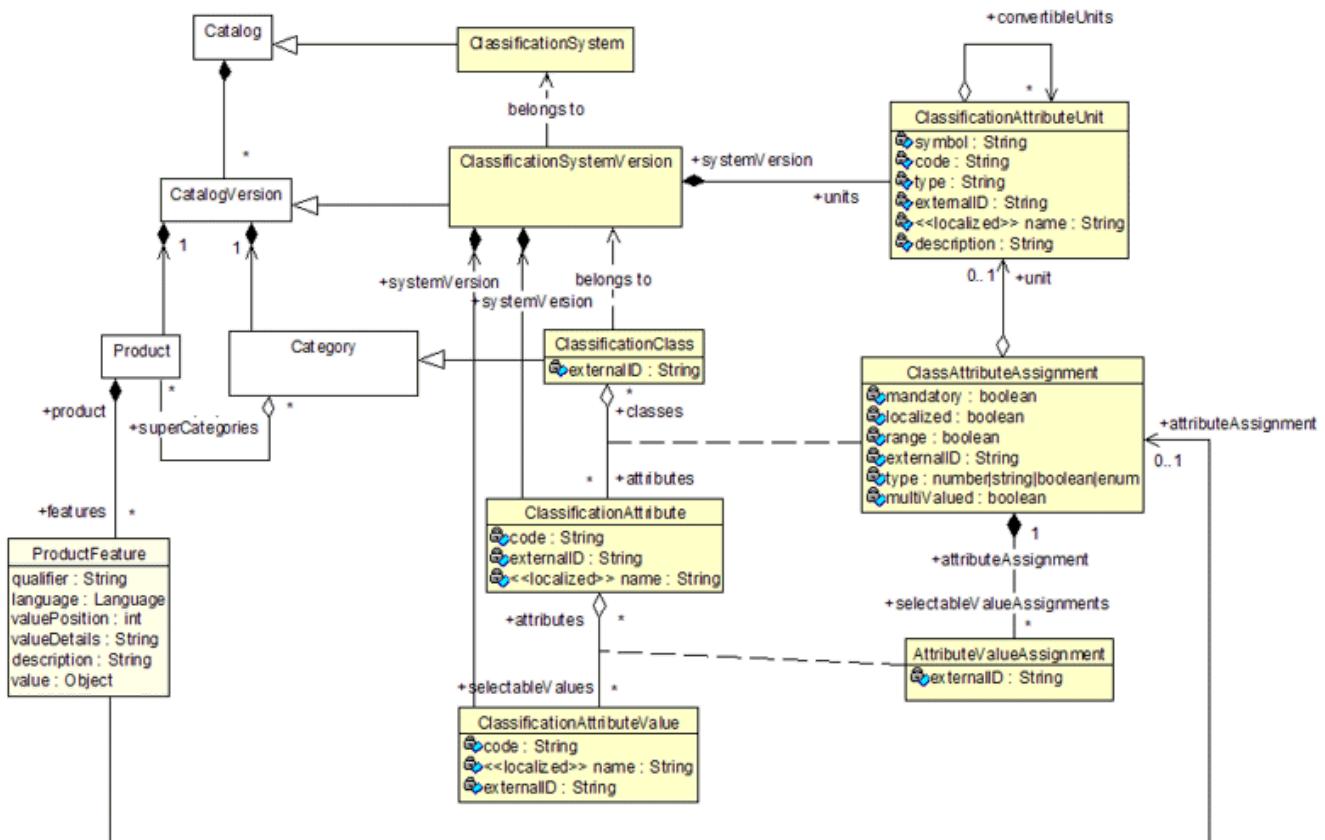
GenericQuery q = new GenericQuery( ProductModel._TYPECODE );
// now order by the class.attribute
q.addOrderBy(
    FeatureValueOrderBy.orderBy( myClass.getAttributeAssignment(myAttr), true /* asc */ )
);
List<ProductModel> product = genericSearchService.search(q).getResults();

```

Classification System API

As part of the SAP Commerce catalog engine, classification systems are defined as a specialized form of catalogs. Therefore most general catalog functionality may also be used for classification systems.

The following model shows the main classification API classes in addition to some of the basic catalog classes.



The whole system is contained within `ClassificationSystemVersion` items, which are grouped by `ClassificationSystem` items that make it possible to hold multiple versions separately (for example, `eClass` holds 4.1.80c and 5.1). The actual class is modeled as a `ClassificationClass` - a subtype of `Category` - mostly for adding convenience method to the API. Classification attributes (or features; these terms are used equally) are provided by the `ClassificationAttribute` class and selectable attribute values by `ClassificationAttributeValue`. Finally, each system may also hold its own set of units (see `ClassificationAttributeUnit`), which must not be confused with the normal product packaging units defined by the code `Unit` class.

Model Details

Classifying Products

There are several ways to classify products:

- Assigning Directly: Because classification classes are categories, you can assign products to them directly. In this method, the product refers to its classes within its `superCategories` attribute. Even though this is easy to understand, it also requires the greatest assignment effort compared to the sharing-the-class method.
 - Sharing the Class: As most catalogs show, all products of one leaf category usually share the same class. Therefore, you are also permitted to assign this normal category to the classification class. In this method, you are only required to put products into their navigation categories and the products then inherit the classification class from them. This method is

recommended over directly assigning products to classes because you only have to create a few links and products are classified automatically by adding and/or removing them to/from navigation categories.

You can assign a product to multiple classification classes even if it belongs to different systems, system versions, or is from the same version.

Classes can have other classes as subclasses which is a general category - subcategory behaviour. In this way, products are classified by their associated classes and their super classes.

Defining Attributes

Each classification system (-version) contains its own set of unique attributes that can be assigned to any of its own classes.

i Note

This behavior differs to the platform type system attribute definition where one `AttributeDescriptor` belongs to one `ComposedType` exclusively.

One class can have multiple attributes assigned to it, but each attribute can be assigned only once. A classification class also inherits all attributes assigned to super classes, except in cases where it has its own assignment for a specific attribute. In this way, the classified products of one class can hold the values for all of these attributes.

The API handles class-attribute associations with `ClassAttributeAssignment` items--as seen in the model above. It carries the most essential information about the attribute at this class as follows:

- `field_type`: Text, number, boolean, and selection are available.
- `unit`: Defines the unit to measure the actual (number) values of the attribute.
- `localized`: Whether or not the attribute values are language dependent.
- `mandatory`: Whether or not one product is required to hold at least one value.
- `multiValued`: Whether or not multiple values may be entered (or selected).
- `range`: Special case of multiValued, up to 2 values can be stored.

It also allows you to define a set of selectable values in addition to the default set of selectable values provided by the attribute; see `selectableValueAssignments` in the model. The default selectable values are not shown.

Selectable Values

The `ClassificationAttributeValue` is one of the new items owned by a classification system. It allows you to define selectable values (or enumeration values--may be used synonymously) to be associated with attributes and selected as the product's attribute value. There are two ways of defining selectable values for one attribute:

- Default value list: List of values is stored right at one attribute regardless of the classes it is assigned to and is used as fallback for the class-attribute assignment value list option.
- Class-attribute assignment value list: List of values is stored for one class-attribute assignment only (for example, `Car.color` has a different set than `Hair.color`)

The position of one value, within the value list that contains it, defines its comparison value. As a result, it can be easily used for querying or ordering product by.

Units

In addition to the `Unit` core class, which is intended to define packaging units, the classification system also holds its own set of `ClassificationAttributeUnit` items. Though they offer almost the same functionality as the `unit` does, a new type has been created for them to avoid mix-ups between the global set of packaging units and multiple system dependent SI units provided by different classification standards. The classification units allow you to measure values and convert values on-the-fly for searching or displaying.

The `unitType` attribute provides conversion in cooperation with the `conversionFactor` attribute. It works as follows:

- All units of the same type, for example: weight, volume, etc., are considered to be convertible.
- Conversion is done by multiplying and dividing using the conversion factors of source and target unit.

API Usage

Systems and Version

- Creation of classification systems is provided by the `CatalogManager`. A catalog language has to be given for the classification:

```
Language de;

// creation
ClassificationSystem system = CatalogManager.getInstance().createClassificationSystem("Hy0815
ClassificationSystemVersion ver = system.createSystemVersion( "0.1.beta", de );
```

- The retrieval of the Classification System's features is provided by the `ClassificationSystemService`. One way to use the service, is to reference it in the `spring.xml`, for example:

```
<bean id="myService" class="MyService">
    <property name="classificationSystemService" ref="classificationSystemService"/>
</bean>
```

The methods from the `ClassificationSystemService` are now available for `MyService`:

```
//retrieve the ClassificationSystemModel
ClassificationSystemModel other = classificationSystemService.getSystemForId("other");
Collection<ClassificationSystemVersionModel> versions = other.getCatalogVersions();
```

The classification system ID must be globally unique and the version must be unique at least within its enclosing system.

Classification Classes

Classes are handled primarily through the `ClassificationSystemVersion` class.

The retrieval of classification classes is provided by the `ClassificationSystemService`:

```
//retrieve the ClassificationSystemVersionModel
ClassificationSystemVersionModel version = classificationSystemService.getSystemVersion("systemCode");
//retrieve all root ClassificationSystemVersionModels
Collection<ClassificationClassModel> rootClasses = classificationSystemService.getRootClassesForSystem("systemCode");
//retrieve the ClassificationClassModel
ClassificationClassModel classificationClass = classificationSystemService.getClassForCode(version,
```

The relations between the classes can be queried as shown in the following examples:

- The subclass and super class relations between classes can be queried in the following way:

```

ClassificationSystemVersion ver;

ClassificationClass root = ver.getClassificationClass("root");

Collection<ClassificationClass> subClasses = root.getSubClasses();

// which is not necessarily the same as:
Collection<Category> subCategories = root.getSubCategories();
// since classes may be assigned to normal categories as well (see above)

```

- The retrieval of the classification class is handle by the ClassificationSystemService:

```

ClassificationSystemVersionModel classificationSystemVersion= classificationSystemService.get

// retrieve classification class for a given code
ClassificationClassModel classificationClass = classificationSystemService.getClassForCode(
    classificationSystemVersion, "classificationClassCode");

```

i Note

The `getSubClasses` and `getSubCategories` methods have not been migrated to the ServiceLayer. Therefore, there is no equivalent service implemented.

The assignment of attributes to classification classes is aided by a range of methods. Some of these are presented in the following code sample.

```

ClassificationSystemVersion ver;

ClassificationClass bag = ver.getClassificationClass("bag");

// how to assign
ClassAttributeAssignment bag_weight = bag.assignAttribute(
    ver.getClassificationAttribute("weight"), // attribute
    CatalogConstants.Enumerations.ClassificationAttributeTypeEnum.NUMBER, // value type
    ver.getAttributeUnit("weight"), // unit
    Collections.EMPTY_LIST, // selectable values
    0 // attribute position since they're ordered
);

// how to fetch all ( inherited or own )
Collection<ClassificationAttribute> all = bag.getClassificationAttributes();
Collection<ClassificationAttribute> allOwn = bag.getDeclaredClassificationAttributes();

// how fetch all assignments directly ( inherited or own )
Collection<ClassAttributeAssignment> all = bag.getClassificationAttributeAssignments();
Collection<ClassAttributeAssignment> allOwn = bag.getDeclaredClassificationAttributeAssignments();

// single attribute access ( finds inherited too )
ClassificationAttribute fabric = bag.getClassificationAttribute( "fabric" );
ClassAttributeAssignment bag_fabric = bag.getAttributeAssignment( fabric );

```

You may retrieve a classification class for a given code using the ClassificationSystemService. Services to assign and fetch attributes are not migrated yet:

```

ClassificationSystemVersionModel classificationSystemVersion = classificationSystemService.getSystem

// retrieve classification class for a given code

```

```
ClassificationClassModel classificationClass = classificationSystemService.getClassForCode(
    classificationSystemVersion, "bag");
```

Even though the cleanest way of managing attribute settings is using ClassAttributeAssignment instances directly, the class offers a range of convenience methods as well.

```
ClassificationSystemVersion ver;
// another way of getting the attribute
ClassificationAttribute weight = ver.getClassificationAttribute("weight");
ClassificationClass bag = ver.getClassificationClass("bag");

// changing the value type
bag.setAttributeType(
    weight,
    EnumerationManager.getInstance().getEnumValue(
        CatalogConstants.TC.CLASSIFICATIONATTRIBUTETYPEENUM,
        CatalogConstants.Enumerations.ClassificationAttributeTypeEnum.STRING
    )
);
// instead of
bag.getAttributeAssignment(weight).setAttributeType(
    EnumerationManager.getInstance().getEnumValue(
        CatalogConstants.TC.CLASSIFICATIONATTRIBUTETYPEENUM,
        CatalogConstants.Enumerations.ClassificationAttributeTypeEnum.STRING
    )
);
```

You can get the attributes in the following way:

```
ClassificationSystemVersionModel version = classificationSystemService.getSystemVersion("systemCode");
//retrieve the ClassificationAttributeModel
ClassificationAttributeModel classificationAttribute = classificationSystemService.getAttributeFor(
//retrieve the ClassificationClassModel
ClassificationClassModel classificationClass = classificationSystemService.getClassForCode(version,
```

i Note

Currently a service to set attribute type is not migrated yet.

Be careful when getting and setting selectable values. Here, the ClassificationClass performs the fallback action described above, instead of simply delegating the value list getter to the ClassAttributeAssignment.

```
ClassificationSystemVersion ver;

ClassificationClass car = ver.getClassificationClass("car");
ClassificationAttribute color = ver.getClassificationAttribute("color");
ClassificationAttributeValue green = ver.getAttributeValue("green"),
    red = ver.getAttributeValue("red"),
    yellow = ver.getAttributeValue("yellow"),
    black = ver.getAttributeValue("black");

// getting selectable values including fallback to default value list
List<ClassificationAttributeValue> values = car.getAttributeValues( color );

// which must not be the same as ...
List<ClassificationAttributeValue> values = car.getAttributeAssignment(color).getAttributeValues();
// ... since it read the assignment's value list only !

// on the other hand ...
car.setAttributeValues(
    color,
    Arrays.asList( red, green yellow )
);
// ... really does the same as ...
car.getAttributeAssignment(color).setAttributeValues(
    Arrays.asList( red, green yellow )
```

```
);
// ... because the default value list changed as follows !
color.setDefaultAttributeValues( green, black );
```

You can use the following methods from the ClassificationSystemService to retrieve data:

```
ClassificationSystemVersionModel version = classificationSystemService.getSystemVersion("systemCode");
//retrieve the ClassificationClassModel
ClassificationClassModel car = classificationSystemService.getClassForCode(version, "car");
//retrieve the ClassificationAttributeModel
ClassificationAttributeModel color = classificationSystemService.getAttributeForCode(version, "color");
//retrieve the different ClassificationAttributeValueModels
ClassificationAttributeValueModel green = classificationSystemService.getAttributeValueForCode(version, "green");
ClassificationAttributeValueModel red = classificationSystemService.getAttributeValueForCode(version, "red");
ClassificationAttributeValueModel yellow = classificationSystemService.getAttributeValueForCode(version, "yellow");
ClassificationAttributeValueModel black = classificationSystemService.getAttributeValueForCode(version, "black");
```

i Note

The getAttributeValues is not yet implemented.

Attributes, Units, and Selectable Values

Attributes, units, and selectable values are usually created and held independently within the classification system. However, their normal use is being assigned to classes or class attributes. Attributes, units, and selectable values have to be unique--in regard to their code--within the system.

```
ClassificationSystemVersion ver;
// create a new attribute
ClassificationAttribute weight = ver.createClassificationAttribute("weight");

// create new units
ClassificationAttributeUnit weight_t = ver.createAttributeUnit( "t", "t", "weight", 1.0 );
ClassificationAttributeUnit weight_kg = ver.createAttributeUnit( "kg", "kg", "weight", 1000.0 );
ClassificationAttributeUnit weight_g = ver.createAttributeUnit( "g", "g", "weight", 1000000.0 );
// ... which are indeed convertible ...
Set<ClassificationAttributeUnit> convertible = weight_g.getConvertibleUnits();
// ( should return 'weight_t' and 'weight_kg' )

// create new selectable values
ClassificationAttributeValue form_cube = ver.createClassAttributeValue("form_cube" );
ClassificationAttributeValue form_cylindric = ver.createClassAttributeValue("form_cylindric" );

// fetching is straight forward using the version
ClassificationAttribute weight = ver.getClassificationAttribute("weight");
ClassificationAttributeUnit weight_t = ver.getAttributeUnit( "t" );
ClassificationAttributeValue form_cube = ver.getAttributeValue("form_cube" );
```

You can use the following methods from the ClassificationSystemService to retrieve data:

```
ClassificationSystemVersionModel version = classificationSystemService.getSystemVersion("systemCode");
//retrieve the ClassificationAttributeModel
ClassificationAttributeModel classificationAttribute = classificationSystemService.getAttributeForCode(version, "classificationAttribute");
//retrieve the ClassificationAttributeUnitModel
ClassificationAttributeUnitModel classificationAttributeUnit = classificationSystemService.getAttributeUnitModel(classificationAttribute);
//retrieve the ClassificationAttributeValueModel
ClassificationAttributeValueModel classificationAttributeValue = classificationSystemService.getAttributeValueModel(classificationAttribute);
//assume that 'weight_g', 'weight_t' and 'weight_kg' are available
Collection<ClassificationAttributeUnitModel> units = classificationSystemService.getConvertibleUnits(classificationAttribute);
// ( should return 'weight_t' and 'weight_kg' )
// retrieves all units of the specific _type_ (e.g. 'weight' -> kg, g, and mg )
Collection<ClassificationAttributeUnitModel> units = classificationSystemService.getUnitsOfTypeForSystem("weight");
```

```
// retrieves all available unit types within the system version(e.g. 'time', 'weight' , 'length', ...
Collection<String> units = classificationSystemService.getUnitTypesForSystemVersion(version);
```

i Note

The createAttributeUnit is not yet implemented

Product Assignment Directly and Via Category

As described above products may be assigned directly or inherit their classification class from their normal categories.

```
//slayer classification refactoring necessary
// class from one system
ClassificationSystemVersion sys1Ver;
ClassificationClass bags = sys1Ver.getClassificationClass("bags");
// ... and another
ClassificationSystemVersion sys2Ver;
ClassificationClass specials = sys1Ver.getClassificationClass("specialOffer");

// here goes the normal product catalog
//slayer
CatalogVersionModel version = ...;
String categoryCode = ...;
String productCode = ...;
CategoryService categoryService = ...;
ProductService productService = ...;
CategoryModel category = categoryService.getCategoryForCode(version, categoryCode);
ProductModel product = productService.getProductForCode(version, productCode);

// a) Assign directly
//slayer
ModelService modelService = ...;
List<ProductModel> products = new ArrayList<ProductModel>(category.getProducts());
products.add(product);
category.setProducts(products);
modelService.save(category);
List<CategoryModel> categories = new ArrayList<ProductModel>(product.getSupercategories());
categories.add(category);
product.setSupercategories(categories);
modelService.save(product);

// b) Do not assign directly, but assign category instead
//slayer
bag = ...;
List<CategoryModel> superCategories = new ArrayList<CategoryModel>(category.getSupercategories());
superCategories.add(bag);
category.setSupercategories(superCategories);
modelService.save(category);
```

To get the associated classes for a given product or category, use the CategoryService or the ProductService.

```
//slayer based code
CatalogVersionModel version = ...;
String categoryCode = ...;
String productCode = ...;
CategoryService categoryService = ...;
ProductService productService = ...;
CategoryModel category = categoryService.getCategoryForCode(version, categoryCode);
ProductModel product = productService.getProductForCode(version, productCode);

//slayer classification refactoring necessary
// the most exact way is getting classes from the product ...
Collection<ClassificationClass> cClasses =
    CatalogManager.getInstance().getClassificationClasses(p0815);
// ... nevertheless if only categories are assigned we may save time doing this ...
```

```
Collection<ClassificationClass> cClasses =
    Catalogmanager.getInstance().getClassificationClasses(cat0815);
```

SameSite Cookie Attribute Handler

SAP Commerce includes a cookie handler that you can use to configure the SameSite attribute of the Tomcat cookie processor to handle cookies differently depending on their origin.

The cookie handler works for cookies added only through standard Servlet API. It doesn't support headers that are assembled as raw values and sent as String values.

To use this feature, make sure that any Accelerator templates that you're using are generated from SAP Commerce version 1905 and later.

Configuration

You can enable or disable the SameSite cookie handler with the `cookies.SameSite.enabled` property in the `local.properties` file. The default value of `cookies.SameSite.enabled` is `false`, but if you install the `samlssologin` extension, this value changes to `true` automatically.

Attribute Properties

The handler defines the value of the SameSite cookie attribute using the following four values, listed from most to least important:

1. `cookies.<domain>.<path>.<name>.SameSite`
2. `cookies.<domain>.<path>.SameSite`
3. `cookies.<domain>.SameSite`
4. `cookies.SameSite`

The handler first looks for a property that respects all the cookie attributes (domain, path, and name). If the handler doesn't find it, it looks for a property that considers the domain and path attributes. If such a property doesn't exist, the handler considers a property that respects the domain attribute. If the handler doesn't find it, it uses a property with the global default value (`cookies.SameSite`).

The property uses the following values:

- Unset
- None
- Lax
- Strict

For definitions of these values, see [SameSite cookies](#).

i Note

If you want to set the value of `cookies.SameSite` to `None`, `Secure` must be enabled as the following:

```
cookies.SameSite=None; Secure
```

Troubleshooting

If you notice that the handler has problems with finding a valid property name for a given cookie, set the `cookies.samesite` logger level to DEBUG to enable debug logging for the handler.

Scripting Engine

Scripting Engine support allows you to execute logic written in scripting languages in run time without restarting the SAP Commerce Server. The scripting engine thus saves time and makes it possible to improve existing logic like cron job, task engine, and other similar tools.

Scripts may be stored in various repositories like a database (based on the `Script` model), classpath, file system, or even in a remote repository (for example Gists at GitHub) or can be executed on the fly as a snippet without being stored.

What Can I Use Scripts For?

These are just some example applications of scripts:

- [Scripts as Event Listeners](#)
- [Cronjob Scripting](#)
- [Task Scripting](#)
- [The SAP Commerce processengine](#)
- [ImpEx API](#)

Supported Languages

Scripting Engine implementation in Platform is based on the JSR-223 standard. At the moment, three languages are supported out-of-the-box:

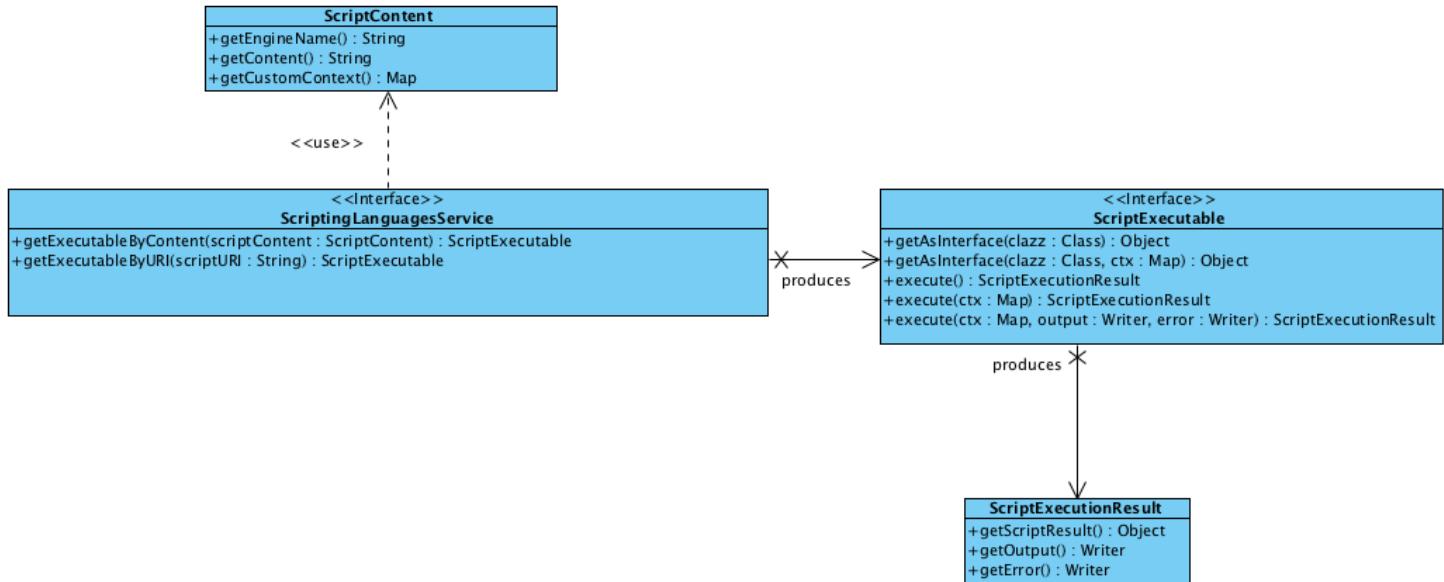
- Groovy
- BeanShell
- JavaScript

It is possible to add more languages from the JSR-223 support list, see [Adding New Languages](#).

API Overview

From the user perspective, the API is very simple. It is based on four main interfaces:

- `ScriptingLanguagesService` provides methods for obtaining a `ScriptExecutable` object.
- `ScriptExecutable` represents an executable object, which provides methods for executing it.
- `ScriptContent` represents an object that wraps script content in a specific language type.
- `ScriptExecutionResult` represents the script execution result.



Creating Scripts

There are three ways of creating scripts:

- programmatically
- using SAP Commerce Administration Console. For details, see [Administration Console](#)
- using Backoffice. For details, see section Managing Script Versions in Backoffice.

Let's create a Groovy script, which searches for all `Media` types that have the `mime` attribute empty, and sets the value of this attribute based on their `realFilename` attribute.

```

import de.hybris.platform.servicelayer.search.FlexibleSearchQuery
flexibleSearchService = spring.getBean "flexibleSearchService"
mimeService = spring.getBean "mimeService"
modelService = spring.getBean "modelService"
def findMediasWithoutMime() {
    query = new FlexibleSearchQuery("SELECT {PK} FROM {Media} WHERE {mime} IS NULL")
    flexibleSearchService.search(query).result
}
findMediasWithoutMime().each {
    it.mime = mimeService.getMimeFromFileExtension(it.realfilename)
    modelService.save it
}
  
```

i Note

Note how the interaction with the spring context is done. You can access any Spring bean from the SAP Commerce application context by referring to a special Spring global variable. This topic is covered later on in this document.

Executing Scripts

→ Tip

Scripts are run with the same rights that the current user has.

⚠ Caution

Scripts are not secured in any way, so you can use any language-specific structures; hence, calling `System.exit(-1)` is possible.

Now that you have this code sample, let's execute it directly by using the `ScriptingLanguagesService`. To do that, prepare a special wrapper for the real script content.

Executing Scripts from String - SimpleScriptContent

This implementation allows you to execute the script content directly as a String. Let's imagine that you have the previously mentioned script content in the String variable `content`:

```
final String content = ".... content of the script ...";
final String engineName = "groovy";

// Let's assume we have scriptingLanguagesService injected by Spring
final ScriptContent scriptContent = new SimpleScriptContent(engineName, content);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent)

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

Executing Scripts from Resources - ResourceScriptContent

Writing scripts directly in the Java code and keeping them in variables is not the most convenient way to do it. It is much better to write them in your favorite editor with syntax highlighting. Script Engine allows you to execute scripts that are resource based, stored for instance on the classpath or directly in the file system.

Executing Scripts from the File System

Assuming that our sample script is located on the disk, with the path `/Users/zeus/scripts/setMimesForMedias.groovy`, you can execute it as follows:

```
import org.springframework.core.io.Resource;
import org.springframework.core.io.FileSystemResource;

final Resource resource = new FileSystemResource("/Users/zeus/scripts/setMimesForMedias.groovy");

// Let's assume we have scriptingLanguagesService injected by the Spring
final ScriptContent scriptContent = new ResourceScriptContent(resource);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent)

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

Executing Scripts by Using Classpath

Or, if you have the same script but in the classpath in the folder `scripts`:

```
import org.springframework.core.io.Resource;
import org.springframework.core.io.ClassPathResource;

final Resource resource = new ClassPathResource("scripts/setMimesForMedias.groovy");

// Let's assume we have scriptingLanguagesService injected by the Spring
```

```

final ScriptContent scriptContent = new ResourceScriptContent(resource);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent)

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());

```

Executing Scripts Stored Remotely

You can also store the script content remotely, for instance as a gist at [github.com](https://gist.github.com/zeus/testMimesForMedias.groovy) under the URL <https://gist.githubusercontent.com/zeus/testMimesForMedias.groovy>. In this way, it is easy to get hold of and execute:

```

import org.springframework.core.io.Resource;
import org.springframework.core.io.UrlResource;

final Resource resource = new UrlResource("https://gist.githubusercontent.com/zeus/setMimesForMedia");

// Let's assume we have scriptingLanguagesService injected by the Spring
final ScriptContent scriptContent = new ResourceScriptContent(resource);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent)

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());

```

i Note

Keep in mind that `ResourceScriptContent` determines the script engine name by the proper file extension, thus all scripts stored in files must have valid extensions for the language in which they are written.

Handling Exceptions

The scripting engine implementation uses only unchecked exceptions. This is a list of all exceptions that you can encounter:

Exception	Description
<code>de.hybris.platform.scripting.engine.exception.ScriptingException</code>	Main exception class for all scripting engine related exceptions
<code>de.hybris.platform.scripting.engine.exception.ScriptExecutionException</code>	Exception thrown when the execution of a script has failed (for instance, a script contains errors in the body of the script)
<code>de.hybris.platform.scripting.engine.exception.ScriptCompilationException</code>	Exception thrown when the compilation of a script has failed.
<code>de.hybris.platform.scripting.engine.exception.ScriptNotFoundException</code>	Exception thrown when a persisted script cannot be found in the repository.
<code>de.hybris.platform.scripting.engine.exception.ScriptURISyntaxException</code>	Exception thrown when the Script URI contains an error.

Exception	Description
de.hybris.platform.scripting.engine.exception.DisabledScriptException	Exception thrown when a given ScriptExecutable is disabled due to previous errors.

Logging

To turn on logging for the scripting engine, set the logging level to debug by using the property `log4j.logger.de.hybris.platform.scripting.engine=DEBUG`.

Scripts Backed by Models - ModelScriptContent

SAP Commerce also comes with the model-based `ScriptContent` that is backed by the `Script` type. This special type is a container for storing scripts in the database. Let's now create a `Script` instance that keeps the media maintenance script:

```
import de.hybris.platform.scripting.enums.ScriptType;

// Let's assume we have modelService injected by the Spring
final ScriptModel script = modelService.create(ScriptModel.class);
script.setScriptType(ScriptType.GR00VY);
script.setContent(".... content of the script ...");
// code must be unique
script.setCode("setMimesForMedias");
modelService.save(script);

// now having our model we can wrap it using ModelScriptContent
final ScriptContent scriptContent = new ModelScriptContent(script);
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByContent(scriptContent)

// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

Autodisabling of Model-Based Scripts

Model-based scripts can be autodisabled. This comes in handy if a script throws an execution exception. In that case, the status of an autodisabling script changes to **disabled**. Otherwise, a faulty script would endlessly throw execution exceptions.

`ScriptModel` has two boolean properties: `autodisabling` and `disabled`. Both are by default set to `false`. To enable the auto-disabling feature, set `ScriptModel#autodisabling` item property to `true`. You can to set `Autodisabling` to `true` in Backoffice.

Script code	Script engine type	Active flag	PK
disableRefundDeliveryCostIfRefundedBefore	GROOVY	true	8796093120612
evalOrigRefundAmount	GROOVY	true	8796093087844
evalCustomRefundAmount	GROOVY	true	8796093055076

disableRefundDeliveryCostIfRefundedBefore

Script code: disableRefundDeliveryCostIfRefundedBefore	Version: 0	Content's checksum: 3758acf0d35fe7a576f232e8aa
Active flag: <input checked="" type="radio"/> True <input type="radio"/> False	Disabled	Autodisabling: <input type="radio"/> True <input checked="" type="radio"/> False <input type="radio"/> N/A

Scripts are marked as disabled on first throw of `de.hybris.platform.scripting.engine.exception.ScriptExecutionException`. Each next call to `ScriptExecutable#execute` on a script which was already marked as disabled throws a `de.hybris.platform.scripting.engine.exception.DisabledScriptException`

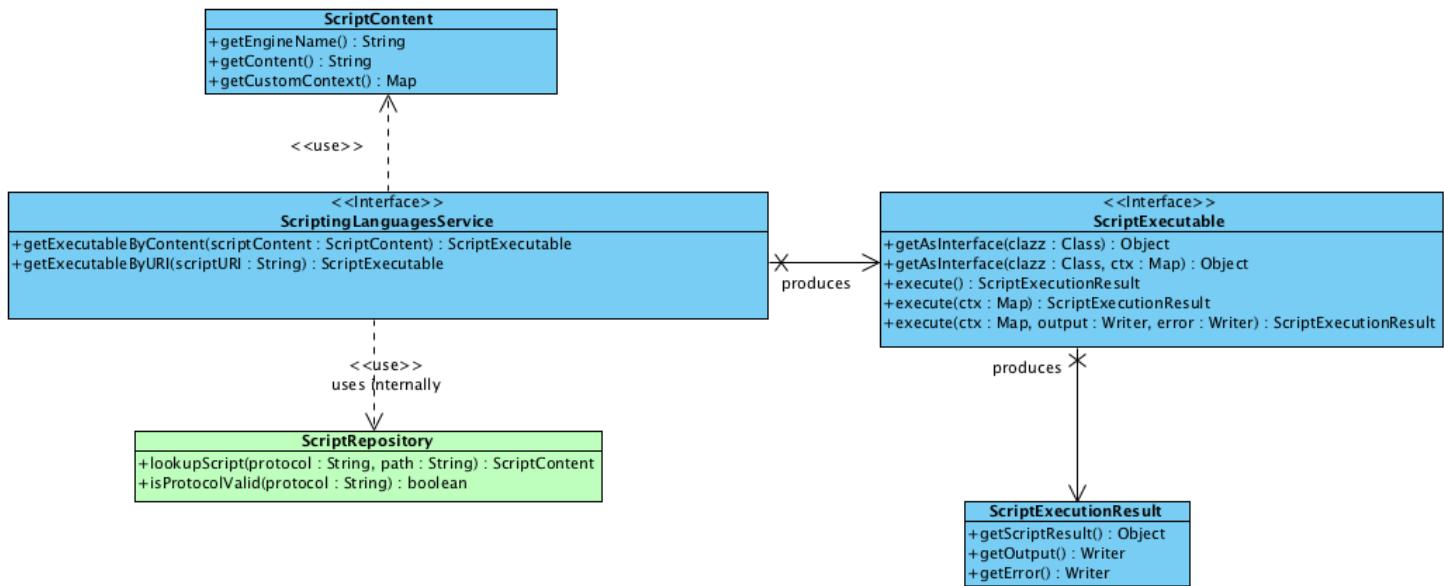
If you want to re-enable the script for execution, go to Backoffice, find the script and change the `disabled` flag to `false`.

Getting Script Executables by scriptURI

Playing with the `ScriptContent` interface for something persisted and whose exact address you know is not very handy. You always need to create a proper `ScriptContent` object. It is better to use the address of the script and get hold of the executable directly. Hence the concept of Script Repositories. A script repository allows you to look for a script in a particular storage - it may be a database, a local disk, or some remote storage. SAP Commerce comes with four implementations that are used internally by the `ScriptingLanguagesService` to resolve script addresses transparently. These are called `scriptURIs` and prepare appropriate `ScriptContent` objects behind the scenes to return a `ScriptExecutable` to the user.

- ModelScriptsRepository
- ClasspathScriptsRepository
- FileSystemScriptsRepository

- RemoteScriptsRepository



ModelScriptsRepository

This repository looks for scripts in the database. It supports the following type of `scriptURI`:

```
model://uniqueCodeOfScript
```

To get hold of a script using this repository, the user needs to use the `ScriptingLanguagesService` as follows:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("model://setMimesForScript")
// now we can execute script
final ScriptExecutionResult result = executable.execute();
// to obtain result of execution
System.out.println(result.getScriptResult());
```

ClasspathScriptsRepository

This repository looks for scripts in the database. It supports the following type of `scriptURI`

```
classpath://path/to/uniqueCodeOfScript.groovy
```

To get hold of a script using this repository, the user needs to use `ScriptingLanguagesService` as follows:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("classpath://script.groovy")
// now we can execute script
final ScriptExecutionResult result = executable.execute();
// to obtain result of execution
System.out.println(result.getScriptResult());
```

i Note

Keep in mind that scripts in this repository must contain a valid file extension according to the language they are written in.

FileSystemScriptsRepository

This repository looks for scripts in the database. It supports the following type of scriptURI

```
file:///absolute/path/to/uniqueCodeOfScript.groovy
file:///c:/absolute/path/to/uniqueCodeOfScript.groovy
```

To get hold of a script using this repository, use the `ScriptingLanguagesService` as follows:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("file:///Users/zei
// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

i Note

Scripts in this repository must contain a valid file extension according to the language in which they are written.

RemoteScriptsRepository

This repository looks for scripts in the database. It supports the following type of scriptURI

```
http://server.com/path/to/uniqueCodeOfScript.groovy
https://server.com/path/to/uniqueCodeOfScript.groovy
ftp://server.com/path/to/uniqueCodeOfScript.groovy
```

To get hold of a script using this repository, use the `ScriptingLanguagesService` as follows:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("http://server.co
// now we can execute script
final ScriptExecutionResult result = executable.execute();

// to obtain result of execution
System.out.println(result.getScriptResult());
```

i Note

Scripts in this repository must contain a valid file extension according to the language in which they are written.

Executing a Script Using Arguments - `ScriptExecutable` and `ScriptExecutionResult`

So far you have seen a simple usage of `ScriptExecutable` that shows how to execute a script without any arguments. Let's now look at a more advanced scenario, where you want to fix all mime types in medias whose `realFilename` has a specific extension:

```
import de.hybris.platform.servicelayer.search.FlexibleSearchQuery
flexibleSearchService = spring.getBean "flexibleSearchService"
mimeService = spring.getBean "mimeService"
modelService = spring.getBean "modelService"

// the query finds all medias for which mime type is null and whose realfile nasme has a specific p
def findMediasWithoutMime() {
    query = new FlexibleSearchQuery("SELECT {PK} FROM {Media} WHERE {mime} IS NULL AND {realfil
    query.addQueryParameter("realfilename", realfilename);
    flexibleSearchService.search(query).result
```

```

}
findMediasWithoutMime().each {
    it.mime = mimeService.getMimeFromFileExtension(it.realpathname)

    modelService.save it
}

```

Now you can execute the script as follows (let's suppose the script is on the classpath) for all medias with a filename with the extension `xml`:

```

final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("classpath://script.groovy");
final Map<String, Object> params = new HashMap<>();
// here we pass arguments into the hashmap.
params.put("realfilename", "%xml");
// now we can execute script
final ScriptExecutionResult result = executable.execute(params);

// to obtain result of execution
System.out.println(result.getScriptResult());

```

The example above is written in Groovy. This means that the last line of the script always affects the result. If it returns something, you get a script result `Object`. If it does not, the call to the `ScriptExecutionResult#getScriptResult()` returns `null`. A script may also yield some output. Let's modify the script a little bit to print at the end the number of fixed mime types in Medias:

```

import de.hybris.platform.servicelayer.search.FlexibleSearchQuery

flexibleSearchService = spring.getBean "flexibleSearchService"
mimeService = spring.getBean "mimeService"
modelService = spring.getBean "modelService"

def findMediasWithoutMime() {
    query = new FlexibleSearchQuery("SELECT {PK} FROM {Media} WHERE {mime} IS NULL AND {realfilename} IS NOT NULL")
    query.addQueryParameter("realfilename", realfilename);
    flexibleSearchService.search(query).result
}

def mediaHit = 0
findMediasWithoutMime().each {
    it.mime = mimeService.getMimeFromFileExtension(it.realpathname)

    modelService.save it
    mediaHit++
}

println "Num of fixed mimetypes for Medias - ${mediaHit}"

```

Now the last line executes the `println` function, which means that it does not return anything but prints something to the standard output. Now you can read the message.

```

final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("classpath://script.groovy");
final Map<String, Object> params = new HashMap<>();
params.put("realfilename", "%xml");
// now we can execute script
final ScriptExecutionResult result = executable.execute(params);

// to obtain result of execution
System.out.println(result.getOutputWriter());// The message is: Num of fixed mimetypes for Medias - 1

```

i Note

Note that `ScriptExecutionResult` contains two methods for getting output and error writers - `getOutputWriter()` and `getErrorWriter()`. When calling methods on `ScriptExecutable` both, by default, return standard `StringWriter` objects. If you want to pass your own `Writer` implementation, you may call the method

```
ScriptExecutable#execute(Map<String, Object> context, Writer outputWriter, Writer errorWriter)
```

Using Returned Objects as Interfaces

It is also possible to get an object of a class from a script and call methods on it directly in the Java code. Let's rewrite our Medias-related script to something more object oriented:

```
class GroovyMimeFixer implements MimeFixer {
    final static FIND_ALL_QUERY = "SELECT {PK} FROM {Media} WHERE {mime} IS NOT NULL"
    final static FIND_FOR_EXT_QUERY = FIND_ALL_QUERY + " AND {realfilename} LIKE ?realfilename"

    def flexibleSearchService
    def mimeService
    def modelService
    def int fixAllMimes() {
        def query = new FlexibleSearchQuery(FIND_ALL_QUERY)
        def counter = 0
        flexibleSearchService.search(query).result.each {
            it.mime = mimeService.getMimeFromFileExtension(it.realfilename)
            modelService.save it
            counter++
        }
        counter
    }

    def int fixMimesForExtension(String extension) {
        def query = new FlexibleSearchQuery(FIND_FOR_EXT_QUERY)
        query.addQueryParameter("realfilename", "%.${extension}");
        def counter = 0
        flexibleSearchService.search(FIND_FOR_EXT_QUERY).result.each {
            it.mime = mimeService.getMimeFromFileExtension(it.realfilename)
            modelService.save it
            counter++
        }
        counter
    }
}

flexibleSearchService = spring.getBean "flexibleSearchService"
mimeService = spring.getBean "mimeService"
modelService = spring.getBean "modelService"

new GroovyMimeFixer(flexibleSearchService: flexibleSearchService, mimeService: mimeService, modelSe
```

i Note

Note that an instance of the class is returned in the last line of the script.

The script above defines a new Groovy class `GroovyMimeFixer` that implements the Java interface `MimeFixer`, which you have in the code base and which looks as follows:

```
public interface MimeFixer
{
    /**
     * Fix all empty mimes.
     *
     * @return num of fixed mimes
     */
    int fixAllMimes();

    /**
     * Fix mimes for particular file extension.
     *
     * @param extension
     * @return num of fixed mimes
     */
}
```

```
    */
    int fixMimesForExtension(String extension);
}
```

Now you can execute this script a bit differently to obtain the object returned by the script:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutableByURI("classpath://script.groovy");
final MimeFixer mimeFixer = executable.getAsInterface(MimeFixer.class);
mimeFixer.fixMimesForExtension("xml");
mimeFixer.fixAllMimes();
```

Accessing Spring Beans in Groovy

In all of the examples, the Spring application context was referred to using the global variable `spring`. This variable is available only at the top level of the script, thus you need to pass it into the classes and other structures that have a limited scope. You can also access spring beans directly by their bean names but remember that top level script scope also applies here. So the following example works:

```
import de.hybris.platform.scripting.model.ScriptModel
modelService.create(ScriptModel.class)
```

But this one does not:

```
import de.hybris.platform.scripting.model.ScriptModel
class ScriptModelCreator {
    def create() {
        modelService.create(ScriptModel.class)
    }
}
new ScriptModelCreator().create()
```

You must pass `ModelService` into the `ScriptModelCreator` class:

```
import de.hybris.platform.scripting.model.ScriptModel
class ScriptModelCreator {
    def modelService

    def create() {
        modelService.create(ScriptModel.class)
    }
}
new ScriptModelCreator(modelService: modelService).create()
```

Accessing Beans in Javascript

Javascript is much less strict, so you can access Spring beans directly:

```
FlexibleSearchQuery = Packages.de.hybris.platform.servicelayer.search.FlexibleSearchQuery;
var query = FlexibleSearchQuery("SELECT {PK} FROM {Media} WHERE {mime} IS NULL")
var found = flexibleSearchService.search(query).getResults()

for each (var media in found.toArray()) {
```

```

        media.setMimeType(mimeService.getMimeTypeFromFileName(media.getRealFileName()));
        modelService.save(media);
    }
}

```

Note how the import of Java `FlexibleSearchQuery` class is done - via a special `Packages` object.

Adding New Languages

Adding a new language is simple, but you need to know which libraries are required in the classpath. Here you'll see how to add Ruby. JRuby implementation has built-in JSR-223 support.

First, you need to define library dependencies in appropriate `external-dependencies.xml` file:

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>de.hybris.platform</groupId>
    <artifactId>scripting</artifactId>
    <version>5.0.0.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <dependencies>
        <dependency>
            <groupId>org.jruby</groupId>
            <artifactId>jruby-complete</artifactId>
            <version>1.7.11</version>
        </dependency>
    </dependencies>
</project>

```

Next, declare bean of type `de.hybris.platform.scripting.engine.internal.ScriptEngineType` in the Spring context as follows:

```

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

    <bean id="jrubyEngine" class="de.hybris.platform.scripting.engine.internal.impl.DefaultScriptEngineType" c:name="ruby" c:fileExtension="rb" c:mime="text/x-ruby" />
</beans>

```

i Note

Note the `c` namespace in Spring file definition (<http://www.springframework.org/schema/c>) - this is a handy shortcut for passing constructor arguments to the bean.

This bean generally represents a new type of the scripting engine and keeps its name, file extension, and mime.

Finally, add a new value to the existing `ScriptType` enumtype in the SAP Commerce type system. This one is required by the `Script` type mentioned above - if you need to store your scripts into a database. The value is the name of the scripting engine.

```

<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="items.xsd">
    <enumtypes>
        <enumtype code="ScriptType">
            <value code="GROOVY"/>
            <value code="BEANSHELL"/>
            <value code="JAVASCRIPT"/>
        </enumtype>
    </enumtypes>

```

```

<!-- new value comes here -->
<value code="RUBY"/>
</enumtype>
</enumtypes>
</items>

```

That's it - your new language is ready to use.

Caution

Methods of interface ScriptExecutable are not synchronised, so the underlying ScriptEngine must provide thread-safety. Before adding a new script language, test that the new language behaves as expected in a multithread environment.

Script Versioning

1) Let's create a simple print script with the code `testScript` in the Scripting Language Console in SAP Commerce Administration Console.

```
println "Groovy Rocks"
```

Save the script.

2) Now modify this script such that it prints:

```
println "Groovy Wows"
```

Save the modified script.

3) Modify the script again:

```
println "Groovy Amazes"
```

4) Now go to the **Browse** tab. You can see only one instance of the `testScript` in the directory tree.

However, if you query the database using the following query, you will see three versions of the `testScript`.

```
select * from scripts where p_code = 'testScript' order by p_version;
```

You can call revision 1 of this script in the following manner:

```
final ScriptExecutable executable = scriptingLanguagesService.getExecutablebyURI("model://testScript");
final ScriptExecutionResult result = executable.execute();
```

If you do not specify any revision number, the latest version will be used.

Managing Script Versions in Backoffice

You can create, edit, and search for your scripts in Backoffice.

Searching for existing scripts only shows the latest script version. To see all versions, change the `Active` flag attribute to `False`.

Related Information

[Scripts as Event Listeners](#)[Cronjob Scripting](#)[The SAP Commerce processengine](#)[Task Scripting](#)[scripting Extension](#)

Cronjob Scripting

SAP Commerce includes a built-in scripting API for using dynamic typed languages in the Platform at runtime. An example of its usage is for cronjob scripting.

Benefits of Using Dynamic Scripting to Create Cronjobs

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.

How It Works

Let's look at some basic concepts:

- **Script** - the item type where the script content is stored (a separate deployment table)
- **ScriptingJob** - a new ServicelayerJob item, which contains additionally the scriptURI (consequently, the stored script can be found at runtime from different locations (classpath, db, etc..))
- **ScriptingJobPerformable** - the spring bean assigned to every **ScriptingJob** instance; it implements the usual `perform()` method (like for any other cronjob). This is where the "scripted" cronjob logic is executed

You can execute the cronjob by java API (`CronjobService.perform(Cronjob)`) or by defining a trigger. The execution is delegated to the **ScriptingJobPerformable**.

Related Information

[Scripting Engine](#)[The Cronjob Service](#)

Using Scripts Stored in Database

See the example to learn how to use a script stored in the database.

Define a Script

The scripting language used in this example is Groovy. Use the impex import console to import a script that will print out the 'hello world' text with the current date.

```
INSERT_UPDATE Script; code[unique=true];content
;myGroovyScript;println 'hello groovy! '+ new Date()
```

This stores your script in the database.

Define a ScriptingJob

Use the impex import console to import the following:

```
INSERT_UPDATE ScriptingJob; code[unique=true];scriptURI
;mydynamicJob;model://myGroovyScript
```

This stores your **ScriptingJob** (aptly named **mydynamicJob**) in the database. Regarding the scriptURI, if you want to point to a script stored in database, you can use the convention: **model://myscriptreference**

Define a CronJob

Use the impex import console to import the following:

```
INSERT_UPDATE CronJob; code[unique=true];job(code);singleExecutable;sessionLanguage(isocode)
;mydynamicCronJob;mydynamicJob;true;en
```

This stores the new cronjob holding the reference to our previously defined job.

Execute the Cronjob

Go to the scripting languages console in SAP Commerce Administration Console, make sure that you have the groovy language selected, enable the mode, and commit try the following:

```
def dynamicCJ = cronJobService.getChronicalJob("mydynamicCronJob")
cronJobService.performChronicalJob(dynamicCJ, true)
```

This executes your cronjob - just look at the console logs, where you should see the result - "hello groovy!", followed by the current date.

Optional Step - Use a Trigger

Use the impex import console to import the following:

```
INSERT_UPDATE Trigger;chronicalJob(code)[unique=true];cronExpression
;mydynamicCronJob; 0 * 16 * * ?
```

This executes the cronjob every minute between 16:00 and 17:00. Examine the logs to verify.

Cronjob Scripts Advanced Features

See some advanced features of cronjob scripts.

The previous example showed a simple one-line script that only printed out a message. However, it is possible to do much more, for example, you can influence the result, pass information to logs, and access the cronjob that is being executed.

Returning Cronjob Results

i Note

Returning a script result is optional.

`ScriptingJobPerformable` handles returning the `PerformResult()`, holding the status of the cronjob, which is the final result of the cronjob perform logic.

For example, returning a `String` 'My script is almost finished', will only print the message out, but the final `PerformResult` is set "outside".

The following example shows how to influence the `PerformResult`.

```
import de.hybris.platform.servicelayer.cronjob.PerformResult
import de.hybris.platform.cronjob.enums.CronJobStatus
import de.hybris.platform.cronjob.enums.CronJobResult
println 'hello groovy! ' + new Date()
new PerformResult(CronJobResult.UNKNOWN, CronJobStatus.PAUSED)
```

This sets the Cronjob status to Paused. Of course, it is possible to set any kind of `PerformResult`, for example depending on some conditions set earlier.

Cronjob Item as Context Parameter

It is possible to access the currently executed cronjob (as a `CronJobModel` instance). It is passed as a context parameter (`key="cronjob"`).

```
println 'hello groovy! ' + new Date()
println cronjob.code
println cronjob.status

//etc...
```

Logging Inside Cronjob Scripts

The cronjob API offers many logging possibilities, including storing logs in the database.

Proper logger is available inside scripts as context parameter (`key="log"`). You can add your own logs:

```
println 'hello groovy! ' + new Date()
log.info('my custom info log')
log.warn('my custom warn log')
log.error('my custom error log')

//etc...
```

Use the `logtodatabase` property to create a cronjob that will use a specific log level.

Logs outside the script (`ScriptingJobPerformable`), will also be considered in such case. Might be useful, if, for example, the script lookup fails. In this case, the script internal logs will not be visible, but the error log with `ScriptNotFoundException` will be stored.

Scripts as Event Listeners

SAP Commerce provides a scripting API for dynamic typed languages such as groovy or any JSR-223 compliant script engine. One of the advantages offered by this API is the ability to create script-based event listeners.

Benefits of Using Scripts to Create Event Listeners

Using the traditional SAP Commerce event system, the user is forced to rebuild the system and restart the server, because it is necessary to extend the `AbstractEventListener` class or, in some cases, to set up a Spring bean. For more information, see [Event System](#).

Using dynamic scripting it is all done at runtime.

How It Works

The `Script` item contains the script content. `ScriptingEventService` is a dedicated event service that allows registering and unregistering the dynamic event listeners by `scriptURI` at runtime. The `scriptURI` points to the stored script in a given location such as the classpath or database. Finally, the `ScriptListenerWrapper` wraps the dynamic listener, which is necessary to always get the same listener instance for a given `ScriptURI`.

Registering and Unregistering Listeners

To store the script with the custom event listener as content, use the existing scripting languages console or Backoffice. Bear in mind, though, that they are not registered dynamically - you need to do it yourself. To register the event listener, use the `scriptingEventService`. Based on a given `ScriptURI`, this service can find the script and return a proper listener instance to be registered.

i Note

Registering is **not persistent**. If you decide to restart the server at some point, dynamic scripts are not registered. However, the script items with the content are kept in the database. The user is responsible for registering scripting listeners again, if necessary.

Example

The following example uses a Groovy script that is stored in the database. Neither a server restart nor a Platform rebuild is necessary in any of the steps.

Prepare Your Script Content (EventListener)

Use the scripting languages console in SAP Commerce Administration Console to execute the following:

```
import de.hybris.platform.servicelayer.event.impl.AbstractEventListener
import de.hybris.platform.servicelayer.event.events.AbstractEvent
import de.hybris.platform.scripting.events.TestScriptingEvent
class MyScriptingEventListener extends AbstractEventListener<AbstractEvent>{

@Override
void onEvent(AbstractEvent event)
{
    if(event instanceof TestScriptingEvent){
        println 'hello groovy! '+ new Date();
    }
    else{
        println 'another event published '
        println event
    }
}
new MyScriptingEventListener();
```

Run the script to make sure that there are no syntax errors. If the script is fine, Groovy shows the `ScriptingEventListener` instance in the **Result** tab. This test script reacts to every event published in the system and prints out relevant logs. The `ScriptingEvent` is just a test event.

Store Your Script

Stay in the the scripting languages console and do not remove the previous content. Enter `myEventListenerScript` in the code input field and click **Save**. The script item gets persisted with the given code and content.

Register Your Listener

To register your listener, use the `ScriptingEventService` in the scripting languages console, as follows:

```
scriptingEventService.registerScriptingEventListener('model://myEventListenerScript')
```

The system returns `true` in the **Result** tab, which means that your listener is registered successfully. You can do an additional check by retrieving the list of the currently registered listeners:

```
eventService.eventListeners  
//or getting more readable output:  
//eventService.eventListeners.each{ println it}
```

Your listener is triggered by already-existing events. Examine the logs to verify that this is happening.

Create and Publish Your Custom Event

Assuming that there is a custom `ScriptingEvent` class, create an instance of that class and publish it.

```
import de.hybris.platform.scripting.events.TestScriptingEvent  
event = new TestScriptingEvent('myEvent')  
eventService.publishEvent(event);
```

This publishes your event and this way the scripted event listener also reacts to your custom event. As a result, you should see `hello groovy!`, followed by the current date printed in the logs.

Unregister Your Listener

```
scriptingEventService.unregisterScriptingEventListener('model://myEventListenerScript')
```

The listener is unregistered and stops reacting to the published events. You can verify it by publishing the previous event again.

Optionally Use Typed Events

Thanks to java generics, it is also possible to allow the listener to listen to only one type of events. The script can look like the following:

```
import de.hybris.platform.servicelayer.event.impl.AbstractEventListener  
import de.hybris.platform.servicelayer.event.events.AbstractEvent  
import de.hybris.platform.scripting.events.TestScriptingEvent  
class MyScriptingEventListener extends AbstractEventListener<TestScriptingEvent>{  
  
    @Override  
    void onEvent(TestScriptingEvent event)  
    {  
        println 'hello groovy! ' + new Date();  
    }  
}
```

```

}
}

new MyScriptingEventListener();

```

This way you can easily simplify the code by avoiding the `instanceof` checks, because the listener will only react to a specific type of event. Registering and publishing events works the same as in the previous steps.

Related Information

[Scripting Engine](#)

[Event System](#)

Task Scripting

You can use the SAP Commerce scripting API to easily and quickly create dynamic tasks based on scripts.

Benefits of Using Dynamic Scripting to Create Tasks

Using the conventional approach, implementing the `taskRunner` interface and setting up the runner bean forces you to rebuild the Platform and restart the server, see [The Task Service](#).

With dynamic scripting, there is no need to rebuild the Platform or restart the server, because we do not need to add and compile an implementation of the `taskRunner`. Using the new, all-in-runtime approach, you can go to the scripting console, create and store a script that is an implementation of the `taskRunner`, and execute the script; all this is done without ever having to rebuild the Platform.

How It Works

`Script` is the item type where the script content is stored. On the database level, it is a separate deployment table.

`ScriptingTask` is the new Task item, which additionally contains the `scriptURI`. Consequently, the stored script can be found at runtime from various locations (for example from classpath, database). `ScriptingTaskRunner` is the spring bean assigned to every `ScriptingTask` instance. It implements the `run()` and `handleError()` method, like for any other task. That is the place where the "scripted" task logic is executed.

All you need to do is create your own `taskRunner` in any supported dynamic typed language. The `ScriptingTaskRunner` then takes care of finding the script and delegating the execution to the `taskRunner` specific methods that the user needs to implement, that is `run()` and `handleError()` methods.

→ Tip

You can execute a task via java API using `taskService`.

Creating and Storing Scripts as Tasks in the Database

In this example, we use the Scripting console (see [Administration Console](#)) to create a Groovy script that is an implementation of `taskRunner`. Then we execute it. Finally, we show how to handle errors in scripts.

Prepare Script Content (TaskRunner)

Use the scripting languages console to execute the following:

```

import de.hybris.platform.task.TaskRunner;
import de.hybris.platform.task.TaskService;
import de.hybris.platform.task.TaskModel;
class MyScriptRunner implements TaskRunner<TaskModel>{

@Override
void run(TaskService taskService, TaskModel task)
{
    println 'hello groovy! ' + new Date();
}

@Override
void handleError(TaskService taskService, TaskModel task, Throwable error)
{
    println 'my groovy has errors! ' + new Date();
}
}
new MyScriptRunner();

```

This is the script content that is stored in the next step. Now execute the script to validate that there are no syntax errors. If the script is fine, groovy shows the Runner instance in the **Result** tab.

Store Your Script

Stay in the scripting languages console (don't remove the previous content), enter 'myGroovyScript' in code input field and click **save**. The script item should be persisted with given code and content.

Create and Execute Your Task

Stay in the the scripting languages console and remove the previous content (your script is already stored). Make sure you are executing the following in the commit mode:

```

import de.hybris.platform.task.model.ScriptingTaskModel;
import de.hybris.platform.task.TaskService;

task = modelService.create(ScriptingTaskModel.class);
task.scriptURI='model://myGroovyScript'

taskService.scheduleTask(task)

```

This runs your task; as a result you should see the '*hello groovy!*' text followed by the current date printed out in the logs.

i Note

The task is not persisted before scheduling by the `taskService`. It is enough to prepare it (via `modelService.create()`) and then the task engine handles the whole process - persisting the task, executing it, and removing from database. There were no errors during the execution of this task, so only the `run()` method was processed. Otherwise - it would be redirected to the `handleError()` method.

Try with Error Handling (Optional)

Prepare your script such that it throws an Exception in the `run()` method

```

import de.hybris.platform.task.TaskRunner;
import de.hybris.platform.task.TaskService;
import de.hybris.platform.task.TaskModel;
class MyScriptRunner implements TaskRunner<TaskModel>{

@Override

```

```

void run(TaskService taskService, TaskModel task)
{
    println 'hello groovy! ' + new Date();
    throw new NullPointerException("Error")
}

@Override
void handleError(TaskService taskService, TaskModel task, Throwable error)
{
    println 'my groovy has errors! ' + new Date();
}
}

new MyScriptRunner();

```

Save the script with a new name, for example as `MyErrorGroovyScript`. Finally, execute it, pointing to the error script `'model://myGroovyScript'`

Examine the logs. A `NullPointerException` should be thrown and finally '*my groovy has errors!*' followed by the current date should be printed out. This is because the logic has been redirected to the `handleError()` method

Related Information

[Scripting Engine](#)

[Administration Console](#)

Search Mechanisms

Both the `FlexibleSearchService` and the `GenericSearch API` allow developers to construct and execute queries against the SAP Commerce database, by focusing on SAP Commerce items rather than raw SQL. However, the way in which the queries are constructed is based on two completely different and complimentary approaches.

The topics covered include:

[FlexibleSearch](#)

SAP Commerce comes with a built-in query language of an SQL-based syntax, **FlexibleSearch**. FlexibleSearch enables searching over the items in SAP Commerce.

[GenericSearch](#)

While SAP FlexibleSearch offers a powerful search API to developers, some of them may prefer the GenericSearch API that is similar to Hibernate Criteria Queries. Hibernate is a collection of related projects, enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object and Relational Mapping.

FlexibleSearch

SAP Commerce comes with a built-in query language of an SQL-based syntax, **FlexibleSearch**. FlexibleSearch enables searching over the items in SAP Commerce.

FlexibleSearch is a powerful retrieval language built into SAP Commerce. It enables searching for SAP Commerce types and items using an SQL-based syntax. The execution of a FlexibleSearch statement takes place in two phases: pre-parsing into an SQL-compliant statement and running that statement on the database. During the pre-parsing phase, the FlexibleSearch framework resolves the FlexibleSearch syntax into SQL-compliant syntax. For example, the following two code snippets show a FlexibleSearch query and the statement that results from the FlexibleSearch query, which is executed on the database:

- FlexibleSearch query:

```
select {pk}, {code}, {name[de]} from {Product}
```

- SQL statement (executed on the database):

```
SELECT item_t0.PK , item_t0.Code , lp_t0.p_name
FROM products item_t0 JOIN productslp lp_t0 ON item_t0.PK = lp_t0.ITEMPK AND lp_t0.LA
WHERE (item_t0.TypePKString IN ( 23087380955301264 , 23087380955663520 , 23087380955
23087385363574432 , 23087380955568768 , 23087380955206016 ) )
```

FlexibleSearch abstracts the SAP Commerce type system from the actual database tables so that you can run searches on the type system level. Unlike on conventional SQL statements, in a FlexibleSearch query, you do not have to specify explicit database table names. The FlexibleSearch framework resolves type and database table dependencies automatically and specifies **UNIONS** and **JOINS** where necessary. The entire conversion process between type system and database representation takes place automatically. To access a type in a FlexibleSearch query, surround the type code with curly braces **{** and **}**, as in:

```
SELECT * FROM {Product}
```

i Note

FlexibleSearch queries are executed on the database directly using SQL statements. By consequence, it is not possible to run FlexibleSearch queries on jalo-only attributes as these attributes are not written into the database. Find more information in the [Jalo-only Attributes](#) document.

SAP Commerce executes FlexibleSearch queries in the context of a certain user account, using a session. As different user accounts have access to different items on SAP Commerce, the number of search results depends on the user account. The number of search results is defined by type access rights (these affect the Backoffice search results only), restrictions, catalog versions, and categories, for example. The more privileged a user account is, the more search results a FlexibleSearch yields in the context of that user account. By default, the user account assigned to a session is **anonymous**, so any FlexibleSearch query returns the search results matching the **anonymous** account by default. To run FlexibleSearch queries in the context of a user account different from **anonymous**, the session needs to be assigned to a different user account, such as:

```
import de.hybris.platform.servicelayer.user.UserService;
...
// Injected by Spring
userService.setCurrentUser(userService.getUserForUID("myUserAccount"));
...
```

Related Information

[The Type System](#)

Syntax Overview

See the overview of the FlexibleSearch syntax.

The basic syntax of a FlexibleSearch query looks like this:

```
SELECT <selects> FROM <types> ( WHERE <conditions> )? ( ORDER BY <order> )?
```

A FlexibleSearch query consists of:

- The mandatory **<selects>** parameter for the **SELECT** clause.

- The mandatory <types> parameter for the FROM clause.
- An optional <conditions> field for the WHERE clause.
- An optional ORDER BY clause.

SQL Command / Keyword	Description / Comment	Code Example
ORDER BY {alias:attribute}	Display results ordered by the value of attribute.	SELECT ... FROM ... ORDER BY...
ASC	Sort results in ascending order (a...z, 0...9).	SELECT ... FROM ... ORDER BY ... ASC
DESC	Sort results in descending order (z...a, 9...0).	SELECT ... FROM ... ORDER BY ... DESC
DISTINCT	Eliminates double entries in the query result.	SELECT DISTINCT ... FROM ...
OR	Performs a logical OR compare between two queries.	... WHERE ... OR ...
AND	Performs a logical AND compare between two queries.	... WHERE ... AND ...
IS [NOT] NULL	Returns the results that are [not] null	... WHERE ... IS [NOT] NULL
[NOT] IN	Returns the results that are [not] part of the following statement ... WHERE ... [NOT] IN ...	
[NOT] EXISTS	Returns the results that are [not] matching a given subquery.	... WHERE ... EXISTS ({{ SELECT ... }})
LIKE	Compares to a pattern.	... WHERE ... LIKE '....'
%	Wildcard matching any number of characters.	... WHERE ... LIKE '%....' '....%' '....%
-	Wildcard matching a single character.	... WHERE ... LIKE '....' '.....' '...._'
LEFT JOIN ON	Merges two tables into one.	... LEFT JOIN ... ON ... = ...
=	True if results are equal.	
!=, <>	True if results are not equal.	
<	True if result 1 is less than result 2.	
<=	True if result 1 is equal to or less than result 2.	
>	True if result 1 is greater than result 2.	
>=	True if result 1 is equal to or greater than result 2.	

SQL Command / Keyword	Description / Comment	Code Example
CONCAT	Concatenates two results - the example on the right hand side would return the string result.  Note that SQL Server 2012 doesn't provide a CONCAT function. Strings are connected via '+', for example 'foo' + 'bar' instead.	NCONCAT('resul', 't')
:o	Outer join parameter is used to include matches with missing rows in the ProductsLP table (= the table that holds localized products) as well. Otherwise, the example query would only return products with an existing row in ProductsLP table, because it would only use JOIN.	<pre>SELECT {p:PK} FROM {Product AS p} WHERE {p:description[en]:o} LIKE '%text%' OR {p:description[de]:o} LIKE '%text%'</pre>
{locAttr[ANY] }	A special version of a localized attribute condition, where an item is returned if any localization record holds a value for that attribute which fulfills the given condition regardless of the actual language. Technically, this means that the LP table is joined without a language parameter, which means that the same item may occur multiple times in the search result! You should use DISTINCT to compensate.	<pre>SELECT DISTINCT {p:PK} FROM {Product AS p} WHERE {p:description[ANY]} LIKE '%hybris%'</pre>

The <selects> Field

The values for the <selects> field specify the database columns to be returned.

The asterisk (*) returns all database columns, as by SQL convention. To search for an attribute, specify the attribute identifier in curly braces, such as: `SELECT {code} FROM {Product}`.

To retrieve values of localized attributes, use the language identifier as a suffix in the attribute name, enclosed in squared brackets ([and]), such as:

```
SELECT {name[de]}, {name[en]} FROM {Product}
```

Find two examples of different queries:

- `SELECT * FROM {Category}`

This query returns every database column from the **Category** table.

- `SELECT {pk}, {code}, {name[de]} FROM {Product}`

This query returns the database columns `pk`, `code`, and the German localized entries of the `name` column [`de`] from the `Product` table.

The <types> Field

The values for the `<types>` field in the `FROM` clause specify SAP Commerce types.

The values for the `<types>` field are nested in curly braces `{}` and `}` which are to be searched, such as:

```
SELECT * FROM {Product}
SELECT * FROM {Category JOIN Catalog}
```

You can specify an alias to be used for distinguishing attribute fields, using the `AS` operator:

```
SELECT {p.code} FROM {Product AS p} ORDER BY {p.code}
```

You may also run `JOIN` and `LEFT JOIN` queries, as in:

```
SELECT {cj.code}
FROM {SyncItemCronJob AS sicj
      JOIN SyncItemJob AS sij
      ON {sicj:job} = {sij:pk}}
}

SELECT {p1.PK},{p2.PK}
FROM {Product AS p1
      LEFT JOIN Product AS p2
      ON {p1.code} = {p2.code}}
}
WHERE {p1.PK} <> {p2.PK}
ORDER BY {p1.code} ASC
```

Always remember that it is most important that the whole `<types>` block must be enclosed by `{}` and `}` no matter how many types are actually inside it. Do not try to put in multiple `<types>` blocks in the `FROM` clause. Even though this may appear to be working, it may cause unpredictable errors.

Searching Subtypes

Specifying a type to search causes a FlexibleSearch query to search that type and any subtypes.

The following code snippet returns the codes and the PKs of all instances of `Product` and `VariantProduct`:

```
SELECT {code},{pk} FROM {Product}
```

By adding a trailing exclamation mark (`!`), the FlexibleSearch query searches only the specified type and omits all subtypes. For example, the following code snippet searches only instances of `Product`, not of `VariantProduct`:

```
SELECT {code},{pk} FROM {Product!}
```

When searching for subtypes, the FlexibleSearch first retrieves the subtypes to search, for example in the case of `Product`, types to search are `Product` and `VariantProduct`. As a type definition in SAP Commerce is an item and therefore has a Primary Key (PK), the FlexibleSearch retrieves the PK of all types to search. The list of the PKs of the types to search is put into an `IN` clause within the `WHERE` clause.

FlexibleSearch Query	SQL Statement
SELECT {p:code}, {p:pk} FROM {Product AS p}	SELECT item_t0.Code , item_t0.PK FROM products item_t0 WHERE (item_t0.TypePkString IN (23087380955301264 , 23087380955663520 , 23087380955662768 , 23087380955661760 , 23087385363574432 , 23087380955568768 , 23087380955206016))
SELECT {p:code}, {p:pk} FROM {Product! AS p}	SELECT item_t0.Code , item_t0.PK FROM products item_t0 WHERE (item_t0.TypePkString = 23087380955206016)

i Note

In the code above, the use of ':' is the legacy alternative to '.' and is still supported for backward compatibility and a few special features.

Changing FlexibleSearch Subtype Handling Behavior

Queries to types that have subtypes with separate deployment tables are joined using the `UNION ALL` clause. Sometimes this behavior can lead to duplicated records. You can change this behavior by setting the `unionAllTypeHierarchy` property to `false` in a current session. FlexibleSearch will use the `UNION` clause instead of `UNION ALL` as a result.

```
sessionService.executeInLocalViewWithParams(ImmutableMap.of(de.hybris.platform.jalo.flexiblesearch.  
    new SessionExecutionBody()  
    {  
        @Override  
        public List<Object> execute()  
        {  
            return r.get();  
        }  
    }));
```

Excluding Types from a Search

You can omit certain types from a FlexibleSearch query run.

If you want to make sure that certain types are omitted from a FlexibleSearch query run, there are two approaches at your disposal:

- Using the `itemtype` operator and a parameter.

This approach is feasible if you can prepare and pass a Map with references to the types you want to exclude as a FlexibleSearch parameters, such as:

```
final Set<ComposedTypeModel> excludedTypes = new HashSet<ComposedTypeModel>();  
excludedTypes.add(getComposedType("mySuborderType"));  
  
StringBuilder queryString = new StringBuilder("SELECT {").append(OrderModel.PK).append("} ");  
queryString.append("FROM {").append(OrderModel._TYPECODE).append("} ");  
queryString.append("WHERE {").append(OrderModel.ITEMTYPE).append("} NOT IN (?excluded)");
```

```
final FlexibleSearchQuery query = new FlexibleSearchQuery(queryString.toString(), Collections
```

- Using a JOIN clause

This approach is feasible if you cannot pass parameters, for example, because you need to enter a FlexibleSearch statement directly:

```
SELECT {o.PK} FROM {Order AS o JOIN ComposedType AS t ON {o.itemtype}={t.PK} }
WHERE {t.code} NOT IN ( 'Foo', 'Bar' )
```

The <conditions> Field

The values for the <conditions> field in the optional WHERE clause narrow down the number of matches by specifying at least one condition that is matched by all search results.

i Note

Make sure to avoid spaces at the beginning and end of the search condition term, as = 'al' and = 'al ' are not identical search conditions and cause different search results.

Avoid Spaces in Search Condition Terms

- SELECT * FROM {Product} WHERE {code} LIKE '%al%'
- Using the common SQL boolean operators (AND, OR) you can connect conditions, such as:

```
SELECT * FROM {Product} WHERE {code} LIKE '%al%' AND {code} LIKE '%15%'
```

- Use the IS NULL operator to find all entries that have no value:

```
SELECT * FROM {Product} WHERE {code} IS NULL
```

- Negating a condition is possible using the SQL boolean operators NOT:

```
SELECT * FROM {Product} WHERE {code} NOT LIKE '%al%'
```

- It is possible to combine negating and connecting conditions:

```
SELECT * FROM {Product} WHERE {code} LIKE '%al%' AND {code} NOT LIKE '%15%'
```

- The negation of the IS NULL operator is IS NOT NULL:

```
SELECT * FROM {Product} WHERE {code} IS NOT NULL
```

- The WHERE clause also allows subselects using double curly braces ({{ and }}), such as:

```
SELECT {cat:pk} FROM {Category AS cat} WHERE NOT EXISTS (
  {{ SELECT * FROM {CategoryCategoryRelation} WHERE {target}={cat:pk} }} /* Sub-select */
)
```

The <order> Field

The FlexibleSearch complies with the SQL syntax in terms of ordering results. By specifying an attribute in an ORDER BY clause, the list of search results are sorted according to the specified type.

You can also optionally specify ASC to sort the search results in ascending order (null, 0 through 9, A through Z) or DESC to sort the search results in descending order (Z through A, 9 through 0, null). ASC and DESC are mutually exclusive, ASC is default.

Examples:

- The following FlexibleSearch query sorts the search results by the values of the code database column, in descending order:

```
SELECT {code},{pk} FROM {Product} ORDER BY {code} DESC
```

- The following FlexibleSearch query sorts the search results by the values of the code database column, in ascending order: (ASC is default order):

```
SELECT {code},{pk} FROM {Product} ORDER BY {code}
```

Parameters

A FlexibleSearch query optionally contains parameters, marked by a prefixed question mark.

Parameters enable you to pass values into the FlexibleSearch query. For example, in the following code snippet, the parameter product can be used to pass a search pattern:

```
SELECT {p:pk} FROM {Product AS p} WHERE {p:code} LIKE ?product
```

The following FlexibleSearch query has two parameters, startDate and endDate:

```
SELECT {pk} FROM {Product} WHERE {modifiedtime} >= ?startDate AND {modifiedtime} <=?endDate
```

Parametrized Queries

Parametrized queries, also called prepared statements or parametrized statements, allow you to create templates you can then use to repetitively run SQL statements in the database management system (DBMS) of your choice. The generic nature of parametrized queries improves the performance of database operations through the use of parameters. They also reduce the risk of SQL injections, because parameter values in the query templates are automatically escaped.

To use parametrized queries, first prepare a template statement of a generic query. Leave certain values of that query unspecified as in the following plain SQL example:

```
SELECT FROM <table name> VALUES (?, ?, ?);
```

? are the values the DBMS parses, then stores the template and its result without executing it. The stored template is ready for the application to supply the values and running the query as needed. Before using the parametrized queries, examine and determine advantages of using them. There are cases in which running a single query makes for a better database performance than using a parametrized query. Such situations include running a query only once, or not enough cache space.

The following code samples are an example of how to use a parametrized query in FlexibleSearch to return the list of inactive catalogs based on their ID. First, determine the parameters:

```
SELECT {catalogVersion.pk} FROM {CatalogVersion AS catalogVersion JOIN Catalog as catalog ON {catalogVersion.catalogPK = catalog.PK}}
```

Using it in a Groovy script in the Administration Console returns the list of inactive catalogs:

```
import de.hybris.platform.catalog.model.CatalogModel
import de.hybris.platform.catalog.model.CatalogVersionModel
import de.hybris.platform.servicelayer.search.FlexibleSearchQuery
import de.hybris.platform.servicelayer.search.FlexibleSearchService

final String catalogVersionQuery = "SELECT {catalogVersion." + CatalogVersionModel.PK + "}" +
    "FROM {" + CatalogVersionModel._TYPECODE + " AS catalogVersion " +
    "JOIN " + CatalogModel._TYPECODE + " as catalog " +
    "ON {catalog." + CatalogModel.PK + "} = {" + CatalogVersionModel.CATALOG + "}}" +
    "WHERE {catalog." + CatalogModel.ID + "} = ?id AND {catalogVersion.active} = ?isActive"

final FlexibleSearchQuery catalogVersionFlexibleSearchQuery = new FlexibleSearchQuery(catalogVersionQuery)

catalogVersionFlexibleSearchQuery.addQueryParameter("id", "testCat")
catalogVersionFlexibleSearchQuery.addQueryParameter("isActive", false)

FlexibleSearchService flexibleSearchService = spring.getBean("flexibleSearchService")
List<CatalogVersionModel> inactiveCatalogVersions = flexibleSearchService.search(catalogVersionFlexibleSearchQuery)

return inactiveCatalogVersions.collect { catalogVersion -> catalogVersion.getVersion() }.join(", ")
```

No-Cache Mode for FlexibleSearch

FlexibleSearch heavily uses cache for storing search results. In most cases, this behavior is desirable but sometimes it is better to skip cache completely.

When a query contains a frequently changing parameter such as current time in milliseconds, or a query result is huge and actually would pollute the cache, it's better to use the no-cache mode for FlexibleSearch. The FlexibleSearchService allows you to easily skip cache on demand. The following code example shows how to do that:

```
final FlexibleSearchQuery fQuery = new FlexibleSearchQuery("SELECT {PK} FROM {Foo} WHERE {modificationTime <= ?time}")
fQuery.addQueryParameter("modificationTime", Long.valueOf(System.currentTimeMillis()));
fQuery.setDisableCaching(true);

final SearchResult<FooModel> searchResult = flexibleSearchService.search(fQuery);
```

By simply using the `setDisableCaching` method on a `FlexibleSearchQuery` object, you are instructing FlexibleSearch to skip cache just for the current query.

Keep in mind that the type `Foo` used in the example above is an artificial one.

Using FlexibleSearch in Backoffice

Triggering FlexibleSearch queries within Backoffice is possible in two ways: using the Backoffice Saved Queries functionality or using `ViewType` instances.

A `ViewType` instance is SAP Commerce representation of a database view. The `ViewType` representation in Backoffice is called a Report Definition. The Backoffice Saved Queries feature provides a way to execute custom FlexibleSearch queries and browse the results. A `SavedQuery` instance is a means of using a FlexibleSearch query to retrieve items in the SAP Commerce instead of using the GenericSearch.

Using FlexibleSearch Using the SAP Commerce API

Using FlexibleSearch queries using the SAP Commerce API takes place in two steps, both of which can be done in one Java statement: setting up and running the query.

Constructing a FlexibleSearch Query

A FlexibleSearch query is constructed as a string which contains the query, such as:

```
final String query = "SELECT {pk} FROM {Product}"
// Flexible search service injected by Spring
final SearchResult<ProductModel> searchResult = flexibleSearchService.search(query);
```

To refer to a SAP Commerce type attribute in the FlexibleSearch query such as the primary key (PK) of an item, you need to reference the attribute when constructing the query. In cases where the attribute is clear without ambiguity, specifying the attribute alone is enough. Still, it is recommended to reference the type of the attribute as well for unambiguity. SAP Commerce resolves and translates the attribute reference automatically into the FlexibleSearch query:

Examples:

Java Code	FlexibleSe
final String query = "SELECT {" + ProductModel.PK + "} FROM {" + ProductModel._TYPECODE + "}";	SELECT
String query = "SELECT {p:" + ProductModel.PK + "} FROM {" + ProductModel._TYPECODE + " AS p}\n"+ "WHERE {" + ProductModel.VARIANTTYPE + "} IS NOT NULL"	SELECT WHERE

Calling a FlexibleSearch

→ Tip

Use Paging on Queries with Many Results

If you run a FlexibleSearch query that potentially returns more than 50 search results, be sure to use the paging mechanism of the FlexibleSearch described in the **Hints** section.

To call a FlexibleSearch statement using the API use `flexibleSearchService`, which is always available through the Spring, and has to be properly injected to your service as follows:

```
<bean id="myFancyService" class="de.hybris.platform.foobar.MyFancyService" >
  <property name="flexibleSearchService" ref="flexibleSearchService"/>
</bean>
```

```
public class MyFancyService implements FancyService
{
```

```

...
private FlexibleSearchService flexibleSearchService;

@Required
public void setFlexibleSearchService(final FlexibleSearchService flexibleSearchService)
{
    this.flexibleSearchService = flexibleSearchService;
}
...
}

```

The `flexibleSearchService search(...)` method returns a `de.hybris.platform.servicelayer.search.SearchResult` instance, which holds a List of the individual search results. To access this List, call the `SearchResult` class `getResult()` method, such as:

```

final String query = "SELECT {" + ProductModel.PK + "} FROM {" + ProductModel._TYPECODE + "}";
final SearchResult<ProductModel> searchResult = flexibleSearchService.search(query);
List<ProductModel> result = searchResult.getResult();

```

You can then process this `Collection` instance like any other `Collection` instances:

```

final String query = "SELECT {" + ProductModel.PK + "} FROM {" + ProductModel._TYPECODE + "}";
final SearchResult<ProductModel> searchResult = flexibleSearchService.search(query);
final ProductModel product = searchResult.getResult().iterator().next();

```

⚠ Caution

The Collection returned by `SearchResult.getResult()` uses the lazy translation approach. At first access to a collection element, the element is translated to an item.

In case the item was removed between gathering of the search result and translation of the specific element, the returned collection has a null value at this position.

Passing Parameters

To pass parameters, create a Map instance holding the parameters and pass the Map to the `search(...)` method, as in:

→ Tip

If you do not need to pass parameters to the query, you can pass `null` for the parameter map.

```

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

String query =
"SELECT {" + PriceRowModel.PK + "} FROM {" + PriceRowModel._TYPECODE + "} "+
"WHERE {" + PriceRowModel.PRODUCT      + "} = ?product AND "+
"{" + PriceRowModel.NET      + "} = ?net AND "+
"{" + PriceRowModel.CURRENCY      + "} = ?currency AND "+
"{" + PriceRowModel.UNIT      + "} = ?unit AND "+
"{" + PriceRowModel.UNIT_FACTOR      + "} = ?unitfactor AND "+
"{" + PriceRowModel.UG      + "} = ?userpricegroup AND "+
"{" + PriceRowModel.MIN_QUANTITY      + "} = ?minquantity AND "+
"{" + PriceRowModel.PRICE      + "} = ?price ";

params.put("product",           product);
params.put("net",               priceCopy.isNet());
params.put("currency",          priceCopy.getCurrency());
params.put("unit",              priceCopy.getUnit());
params.put("unitfactor",        priceCopy.getUnitFactor());
params.put("userpricegroup",   priceCopy.getUserPriceGroup());
params.put("minquantity",      priceCopy.getMinQuantity());

```

```
params.put("price", priceCopy.getPriceValue());
final SearchResult<PriceRowModel> searchResult = flexibleSearchService.search(query, params);
```

i Note

If your query uses the statement `?product`, then the key in the parameter map has to be `product` (without the `?`).

Instantiating of Search Results

If you retrieve only the PK database column (that is, only the PKs of SAP Commerce items), and provide the kind of type as a Java class, you can immediately cast the models represented by the PKs into the actual model instances. In other words, executing the following code returns a **Collection** of `CatalogModel` instances, not a **Collection** of PKs:

```
final String query = "SELECT {" + CatalogModel.PK + "} FROM {" + CatalogModel._TYPECODE + "} ORDER BY _TYPECODE";
final SearchResult<CatalogModel> searchResult = flexibleSearchService.search(query);
```

If you retrieve more than one database column, you receive several individual entries per row of result and you will not be able to cast the search result into item instances directly, not even if one of the database columns retrieved is the PK column.

Paging of Search Results

Managing more than some 50 or 100 search results in one single Collection is complicated and performs comparably slow. For this reason, the FlexibleSearch framework offers a paging mechanism.

Some FlexibleSearch queries run the risk of returning a very large number of search results, such as `SELECT * FROM {Products} WHERE {code} LIKE ?search OR {name} LIKE ?search`, where `?search` is a parameter from a text field.

To use this paging mechanism, use the `search(...)` method with `FlexibleSearchQuery` object as parameter. You have to set on `FlexibleSearchQuery` the `setNeedTotal` to `true`. If this parameter is set to `true`, the FlexibleSearch framework splits the number of returned search results into pages. Using the `start` and `range` parameters, you can retrieve pages of search results. The following code snippet, for example, iterates over all the search results of the FlexibleSearch query, three at a time:

```
int start = 0;
final int range = 3;
int total;

String query = "SELECT {" + UnitModel.PK + "} FROM {" + UnitModel._TYPECODE + "} ORDER BY " + UnitModel._NAME;
final FlexibleSearchQuery fQuery = new FlexibleSearchQuery(query);
fQuery.setCount(range);
fQuery.setNeedTotal(true);

do
{
    fQuery.setStart(start);
    final SearchResult<LanguageModel> searchResult = flexibleSearchService.search(fQuery);
    total = searchResult.getTotalCount();
    start += range;
}
while(start < total);
```

Be aware that every navigation, either backward or forward, through a paged search result triggers a new search query on the database. Internally, the FlexibleSearch runs the query in full and uses an offset parameter to specify the portion of all search results to return. The fact that every navigation causes a database query has three major consequences:

1. Complex queries cause heavy load on the database:

Executing a simple SELECT statement is rather fast, even with millions of search results. However, if your FlexibleSearch query requires JOIN or UNION to execute, load on the database (and, by consequence, response times) increases rapidly. As a rule of thumb, remember that the more different items are involved, the longer the execution time is. For example, the following table gives some short examples of some rather basic FlexibleSearch statements and the actual SQL queries triggered:

FlexibleSearch statement	SQL statement	Description
<pre>SELECT {pk} FROM {Product}</pre>	<pre>SELECT item_t0.PK FROM products item_t0 WHERE (item_t0.TypePkString IN (23087950835790560 , 23087950835774560 , 23087950837855968 , 23087950837852464 , 23087950837859216 , 23087950837848976 , 23087950837843968 , 23087950835790306 , 23087950835765569))</pre>	This FlexibleSearch statement results in no big surprises. Basically, SAP Commerce translates the type system related data into the matching SQL data. Note that the actual database table to be searched is <code>products</code> , and not <code>product</code> as you might expect.
<pre>SELECT {description} FROM {Product}</pre>	<pre>SELECT lp_t0.p_description FROM productslp lp_t0 WHERE ((lp_t0.LANGPK=?)) AND (lp_t0.ITEMTYPEPK IN (23087950835790560 , 23087950835774560 , 23087950837855968 , 23087950837852464 , 23087950837859216 , 23087950837848976 , 23087950837843968 , 23087950835790306 , 23087950835765569))</pre>	Even though you run the FlexibleSearch on the Product type, the SQL statement is run on the <code>productslp</code> table instead of on the <code>products</code> table. This is because the <code>description</code> attribute is localized, and localized attributes for a type are stored in an individual database table.
<pre>SELECT {code}, {description} FROM {Product}</pre>	<pre>SELECT item_t0.Code , lp_t0.p_description FROM products item_t0 JOIN productslp lp_t0 ON item_t0.PK = lp_t0.ITEMPK AND lp_t0.LANGPK=? WHERE ()</pre>	The seemingly simple example, <code>SELECT {code}, {description} FROM {Product}</code> , requires a JOIN

FlexibleSearch statement	SQL statement	Description
	<pre> item_t0.TypePkString IN (23087950835790560 , 23087950835774560 , 23087950837855968 , 23087950837852464 , 23087950837859216 , 23087950837848976 , 23087950837843968 , 23087950835790306 , 23087950835765569)) </pre>	over the products table and the localized tables of the products table (products_lp) in order to get ahold of the localizations of the description attribute as well. Whenever you run FlexibleSearch statements on localized and non-localized attributes of the same type, a JOIN is necessary.

2. Search results may differ over time:

When a FlexibleSearch query is run for the first time, the search results correspond to the dataset in the database by execution time. As every navigation through paged search results runs a new database query, the FlexibleSearch statement always receives a list of search results which correspond to the point of time when the statement is executed. This means, however, that the list of search results does not necessarily match the list of search results which were found when the FlexibleSearch query was initially executed. Items referenced in the initial query may have been updated, created, or removed in the meantime.

3. ORDER BY required:

The order in which unordered database search results are returned may differ on every database statement execution, depending on the database implementation. By consequence, an unordered FlexibleSearch query might receive different search results on every navigation through paged search results. To avoid seemingly random search results, using paged search results requires an ORDER BY clause in the FlexibleSearch. Note, that ordering the paged search results does not solve the issue with search results differing over time.

Specifying a Minimum Time to Live for Cached FlexibleSearch Results

The system enables you to specify a minimum time to live for FlexibleSearch results. For more information, see [Specifying a Minimum TTL for Cached FlexibleSearch Results](#).

Testing FlexibleSearch Queries Using The SAP Commerce Administration Console

The SAP Commerce offers a tool to test FlexibleSearch queries.

Procedure

1. Open the SAP Commerce Administration Console.

For more information, see [Accessing the Administration Console](#).

2. Go to the **Console** tab and select the **FlexibleSearch** option.
3. The **FlexibleSearch** page displays. If you do not have any queries ready, you can use **Query samples** located in the sidebar.
4. Enter the query in **FlexibleSearch query** or **Direct SQL query** field.
5. Click the **Execute** button.

Restrictions

Restrictions are rules obeyed by FlexibleSearch which allow to limit search results depending on which type is searched and which user is currently logged in.

This happens transparently and does not require any code adjustment on business layer.

A restriction is basically just a fragment of the **WHERE** clause of a [FlexibleSearch](#) statement - that includes other UserGroups (whose members are affected by the restriction as well). A restriction always applies to a specified type and a specified User or UserGroup. It automatically adds them to the **WHERE** clauses of all applicable FlexibleSearch statements and thereby restricts the number of search results of these statements due to these additional search conditions.

For example:

FlexibleSearch Statement	Restriction Statement	Effective
<pre>SELECT {p:pk} FROM {Product AS p} WHERE {p:code} LIKE '%test%' statement - that is, a restriction adds search conditions.</pre>	{p:description} NOT NULL	SELECT WHERE AND {p}

Scope of Restrictions

The effect of restrictions is transparent, no interaction is necessary. Whenever a restriction is active and applies to the combination of restricted type and user, the search results are limited.

Unlike type access rights (which are only effective within Backoffice), restrictions apply to FlexibleSearch results throughout SAP Commerce. In other words, restrictions affect FlexibleSearch results in Backoffice, in SAP, and in an SAP Commerce-based web application. Type access rights only affect Backoffice.

Since restrictions work on FlexibleSearch queries only, restrictions do not affect the following use cases:

- External search engines

Search results supplied by third-party search engines are not affected by restrictions. To get third-party search engine search results affected by restrictions, you need to filter these search results by running a FlexibleSearch statement over them.

- **Item.getProperty(), LocalizableItem.getLocalizedProperty()** fetches item references directly via PK - any stored item is returned no matter if it would have been filtered by a currently active restriction

SAP Commerce includes a mechanism that allows using restrictions for cron jobs. For details, please refer to [cronjob - Technical Guide](#).

Using Parameters in Restrictions

Often it will do to specify restrictions having literal query texts like

IS NOT NULL

or `{hidden}=1`.

But sometimes it may also be required to specify a restriction which relies upon a session bound parameters instead of a fixed literal value. To do so use the `?session` FlexibleSearch parameter which is available inside every query.

Now we're able to write restrictions like `{user} = ?session.user` or `{country} IN (?session.countries)` which gives us the freedom to filter differently depending upon which session context settings have been made.

Caution

When referring to custom session context attribute (like `countries` above) make sure that the session context **does** contain this attribute. Otherwise any attempt to run a query which is affected by the restriction will throw a FlexibleSearch error!

Disabling Restrictions

For development, testing and debugging purposes, it may prove useful to disable restrictions as they **falsify** FlexibleSearchService query results.

Assigning the Session to an Admin User

By convention, restrictions do not apply to admin users (that is, users who are members of the `admingroup` user group). This means that a session assigned to a member of the `admingroup` user group is not affected by restrictions. This setting is in effect as long as the session is assigned to an admin user and applies to all restrictions.

Note

Using Filters

Assigning the session to an admin user has the side effect of granting the session access to every CatalogVersion in SAP Commerce. The default user assigned to a session (`anonymous`) requires that CatalogVersions be set for the session explicitly. To make sure that CatalogVersions are set for non-admin user sessions, you need to integrate filters in your Web application. You need to use the Platform filter chain. Use it to add proper filter for catalog version activation into the filter chain. Read [hybris Platform Filters](#) for more details.

Find an example on how to assign the session to the `admin` user below:

```
...
import de.hybris.platform.servicelayer.user.UserService;
...
@Autowired
private UserService userService;
...
userService.setCurrentUser(userService.getAdminUser());
...
```

The following code snippet shows how to get Products for a category available for `admin` user. The important part here is the fact that setting `admin` user in the session as well getting products is executed in local view by using `sessionService.executeInLocalView`.

```
...
import de.hybris.platform.product.ProductService;
import de.hybris.platform.servicelayer.session.SessionService;
...
```

```

import de.hybris.platform.servicelayer.session.SessionService;
import de.hybris.platform.servicelayer.user.UserService;
...
@Autowired
private SessionService sessionService;
@Autowired
private UserService userService;
@Autowired
private ProductService productService;
...
public List<ProductModel> getProductsByCategory()
{
    return (List<ProductModel>) sessionService.execute();
}
@Override
public List<ProductModel> execute()
{
    userService.setCurrentUser(userService.getAdminUser());
    return productService.getProductsForCategory(getSomeCategory());
}
}

private CategoryModel getSomeCategory()
{
    ...
    return someCategoryHere;
}

```

Enabling or Disabling Search Restrictions

Find below an example of how to enable and disable a search restriction:

```

...
import de.hybris.platform.search.restriction.SearchRestrictionService;
...
// Disable search restrictions
searchRestrictionService.disableSearchRestrictions();
// some query goes here

// Enable search restrictions
searchRestrictionService.enableSearchRestrictions();
// some query goes here

```

Creating Restrictions

A restriction is represented by a **SearchRestriction** class instance, which has the following mandatory attributes:

SearchRestriction Attribute	Allowed Value	Description
active	java.lang.Boolean	Specifies whether the SearchRestriction is enabled (true) or disabled (false). Defaults to true .
code	java.lang.String	The identifier for the restriction. Must be unique throughout all SearchRestriction type instances.
principal	Principal	The user or user group to whom the restriction applies.

SearchRestriction Attribute	Allowed Value	Description
query	java.lang.String	The query of the SearchRestriction, that is, a WHERE clause of a FlexibleSearch statement that narrows down a FlexibleSearch statement.
restrictedType	ComposedType	The type on which FlexibleSearch queries are to be restricted when executed with the specified principal.
generate	java.lang.Boolean	Defines whether the build process should generate a jalo code for this type system element. SearchRestriction extends TypeManagerManaged that has the non-optional <code>generate</code> field.

You can create a restriction by using the SAP Commerce API, and with ImpEx.

To learn how to create a restriction in Backoffice, see [Creating Restrictions in Backoffice](#).

Creating Restrictions via the SAP Commerce API

To create a restriction we need to use model service to create a new `SearchRestrictionModel` instance and then populate it with desired attributes, for example:

```
final ComposedTypeModel restrictedType = typeService.getComposedTypeForClass(LanguageModel.class);
final PrincipalModel principal = userService.getUser(principalId);
final SearchRestrictionModel searchRestriction = modelService.createSearchRestriction();
searchRestriction.setCode("some code");
searchRestriction.setActive(Boolean.TRUE);
searchRestriction.setQuery("{active} IS TRUE");
searchRestriction.setRestrictedType(restrictedType);
searchRestriction.setPrincipal(principal);
searchRestriction.setGenerate(Boolean.TRUE);
modelService.save(searchRestriction);
```

Creating Restrictions via the ImpEx Extension

The following sample code shows how to create restrictions using an Impex-compliant CSV file:

```
INSERT_UPDATE SearchRestriction;code[unique=true];name[lang=de];name[lang=en];query;principal(UUID);
;Frontend_Navigationelement;Navigation;Nav;
```

Related Information

[ImpEx](#)

[Visibility Control](#)

[The Cronjob Service](#)

Creating Restrictions in Backoffice

Define a new personalization rule in Backoffice Administration Cockpit to create restrictions for FlexibleSearch.

Procedure

1. Log in to Backoffice using an account with sufficient rights to create restrictions.
2. Go to .
3. Click the to open the **Create New Personalization rule** window.
4. Enter your restriction query string into the **Filter** field.
5. Click the next to the **Apply on** field and select a user or a user group for the restriction to be effective on.
6. Click the next to the **Restricted Type** field and select a type to apply the restriction to.
7. Give your new restriction a unique identifier.
8. Click **Done** to save the new restriction.

FlexibleSearch Samples

This document discusses a number of FlexibleSearch samples. As the FlexibleSearch is a key component of SAP Commerce, reading this document is recommended for all developers.

This document does not discuss the FlexibleSearch in general, please refer to [FlexibleSearch](#) for general information on FlexibleSearch.

Basic SELECT Statements

This section discusses FlexibleSearch statements with one single **SELECT** operator. Some of these samples overlap in terms of operators, this is hard to avoid for such a subject.

SELECT Statements with Negation

The following FlexibleSearch statements are samples for using the negation operator **NOT**.

- **Getting all Products whose code is not empty**

The following FlexibleSearch statement returns Primary Keys (PK) of every **Product** whose **code** attribute is different from **null**. Be aware that in SQL syntax the empty string "" is not considered to be **null**. In other words: the following FlexibleSearch statement finds **Products** whose code is set to "".

```
SELECT {p.pk} FROM {Product AS p} WHERE {p.code} IS NOT NULL
```

- **Getting all Categories whose code does not contain a certain string**

The following FlexibleSearch statement returns the PKs of every **Category** whose **code** attribute does not contain the string **test**.

```
SELECT {c:pk} FROM {Category AS c} WHERE {c:code} NOT LIKE '%test%'
```

SELECT Statements with Several Return Columns

The arguments passed for the **SELECT** operator specify the columns from the database that are to be returned by the FlexibleSearch query.

- **Returning every database column of every Category**

The following FlexibleSearch statement returns every database column of every Category in SAP Commerce. The list of returned database columns could be something along the lines of: `hjmpts`, `modifiedts`, `createdts`, `typepkstring`, `pk`, `ownerpkstring`, `accts`, `propts`, `p_showemptyattributes`, `p_normal`, `p_thumbnails`, `p_revision`, `p_code`, `p_data_sheet`, `p_logo`, `p_catalogversion`, `p_picture`, `p_detail`, `p_catalog`, `p_others`, `p_order`, `p_thumbnail`, and `p_externalid`.

```
SELECT * FROM {Category}
```

- **Getting the point of time when a Category was last modified, Category code and PK**

The following FlexibleSearch statement returns three database columns of every **Category** in SAP Commerce. Note that the `code` attribute is enclosed by curly braces.

```
SELECT {cat:modifiedtime}, {cat:code}, {cat:pk} FROM {Category AS cat}
```

SELECT Statements Over Several Attributes

A FlexibleSearch statement allows narrowing down the list of search results by specifying several attributes in a search condition. A SAP Commerce item must match the search condition in the respective attribute to become included in the search result list. For example, if the FlexibleSearch statement queries for the `code` and the `name` attribute, the list of search results contains only items that have a matching value in both the `code` and the `name` attribute.

- **Getting all Products whose code or name contains a certain string**

The following FlexibleSearch returns the PKs of all Products whose `code` attribute or `name` attribute contains a string that ends with `myProduct`.

```
SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:code} LIKE '%myProduct'
   OR {p:name} LIKE '%myProduct'
 ORDER BY {p:code} ASC
```

The percent sign (%) works as a wildcard character:

- `a%` finds all strings that start with an `a`,
- `%a` finds all strings that end with an `a`,
- `%a%` finds all strings that contain an `a`.

By introducing a parameter (`?name` in the following FlexibleSearch query) into the query, you can search for any search string:

```
SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:code} LIKE ?name
   OR {p:name} LIKE ?name
 ORDER BY {p:code} ASC
```

Be aware that the search condition `WHERE LIKE ?name` only finds products whose `name` attribute matches the value of the `?name` parameter exactly. To find **Products** whose `name` attribute contains the value of the `?name` parameter only partially, you need to use wildcard characters, as in `WHERE LIKE CONCAT('%', ?name, '%')` or enclosing parameter by wildcard characters like:

`...LIKE ?name; query.addQueryParameter("name", "%h%")`. Both solutions are SQL-injection safe.

SELECT Statements Over Several Languages

SAP Commerce allows for attributes to be localized, that is, to have an individual value for each language in SAP Commerce. By specifying the language code enclosed by square brackets ([and], respectively) after the attribute name, such as **{description[de]}**. Not specifying a language code for a localized attribute causes SAP Commerce to use the default language for the current session.

- **Getting all Products with an empty name in the current SAP Commerce language**

The following FlexibleSearch query returns the PKs of all **Products** whose **name** attribute is not set (IS NULL). The **name** attribute is localized, but the FlexibleSearch query does not specify a language explicitly, therefore the FlexibleSearch defaults to the current session language.

```
SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:name} IS NULL
```

- **Getting all Products with an empty name in German or an empty description in English**

The following FlexibleSearch query returns the PKs of all **Products** and whose...

- **name** attribute is not set for the German language (IS NULL) or
- **description** attribute is not set for the English language (IS NULL).

Both the **name** attribute and the **description** attributes are localized. The FlexibleSearch query specifies the language explicitly (via de and en, respectively). If no language is specified, the FlexibleSearch would default to the current session language.

```
SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:name[de]} IS NULL
   OR {p:description[en]} IS NULL
```

- **Searching several languages at once**

By specifying different language codes, you can search localized attributes in various languages at a time. The following FlexibleSearch statement searches both the English and German values of the **description** attribute:

```
SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:description[en]:o} LIKE '%text%'
   OR {p:description[de]:o} LIKE '%text%'
```

Here, **:o** (outer join) parameter is used to include matches with missing rows in the **ProductsLP** table (= the table that holds localized products) as well. Otherwise, the query would only return products with an existing row in **ProductsLP** table, because it would only use JOIN.

→ Tip

Add OR Clause to Search Additional Attributes or Languages

To search another language, add the attribute to be searched with an explicit specification of the language to be searched to the **WHERE** clause via an **OR** clause. For example, the following FlexibleSearch statement searches the **description** attribute of the **Product** in the three languages English, German, and French (**en**, **de**, and **fr**, respectively) and the **name** attribute in German:

```
SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:description[en]:o} LIKE '%text%'
   OR {p:description[de]:o} LIKE '%text%'
   OR {p:name[de]:o} LIKE '%text%'
   OR {p:description[fr]:o} LIKE '%text%'
```

It is also possible to replace the hard-coded search string with a parameter. Please refer to the [SELECT statements with parameters](#) section below for details.

SELECT Statements with Parameters

A parameter in a FlexibleSearch query allows inserting varying search patterns. This is a common field of use for applications where one single query is intended to be used for various searches, such as:

- - Search fields in the store frontend.
 - Search fields in Backoffice.
 - Item retrieval in the application business code.
- **Using one parameter in a FlexibleSearch statement**

The following FlexibleSearch statement queries the **description** attribute in three SAP Commerce languages (**en**, **de**, **fr**) for one single parameter, **?param**. In other words, the FlexibleSearch finds all **Product** instances whose description in English, German, or French contains the search pattern specified by **?param**.

```
SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:description[en]:o} LIKE ?param
 OR {p:description[de]:o} LIKE ?param
 OR {p:description[fr]:o} LIKE ?param
```

Note, that there are **:o** characters in **WHERE** clause. They are used to force the related table to be outer-joined (in case of localized properties the **xxxLP** table).

- **Using two parameters in a FlexibleSearch statement**

- Getting every **Product** that is in at least one of two **Categories**:

```
SELECT {cpr:target}
  FROM {CategoryProductRelation AS cpr}
 WHERE {cpr:source} LIKE ?param1
 OR {cpr:source} LIKE ?param2
```

- Getting every **Product** that was changed between two dates:

```
SELECT {pk}
  FROM {Product}
 WHERE {modifiedtime} >= ?startDate
 AND {modifiedtime} <= ?endDate
```

SELECT Statements with Concatenation

The FlexibleSearch feature allows concatenating strings within a statement. Be aware that each **CONCAT** operator call allows only two parameters. To concatenate more than two parameters, you need to run more than one **CONCAT** operator calls.

- **Enclosing a search string by percent signs %**

The following FlexibleSearch statement gives an example on the **CONCAT** operator by concatenating the leading **%** character with the concatenation of **myProduct** and the **%** character for a total of **%myProduct%**:

```
SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:description[de]}
       LIKE
        CONCAT(
          '%',
          'myProduct',
          '%')
```

```

        CONCAT(
            'myProduct',
            '%'
        )
    )
OR {p:description[en]}
LIKE
    CONCAT(
        '%',
        CONCAT(
            'myProduct',
            '%'
        )
    )
)
ORDER BY {p:code} ASC

```

This function is useful in combination with parameters, as in the following FlexibleSearch statement:

```

SELECT {p:PK}
FROM {Product AS p}
WHERE {p:description[de]} LIKE
    CONCAT(
        '%',
        CONCAT(
            ?param,
            '%'
        )
    )
)
OR {p:description[en]} LIKE
    CONCAT(
        '%',
        CONCAT(
            ?param,
            '%'
        )
    )
)
ORDER BY {p:code}

```

SELECT Statements with DISTINCT Operator

The **DISTINCT** operator makes sure that duplicate results are returned only once. Duplicate return results may occur from sub-selects, **JOIN** clauses or from identical parameters, for example.

- **Finding every Product that is in at least one of two given Categories**

The following FlexibleSearch statement returns every **Product** that is assigned to at least one of the two **Categories** provided by the two parameters **?param1** and **?param2**. The **DISTINCT** operator ensures that every **Product** is returned only once even if it were assigned to both **Categories**.

```

SELECT DISTINCT {cpr:target}
FROM {CategoryProductRelation AS cpr}
WHERE {cpr:source} LIKE ?param1
    OR {cpr:source} LIKE ?param2

```

SELECT Statements with GROUP BY Operator

- **Getting every Product which has been ordered, grouped by the Product**

```

SELECT {oe:product}
FROM {OrderEntry AS oe}
GROUP BY {oe:product}

```

Subselects

A subselect is a **SELECT** statement within a **SELECT** statement. Via a subselect, a **SELECT** statement can affect (narrow down or expand, for example) a search result list. The basic syntax looks as follows:

```
SELECT *
  FROM ${type}
 WHERE

{{{
  SELECT *
    FROM ${other_type}
   WHERE ${subselect_search_condition}
}}}
```

i Note

Subselects in **FROM** clause of the query are also allowed (the example above presents subselect in the **WHERE** clause of the query). See [Subselect with Parameters](#) section below for more details.

Subselect Over Several Types

- **Getting every Product that has a directly or indirectly assigned PriceRow**

The following FlexibleSearch statement lists every Product that:

- has at least one **DiscountRow** directly assigned to it (**subselect 1**) or
- is assigned to a **ProductDiscountGroup** that has a **DiscountRow** assigned to it (**subselect 2**)

```
SELECT DISTINCT {p:PK}, {p:name}, {p:code}
  FROM {Product AS p}
 WHERE {p:PK} IN
 (
  {{{
    -- subselect 1
    SELECT {dr:product}
      FROM {DiscountRow AS dr}
  }}}
 )
 OR {p:PK} IN
 (
  {{{
    -- subselect 2
    SELECT {prod:PK}
      FROM
      {
        Product AS prod
        LEFT JOIN DiscountRow AS dr
        ON {prod:Europe1PriceFactory_PDG} = {dr:pg}
      }
      WHERE {prod:Europe1PriceFactory_PDG} IS NOT NULL }}
 )
 ORDER BY {p:name} ASC, {p:code} ASC
```

- **Getting every Product that is in at least 3 Categories**

The following FlexibleSearch statement returns every **Product** that is in more than three **Categories** (specified by the **WHERE howmany > 3** clause in **subselect 1**). The **subselect 2** returns the number of categories a **Product** is in (via searching the **CategoryProductRelation**). The **subselect 1** returns only those products which are in more than three **Categories** (**WHERE howmany >3**).

```
SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:PK} IN
 (
  -- subselect 1
```

```

SELECT prod
  FROM
  (
    {{
      -- subselect 2
      SELECT {cpr:target} AS prod, count({cpr:target}) AS howmany
        FROM {CategoryProductRelation AS cpr}
        GROUP BY {cpr:target}
    }}
  )
  temptable
 WHERE howmany > 3
)
ORDER BY {p:name} ASC, {p:code} ASC

```

i Note

Add a Parameter for Flexibility

By replacing the hard-coded 3 with a parameter, you could use the statement to find every **Product** that is in a specified number of categories, such as:

```

SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:PK} IN
(
  -- subselect 1
  SELECT prod
    FROM
    (
      {{
        -- subselect 2
        SELECT {cpr:target} AS prod, count({cpr:target}) AS howmany
          FROM {CategoryProductRelation AS cpr}
          GROUP BY {cpr:target}
      }}
    )
    temptable
 WHERE howmany > ?number
)
ORDER BY {p:name} ASC, {p:code} ASC

```

Please also refer to the [Subselect with Parameters](#) section below for additional information.

Subselect with Parameters

- **Getting all Products ordered on or after a certain date**

The following FlexibleSearch statement returns every **Product** that was ordered on or after a certain date. Via **subselect 2**, the FlexibleSearch statement retrieves every **Order** that was created on or after the value specified by the **?date** parameter. Of these search results, **subselect 1** retrieves the **OrderEntries** that belong to these **Orders**. The outermost **SELECT** statement gets the **Products** referred by the **OrderEntries**.

```

SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:PK} IN
(
  {{
    -- subselect 1
    SELECT DISTINCT {oe:product}
      FROM {OrderEntry AS oe}
      WHERE {oe:order} IN
      (
        {{
          -- subselect 2
          SELECT {o:PK}
            FROM {Order AS o}
            WHERE {o:date} >= ?date
        }}
      )
  })
  temptable
 WHERE howmany > 3
)
ORDER BY {p:name} ASC, {p:code} ASC

```

```

        }
    }
)
}

```

- **Getting every Product without a PriceRow in a specified Currency**

The following FlexibleSearch statement returns every **Product** that does not have a **PriceRow** assigned for the specified **Currency**. The subselect returns every **PriceRow** for the specified currency. This search result is then **negated** via the outer FlexibleSearch statement, which returns every **Product** that is not included in the search results that are returned by the subselect.

```

SELECT {p:PK}
  FROM {Product AS p}
 WHERE {p:PK} NOT IN
(
  {{
    -- subselect
    SELECT {pr:product}
      FROM {PriceRow AS pr}
     WHERE {pr:currency} = ?currency
  }}
)
ORDER BY {p:name} ASC, {p:code} ASC

```

- **Reporting query with subselect in FROM clause and SQL aggregate functions.**

This query calculates average **Order** value and average **Order** unit count within specified date range.

Result of this query is a pair of numeric values. The first one is the average **Order** value, and the second one is the average order unit count.

The term **unit count** of the order means the sum of quantities of the order entries. For example if an order consists of:

- 1 red T-shirt
- 1 blue T-shirt
- 2 yellow T-shirt

Then the order unit count for this order is 4.

```

SELECT AVG(torderentries.totprice), AVG(torderentries.totquantity)
  FROM (
  {{{
    SELECT SUM({totalPrice}) AS totprice, SUM({quantity}) AS totquantity FR
       WHERE {creationtime} >= ?startDate AND {creationtime} < ?endDate
  }}}
) AS torderentries

```

Combined SELECT Statements with UNION Operator

The following FlexibleSearch statement uses the **UNION** operator to retrieve the set of results for two **SELECT** statements:

```

SELECT x.PK
  FROM
(
{{SELECT {PK} as PK FROM {Chapter}
WHERE {Chapter.PUBLICATION} LIKE 6587084167216798848
}}}
UNION ALL
{{{
SELECT {PK} as PK FROM {Page}
WHERE {Page.PUBLICATION} LIKE 6587084167216798848
}}}) x

```

FlexibleSearch Tips and Tricks

You can use the FlexibleSearch to build advanced queries for reports, get information from collection attributes, use and format dates, or use conditional parts of the query.

Flexible Search and Collections

Let's say you have a subtype of `Order` that holds a collection of `VoucherCard` items. You wish to get all `VoucherCards` that are assigned to a particular order, and the prices assigned to the `VoucherCards`.

As long as the collection element type has a reference to the item that holds the collection, all is simple:

```
SELECT {vc.PK}, {vc.price}
FROM
{
    Order AS o JOIN VoucherCard AS vc
    ON {vc.order} = {o.pk}
}
WHERE {o.PK} = ?order
```

Things get more complicated if the reference between element and holder is missing. You can use some workaround based on the fact that a collection attribute is stored as a list of **PKs**:

```
SELECT {dm.code}, {pm.code}
FROM
{
    DeliveryMode AS dm JOIN PaymentMode AS pm
    ON {dm.supportedPaymentModeInternal} LIKE CONCAT( '%', CONCAT( {pm.PK} , '%' ) )
}
```

JOIN Clauses

In FlexibleSearch queries, only two kinds of **JOIN** clauses are available:

- **LEFT JOIN**
- **JOIN**

These clauses can be useful if you wish to select items that are connected to other items via a relation, for example:

```
SELECT {p:PK}, {c:code} FROM
{
    Product as p JOIN CategoryProductRelation as rel
    ON {p:PK} = {rel:target}
    JOIN Category AS c
    ON {rel:source} = {c:PK}
}
```

UNION Clauses

UNION clauses allow to unite the results of different queries. Here is a simple example of how to get all pages and chapters of a publication. `?pk` is a query parameter, for which a value must be specified at run time.

```
SELECT uniontable.PK, uniontable.CODE FROM
(
    {{ SELECT {c:PK} as PK, {c:code} AS CODE FROM {Chapter AS c}
```

```

        WHERE {c:PUBLICATION} LIKE ?pk
    }}
UNION ALL
{{{
    SELECT {p:PK} AS PK, {p:code} AS CODE FROM {Page AS p}
    WHERE {p:PUBLICATION} LIKE ?pk
}}}
) uniontable

```

Using Temporary Tables

When you use temporary tables, you need to use a slightly different syntax to retrieve values from a temporary table. Instead of the FlexibleSearch syntax (`INNERTABLE:PK`), you have to use the native SQL syntax (`INNERTABLE.PK`), such as:

```

SELECT INNERTABLE.PK, INNERTABLE.CatCode FROM
(
{{{
    SELECT {p:PK} AS PK, {c:code} AS CatCode FROM
    {
        Product as p JOIN CategoryProductRelation as rel
        ON {p:PK} = {rel:target}
        JOIN Category AS c
        ON {rel:source} = {c:PK}
    }
}}}
) INNERTABLE

```

Conditional CASE Statements

If you create a complex query and you want to return different results of an attribute based on some condition, try the `CASE` statement.

See an example of getting all categories names, codes, and number of super categories. If category does not have any subcategories mark it as root category, otherwise mark it as normal category.

```

SELECT {c:name[en]} AS Name, {c:code} AS Code,
(
CASE
    WHEN COUNT(DISTINCT{superCategory:PK}) <= 0
    THEN 'root category'
    ELSE 'normal category'
END
) as TYPE,
COUNT(DISTINCT{superCategory:PK}) AS SuperCategories
FROM
{
    Category as c LEFT JOIN CategoryCategoryRelation as rel
    ON {c:PK} = {rel:target}
    LEFT JOIN Category AS superCategory
    ON {rel:source} = {superCategory:PK}
}
GROUP BY {c:PK}, {c:code}, {c:name[en]}

```

NAME	CODE	TYPE	SUPERCATEGORIES
AMD	HW2120	normal category	2
AMD	HW2120	normal category	2
ATI	HW2320	normal category	1
ATI	HW2320	normal category	1

NAME	CODE	TYPE	SUPERCATEGORIES
Accessories	accessories	root category	0
Accessories	accessories	root category	0
Anti-Virus Software	antivirus	normal category	1
Apparel	apparel	root category	0
Apparel	apparel	root category	0

You can also put SELECT statements into CASE statements, as in this little more complicated query snippet below. The query basically counts super categories for each category and sub categories only for the root category. It may seem that such a query would never be used but it sometimes is the only solution for a report table filling query - a table that also is responsible to aggregate and sort results.

```

SELECT {c:name[en]} AS Name, {c:code} AS Code,
(
CASE
    WHEN COUNT(DISTINCT{superCategory:PK}) <= 0
    THEN 'root category'
    ELSE 'normal category'
END
) as TYPE,
(
CASE
    WHEN ( COUNT(DISTINCT{superCategory:PK}) <= 0 )
    THEN
    (
        {{
            SELECT COUNT({innerC:PK}) FROM
            {
                Category as innerC LEFT JOIN CategoryCategoryRelation as innerRel
                ON {innerC:PK} = {innerRel:target}
            }
            WHERE {innerRel:source} = {c:pk}
        }}
    )
    ELSE 0
END
) as RootSubCategories,
COUNT(DISTINCT{superCategory:PK}) AS SuperCategories
FROM
{
    Category as c LEFT JOIN CategoryCategoryRelation as rel
    ON {c:PK} = {rel:target}
    LEFT JOIN Category AS superCategory
    ON {rel:source} = {superCategory:PK}
}
GROUP BY {c:PK}, {c:code}, {c:name[en]}

```

NAME	CODE	TYPE	ROOTSUBCATEGORIES	SUPERCATEGORIES
Accessories	accessories	root category	0	0
Accessories	accessories	root category	0	0
Anti-Virus Software	antivirus	normal category	0	1
CPU	cpu	normal category	0	1
Content blocks	contentblocks	root category	3	0

NAME	CODE	TYPE	ROOTSUBCATEGORIES	SUPERCATEGORIES
Content blocks	contentblocks	root category	3	0
Digital photography	photography	normal category	0	1
Electronical Goods	electronics	root category	2	0
Hardware	hardware	normal category	0	1
Mainboards	boards	normal category	0	1
Memory	memory	normal category	0	1
Men's shoes	CL2100	normal category	0	1
Men's shoes	CL2100	normal category	0	1
Operating Systems	operating	normal category	0	1

Date Formatting

If you use dates in SQL, it is very likely that you would need to compare only parts of the date - for example, you would need to consider only months, dates, hours, but skip seconds and milliseconds. Nearly in every report you need to group your results by some date - months, hours, and so on. Here is how you do that on MySQL and Oracle.

Oracle

```
SELECT to_char({o:date}, 'mm/yyyy'), COUNT(DISTINCT{o:PK})
FROM {Order AS o}
GROUP BY to_char({o:date}, 'mm/yyyy')
```

MySQL

```
SELECT DATE_FORMAT({o:date}, '%M/%Y'), COUNT(DISTINCT{o:PK})
FROM {Order AS o}
GROUP BY DATE_FORMAT({o:date}, '%M/%Y')
```

String Formatting

In addition to date-specific operators, database system also offers String-specific operators.

MySQL Functions

Function	Description
ASCII()	Returns numeric value of left-most character.
BIN()	Returns a string representation of the argument.
BIT_LENGTH()	Returns length of argument in bits.
CHAR_LENGTH()	Returns number of characters in argument.
CHAR()	Returns the character for each integer passed.
CHARACTER_LENGTH()	A synonym for CHAR_LENGTH().

Function	Description
CONCAT_WS()	Returns concatenate with separator.
CONCAT()	Returns concatenated string.
ELT()	Returns string at index number.
EXPORT_SET()	Returns a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string.
FIELD()	Returns the index (position) of the first argument in the subsequent arguments.
FIND_IN_SET()	Returns the index position of the first argument within the second argument.
FORMAT()	Returns a number formatted to specified number of decimal places.
HEX()	Returns a hexadecimal representation of a decimal or string value.
INSERT()	Inserts a substring at the specified position up to the specified number of characters.
INSTR()	Returns the index of the first occurrence of substring.
LCASE()	Synonym for LOWER().
LEFT()	Returns the leftmost number of characters as specified.
LENGTH()	Returns the length of a string in bytes.
LIKE	Simple pattern matching.
LOAD_FILE()	Loads the named file.
LOCATE()	Returns the position of the first occurrence of substring.
LOWER()	Returns the argument in lowercase.
LPAD()	Returns the string argument, left-padded with the specified string.
LTRIM()	Removes leading spaces.
MAKE_SET()	Returns a set of comma-separated strings that have the corresponding bit in bits set.
MATCH	Performs full-text search.
MID()	Returns a substring starting from the specified position.
NOT LIKE	Negation of simple pattern matching.
NOT REGEXP	Negation of REGEXP.
OCTET_LENGTH()	A synonym for LENGTH().
ORD()	Returns character code for leftmost character of the argument.
POSITION()	A synonym for LOCATE().
QUOTE()	Escapes the argument for use in an SQL statement.

Function	Description
REGEXP	Pattern matching using regular expressions.
REPEAT()	Repeats a string the specified number of times.
REPLACE()	Replaces occurrences of a specified string.
REVERSE()	Reverses the characters in a string.
RIGHT()	Returns the specified rightmost number of characters.
RLIKE	Synonym for REGEXP.
RPAD()	Appends string the specified number of times.
RTRIM()	Removes trailing spaces.
SOUNDEX()	Returns a soundex string.
SOUNDS LIKE(v4.1.0)	Compares sounds.
SPACE()	Returns a string of the specified number of spaces.
STRCMP()	Compares two strings.
SUBSTR()	Returns the substring as specified.
SUBSTRING_INDEX()	Returns a substring from a string before the specified number of occurrences of the delimiter.
SUBSTRING()	Returns the substring as specified.
TRIM()	Removes leading and trailing spaces.
UCASE()	Synonym for UPPER().
UNHEX()(v4.1.2)	Converts each pair of hexadecimal digits to a character.
UPPER()	Converts to uppercase.

Oracle Functions

Function	Definition
ASCII	The ASCII function returns the decimal representation in the database character set of the first character of char.
CHR	The CHR function returns the character having the binary equivalent to n as a VARCHAR2 value in either the database character set.
COALESCE	The COALESCE function returns the first non-null expr in the expression list. At least one expr must not be the literal NULL. If all occurrences of expr evaluate to null, then the function returns null.
CONCAT	The CONCAT function returns the concatenation of 2 strings. You can also use the command for this.
CONVERT	The CONVERT function converts a string from one character set to another. The datatype of the returned value is VARCHAR2.

Function	Definition
DUMP	The DUMP function returns a VARCHAR2 value containing the datatype code, length in bytes, and internal representation of expr. The returned result is always in the database character set.
INSTR	Returns the position of a String within a String. For more information, see Oracle instr function.
INITCAP	Transform String to init cap.
INSTRB	Returns the position of a String within a String, expressed in bytes.
INSTRC	Returns the position of a String within a String, expressed in Unicode complete characters.
INSTR2	Returns the position of a String within a String, expressed in UCS2 code points.
INSTR4	Returns the position of a String within a String, expressed in UCS4 code points.
LENGTH	The LENGTH function returns the length of char. LENGTH calculates length using characters as defined by the input character set.
LENGTHB	Returns the length of a string, expressed in bytes.
LOWER	The LOWER function returns a string with all lower case characters.
LPAD	Adds characters to the left of a string until a fixed number is reached. If the last parameter is not specified, spaces are added to the left.
LTRIM	LTRIM removes characters from the left of a string if they are equal to the specified string. If the last parameter is not specified, spaces are removed from the left side.
REPLACE	The replace function replaces every occurrence of a search_string with a new string. If no new string is specified, all occurrences of the search_string are removed.
REVERSE	Reverses the characters of a String.
RPAD	Adds characters to the right of a string until a fixed number is reached. If the last parameter is not specified, spaces are added to the right.
RTRIM	RTRIM removes characters from the right of a string if they are equal to the specified string. If the last parameter is not specified, spaces are removed from the right side.
SOUNDEX	SOUNDEX returns a character string containing the phonetic representation of char. This function lets you compare words that are spelled differently, but sound alike in English.
SUBSTR	Returns a substring. For more information, see Oracle substring.
SUBSTRB	Returns a substring expressed in bytes instead of characters.

Function	Definition
SUBSTRC	Returns a substring expressed in Unicode code points instead of characters.
SUBSTR2	Returns a substring using USC2 code points.
SUBSTR4	Returns a substring using USC4 code points.
TRANSLATE	TRANSLATE returns expr with all occurrences of each character in from_string replaced by its corresponding character in to_string. Characters in expr that are not in from_string are not replaced.
TRIM	The TRIM function trims specified characters from the left and/or right. If no characters are specified, the left and right spaces are left out.
(pipes)	With pipes, you can concatenate strings.
UPPER	Transform a string to all upper case characters.
VSIZE	The VSIZE function returns the byte size of a String.

Boolean Parameters in Queries

Since not all databases recognize `true` as a query parameter, `0` and `1` should be used instead of `false` and `true`.

Query and JDBC Hints

A hint is an optimization directive that you can embed into an SQL statement to instruct the database on how to execute the query. Platform allows you to influence query execution by using hints.

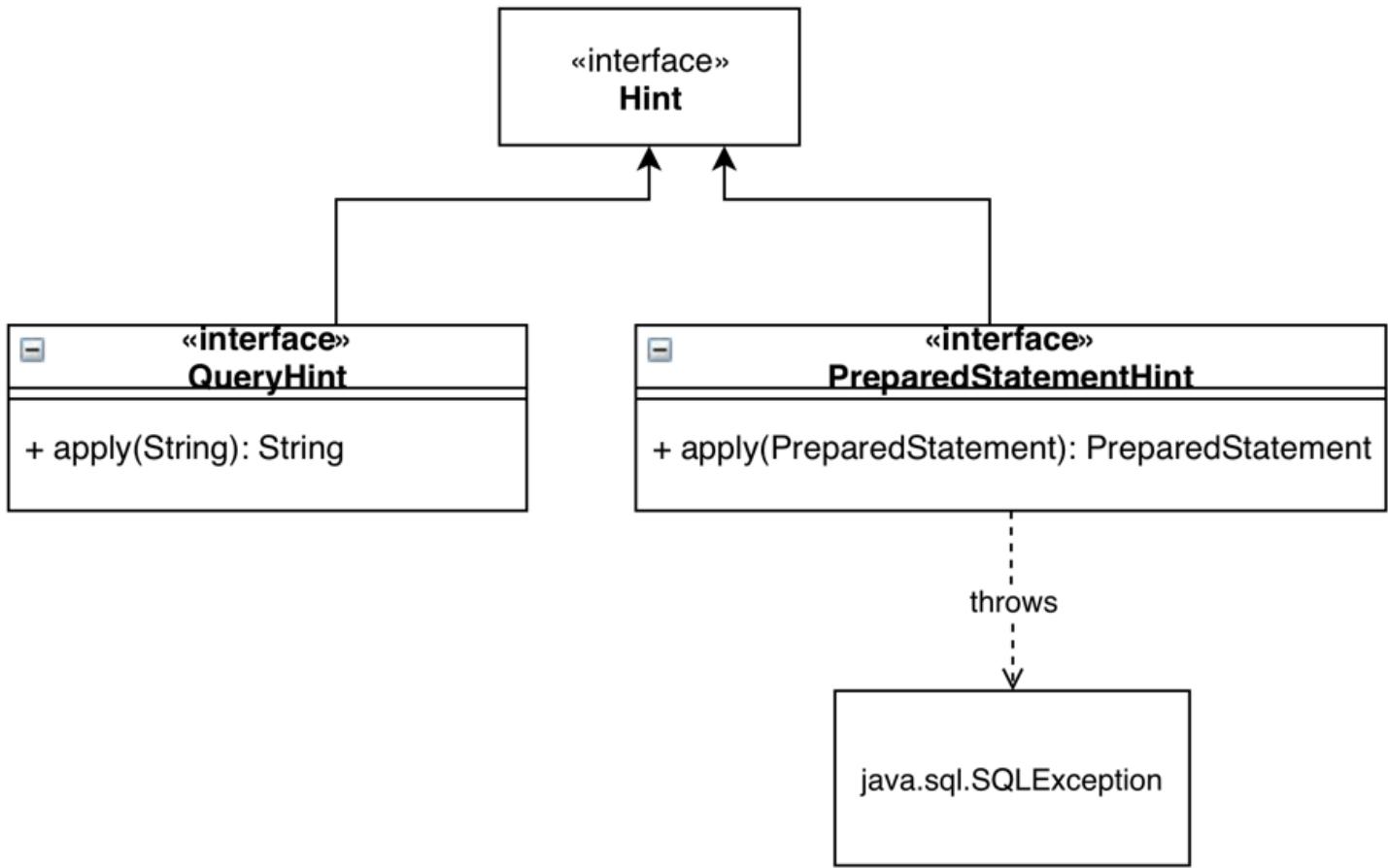
Some database engines allow you to set additional hints for an underlying optimizer that in certain circumstances may speed up query execution. In addition, the Java JDBC driver allows you to set some options on the `PreparedStatement` object that may also help in some situations.

i Note

Keep in mind that providing query hints is an advanced topic and requires deep knowledge of the database engine.

Architecture

Platform provides a set of interfaces that allow you to define both the `Query` and the `PreparedStatement` hints:



The top level Hint interface acts only as a marker interface. The two sub-interfaces are responsible for modifying a query String and the PreparedStatement object respectively. The main `FlexibleSearchQuery` object has two additional methods to pass Hint instances to the FlexibleSearch engine:

```

public void addHints(final Hint... hints)
public void addHints(final List<? extends Hint> hints)
  
```

QueryHint Implementation

Platform provides one implementation of the `QueryHint` interface for use in the SAP S/4HANA database. The `HanaHints` class acts as a factory builder for any hints SAP S/4HANA supports.

i Note

Please keep in mind that the `HanaHints` class is a generic implementation so it doesn't provide any methods that would refer to the existing hints by name. You must exactly know the hint names to use them.

`HanaHints` provides the `create(String... hints)` method that allows you to create a `HanaHints` object as follows:

```

// will add IGNORE_PLAN_CACHE to the query
HanaHints hints = HanaHints.create("IGNORE_PLAN_CACHE");
// will add IGNORE_PLAN_CACHE and USE_OLAP_PLAN to the query
HanaHints hints = HanaHints.create("IGNORE_PLAN_CACHE", "USE_OLAP_PLAN");
  
```

If you need to add more hints to the existing `HanaHints` object, you can use the `add(String hint)` method. This can be useful for an instance in iterations:

```

HanaHints hints = HanaHints.create("IGNORE_PLAN_CACHE");
moreHints.forEach(hints::add);
  
```

After building the `HanaHints` object, it is easy to add it to the `FlexibleSearchQuery` object:

```
FlexibleSearchQuery fQuery = new FlexibleSearchQuery("SELECT {PK} FROM {User}");
fQuery.addHints(hints);
```

After the query is translated into a real SQL query, the hints are compiled into the proper hint directive and added to the very end of the query in the following form:

```
WITH HINT(HINT1,HINT2,...)
```

So the example from above would look like this:

```
SELECT item_t0.PK FROM users item_t0 WHERE (item_t0.TypePkString IN (?,?,?,?,?)) WITH HINT(IGNORE)
```

i Note

`FlexibleSearchQuery` objects allow you to add more than one hint to the query. In case of HANA-specific hints, it doesn't make sense since `HanaHints` compiles each separate hint into one `WITH HINT` directive.

The `HanaHints` class works in a safe way so that hints are applied only when the SAP S/4HANA database is really used.

PreparedStatement Hints

Platform provides the `JdbcHints` class that allows you to create hints that operate on the underlying `PreparedStatement` object. Use the `JdbcHints#preparedStatementHints()` factory method to produce an instance of the `PreparedStatementHint` interface to gain access to the `PreparedStatement` object.

To change the fetch size, use:

```
PreparedStatementHint hints = JdbcHints.preparedStatementHints().withFetchSize(50);
```

To change the timeout, use:

```
PreparedStatementHint hints = JdbcHints.preparedStatementHints().withQueryTimeOut(200);
```

It is also possible to provide your own code by implementing the `PreparedStatementHint` interface and using the generic method:

```
PreparedStatementHint hints = JdbcHints.preparedStatementHints().withHint(yourPreparedStatementHint);
```

or even using a JDK8 inline function as follows:

```
PreparedStatementHint hints = JdbcHints.preparedStatementHints().withHint(ps -> {
    ps.setCursorName("fooBar");
    return ps;
});
```

you can also chain all calls:

```
PreparedStatementHint hints = JdbcHints.preparedStatementHints().withFetchSize(50).withQueryTimeOut(200)
    .setCursorName("fooBar")
```

```
        return ps;
});
```

Since `PreparedStatementHint` is an instance of the `Hint` interface, you can easily add it to the `FlexibleSearchQuery` object:

```
FlexibleSearchQuery fQuery = new FlexibleSearchQuery("SELECT {PK} FROM {User}");
fQuery.addHints(JdbcHints.preparedStatementHints().withQueryTimeOut(200));
```

i Note

All `PreparedStatementHint` objects are applied one by one in a chain on the same underlying `PreparedStatement` object

Polyglot Persistence Query Language

The polyglot persistence query language is a part of `FlexibleSearch`. You can use it to search for items inside an alternative storage, for example inside a document-type storage.

`FlexibleSearch` is used to search in a default, relational storage. The polyglot persistence query language is its counterpart for an alternative storage .

The polyglot query structure is a simplified version of the `FlexibleSearch` query structure. The `FlexibleSearch` `SELECT . . . FROM` is replaced with `GET`.

For example, to get:

- all `Title` instances, ordered by `code`, use:

```
GET {Title} ORDER BY {code}
```

- all `Title` instances with a given `code`, ordered by `code`, use:

```
GET {Title} WHERE {code}=?code ORDER BY {code}
```

Syntax Overview

A polyglot search query consists of the `GET` keyword followed by a type key and an expression:

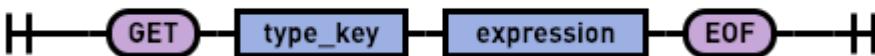


Figure: query

A type key is an item type identifier inside curly brackets, for example `{Title}`:



Figure: type_key

An identifier starts with a letter (a-Z), and then contains zero, or more of: letter (a-Z), digit (0-9), underscore ("_"), or the dot (".").

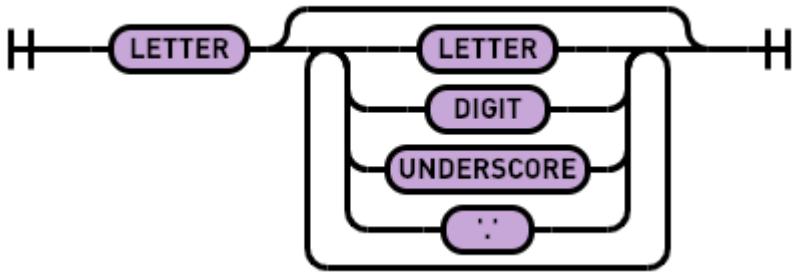


Figure: IDENTIFIER

An expression can start with the optional WHERE keyword.

WHERE can be followed by expr_or:

```
GET {Title} WHERE {code}=?code1 or {code}=?code2
```

WHERE can be also followed by optional order_by:

```
GET {Title} WHERE {code}=?code1 or {code}=?code2 ORDER BY {code}
```

An expression can contain order_by as the only element:

```
GET {Title} ORDER BY {code}
```

An expression can be empty - you don't have to include expressions in your queries at all:

```
GET {Title}
```

The diagram shows the described options:

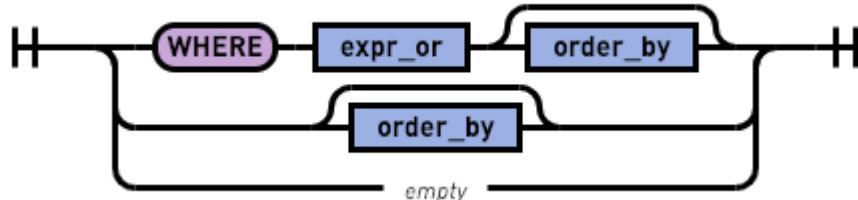


Figure: expression

order_by Structure

Here is an example of an order_by query, with search results ordered by a localized attribute:

```
GET {Product} WHERE {name}=?name ORDER BY {name[en]} ASC, {name[de]} DESC
```

The order_by element consists of the ORDER_BY keyword followed by one or more order_key elements separated by "," (a comma).



Figure: order_by

The order_key element consists of an attribute_key followed by the optional ORDER_DIRECTION element.

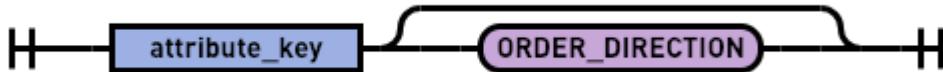


Figure: order_key

Here is an example of order_key - it consists of an attribute_key (includes a language identifier), and ORDER_DIRECTION:

```
{name[en]} ASC
```

attribute_key consists of the identifier element (described above) surrounded by curly brackets, with the optional localization identifier inside square brackets.



Figure: attribute_key

order_direction is: the ASC keyword for ascending, or the DESC keyword for a descending sort order.

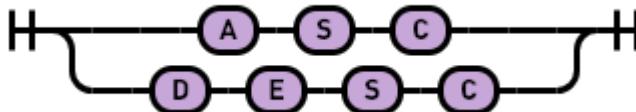


Figure: ORDER_DIRECTION

expr_or Structure

Here is an example of an expr_or query, with logical operations and sort order:

```
GET {Title} WHERE {code}=?code1 OR {code}=?code2 AND {code} IS NOT NULL ORDER BY {code[en]} DESC
```

To keep the operations in order, expr_or element consists of the expr_and element followed by the optional OR_OPERATOR and expr_and elements:



Figure: expr_or

This is the OR part of the query:

```
{code}=?code1 OR expr_and
```

The expr_and element consists of the expr_atom element followed by the optional AND_OPERATOR and expr_atom elements.



Figure: expr_and

This is the AND part of the query:

```
{code}=?code2 AND {code} IS NOT NULL
```

expr_atom consists of:

- attribute_key element followed by the CMP_OPERATOR element and '?' (question mark) followed by the IDENTIFIER element, or
- attribute_key element followed by NULL_OPERATOR, or
- expr_or element inside round brackets

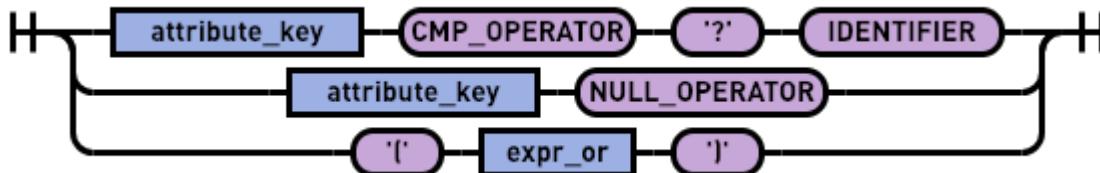


Figure: expr_atom

cmp_operator is one of the following: "=" (equal), "<>" (not equal), ">" (greater than), "<" (lower than), ">=" (greater or equal), "<=" (lower or equal):

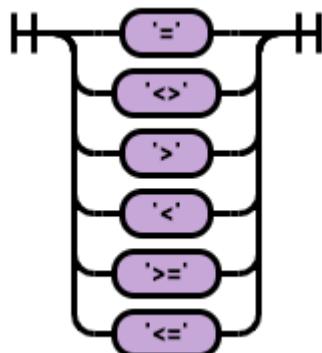


Figure: CMP_OPERATOR

null_operator consists of the IS NULL keywords divided by the optional NOT keyword:

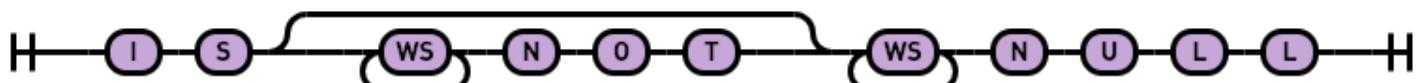


Figure: NULL_OPERATOR

ws is a white space (space, tabulator, new line, or carriage return) character:

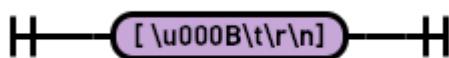


Figure: ws

Related Information

[Polyglot Persistence](#)

GenericSearch

While SAP FlexibleSearch offers a powerful search API to developers, some of them may prefer the GenericSearch API that is similar to Hibernate Criteria Queries. Hibernate is a collection of related projects, enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object and Relational Mapping.

Introduction

Both the **FlexibleSearchService** and the GenericSearch API allow developers to construct and execute queries against the SAP Commerce database, by focusing on SAP Commerce items rather than raw SQL. However, the way in which the queries are constructed is based on two completely different and complimentary approaches.

A **FlexibleSearchService** query is constructed by placing a FlexibleSearch query statement in a String, for example:

```
""SELECT {" + CategoryModel.PK + "} FROM {" + CategoryModel._TYPECODE + "} WHERE {" CategoryModel.]
```

The **?id** placeholder in the query is the place for a real condition parameter that is replaced during query execution.

In contrast to this, a query, for the **GenericSearchService** is constructed by combining instances of **GenericField** and **GenericCondition** to form **GenericQuery**. **GenericQuery** is a Java-based object describing the search criteria, an example of which is shown below. Users of Hibernate ORM can see a close parallel between this approach and the Hibernate's Criteria Queries.

```
final String categoryID = "Foo";
final GenericCondition idCondition = GenericCondition.equals(
    new GenericSearchField(CategoryModel._TYPECODE, CategoryModel.ID ), categoryID);
final GenericConditionList gl = GenericCondition.createConditionList(idCondition);
final GenericQuery query = new GenericQuery(CategoryModel._TYPECODE, gl);
```

To execute the FlexibleSearch, use the **FlexibleSearchService**:

```
Collection result = flexibleSearchService.<CategoryModel> search("SELECT
    {" + CategoryModel.PK + "} FROM {" + CategoryModel._TYPECODE + "}
    WHERE {" CategoryModel.ID "}).getResult();
```

Likewise, to execute the GenericSearch, use the **GenericSearchService**:

```
Collection result = genericSearchService.search(query).getResult();
```

This document describes the latter approach, of performing GenericSearch calls by constructing **GenericQuery** instances. See [FlexibleSearch](#) for a detailed discussion on the former. Additionally, see the [hybris Platform Search Mechanisms](#) document for an overview of all search mechanisms available in SAP Commerce.

GenericSearch

The GenericSearch is a search framework that allows you to easily search on items stored within the Platform. Its functionality may be a bit limited, but on the other hand, it is easy to understand and to handle. Although it technically uses the FlexibleSearch, it encapsulates the FlexibleSearch syntax and usage. The functionality of the GenericSearch includes:

- Searching of items as well as raw data fields
- Unlimited number of conditions
- Inner joins and outer joins between item types possible
- Unlimited number of **order by** clauses

- Subselects.

The GenericSearch is the tool of choice for a typical **Find me all products** sort of query that the real **FlexibleSearchService** would be an overkill to work with. If you do not have extensive experience with the Platform, it is recommended to get familiar with the GenericSearch at first and use it as a ladder to work yourself upwards to the FlexibleSearch.

Before an actual GenericSearch statement is executed, it is run through a syntax checker. In other words, an incorrect statement is never executed. On top of that, you may assign names to GenericSearch statements so you may save them in the database.

To run a GenericSearch statement, set up a query and then have the current session run that query. Technically, you create a **GenericQuery** object and add fields to it, that contain the parts of your query that you want. An example for that:

```
// create a query to find products
GenericQuery query = new GenericQuery(ProductModel._TYPECODE);
// run the search
genericSearchService.search(query).getResults();
```

First, it creates a query that looks for products. Then it runs that query. Since the query is not limited in any way, it will find all products within the platform. Now, refine the query:

```
// create a query to find products
final GenericQuery query = new GenericQuery(ProductModel._TYPECODE);

// create a new field as a container for the search results
// the field searches in ProductModel.NAME
// roughly translated into SQL terms: SELECT FROM ProductModel.Name
final GenericSearchField nameField = new GenericSearchField(ProductModel._TYPECODE, ProductModel.N

// add a search condition to the field created above
// corresponds about to the SQL statement:
// WHERE (nameField) IS LIKE 'display'
final GenericCondition condition = GenericCondition.createConditionForValueComparison(nameField, Or

// Add that search condition to query
query.addCondition(condition);

// order by name - ascendingly
query.addOrderByName(new GenericSearchOrderBy(nameField, true));

// run the search
genericSearchService.search(query).getResults();
```

The listing above searches for all products with names containing the string **display** and presents them in an ascendant order by their name. The next listing extends that even more:

```
// create a query to find products
final GenericQuery query = new GenericQuery(ProductModel._TYPECODE);

// create a new field as a container for the search results
// the field searches in ProductModel.NAME
// roughly translated into SQL terms: SELECT FROM ProductModel.Name
final GenericSearchField nameField = new GenericSearchField(ProductModel._TYPECODE, ProductModel.N

// add a search condition to the field created above
// corresponds about to the SQL statement:
// where (nameField) IS LIKE 'display'
final GenericCondition condition = GenericCondition.createConditionForValueComparison(nameField, Or

// Add that search condition to query
query.addCondition(condition);

// order by name - ascendingly
query.addOrderByName(new GenericSearchOrderBy(nameField, true));

// create a join with all medias within the platform
```

```

GenericCondition joinCondition = GenericCondition.createJoinCondition(productMediaField, mediaField);

// add the join to the previous search so that only entries that
// match both searches are returned

query.addInnerJoin(joinCondition);

// run the search
genericSearchService.search(query).getResults();

```

This listing returns all products with names containing the string **display** that have a media assigned to them. If you need to, you can also assign a parameter to the search. The following listing shows how to do it:

```

// create a query to find products
final GenericQuery query = new GenericQuery(ProductModel._TYPECODE);

// create a new field as a container for the search results
// the field searches in ProductModel.NAME
// roughly translated into SQL terms: SELECT FROM ProductModel.Name
final GenericCondition condition = GenericCondition.createConditionForValueComparison(nameField, Operator.EQUALS, "display");

// Add that search condition to query
query.addCondition(condition);

// order by name - ascendingly
query.addOrderByName(new GenericSearchOrderBy(nameField, true));

// create a join with all medias within the Platform
final GenericCondition joinCondition = GenericCondition.createJoinCondition(productMediaField, mediaField);

// add the join to the previous search so that only entries that match both searches are returned
query.addInnerJoin(joinCondition);

// run the search
genericSearchService.search(query).getResults();

// reset the value
query.getCondition().setResettableValue("myNameQualifier", "floppy"); // (2)

// find products with "floppy" in a name
genericSearchService.search(query).getResults();

```

The main difference is marked by (1) - it is the name of a parameter that you may hand over, for example in the URL. In the (2) this parameter is set to a new value, in this case String **floppy**. The search string to look for is then no longer **display**, but **floppy**. All other search parameters remain the same.

It is also possible to search for raw data instead of Item instances. To search for a code of a product, instead of for the **Product** itself, you should do the following:

```

// the same code as above
// get only a code of products
final GenericSelectField codeField = GenericSelectField(ProductModel._TYPECODE, ProductModel.CODE);
query.addSelectField(codeField);
// find codes of products with "display" in their name
genericSearchService.search(query).getResults();

```

GenericSearch in Action

The most helpful resource for learning and seeing the GenericSearch in action is to take a look at the JUnit test for it in. The code sample below contains a commented example taken from **GenericSearchTest.java** illustrating a simple GenericSearch:

GenericSearchTest.java

```

public void testGenericConditions() throws Exception{
    //Create a GenericCondition
    final GenericSearchField codeField = new GenericSearchField(ProductModel._TYPECODE, ProductMode
    GenericCondition condition = GenericCondition.createConditionForValueComparison(codeField, Ope
    // Add this to a new GenericConditionList
    final GenericConditionList conditionList = GenericCondition.createConditionList(condition);
    // Create another GenericCondition and append it to this GenericConditionList
    final GenericSearchField nameField = new GenericSearchField(ProductModel._TYPECODE, ProductMode
    conditionList.addToConditionList(GenericCondition.createConditionForValueComparison(nameField,
                                                "pRoduct", /* upper */true));
    // Create a GenericQuery
    GenericQuery query = new GenericQuery(ProductModel._TYPECODE);
    // Add the condition list to the GenericQuery
    query.addCondition(conditionList);
    // Also add two GenericSelectField to the GenericQuery
    final GenericSelectField codeSelectField = new GenericSelectField(ProductModel._TYPECODE, Prod
    final GenericSelectField nameSelectField = new GenericSelectField(ProductModel._TYPECODE, Prod
    query.addSelectField(codeSelectField);
    query.addSelectField(nameSelectField);
    // Run the query
    Collection<Object> result = genericSearchService.search(query, new StandardSearchContext(ct
    ...
}

```

GenericSearchQuery as Container for GenericQuery

GenericSearchQuery is a container object that allows to configure some additional search criteria and other options, like a user, for which the query is executed or pagination. The base **AbstractQuery** class keeps common functionality for both **GenericSearchQuery** and **FlexibleSearchQuery**. One of the **GenericSearchQuery** constructors provides a possibility to pass **GenericQuery** object as follows:

```

final GenericCondition idCondition = GenericCondition.equals(new GenericSearchField(CatalogModel._I
final GenericConditionList gl = GenericCondition.createConditionList(idCondition);
final GenericQuery query = new GenericQuery(CatalogModel._TYPECODE, gl);
final GenericSearchQuery gsq = new GenericSearchQuery(query);
gsq.setCount(5);
gsq.setUser(userService.getUserForUID("ahertz"));
gsq.setCatalogVersions(catalogVersionService.getCatalogVersion("clothes", "Online"));

final SearchResult<CatalogModel> searchResult = genericSearchService.search(gsq);

```

Behind the Scenes

Behind the scenes, the **GenericQuery** is ultimately converted into a **FlexibleSearchQuery** by the **toFlexibleSearch** method and than executed as FlexibleSearch query:

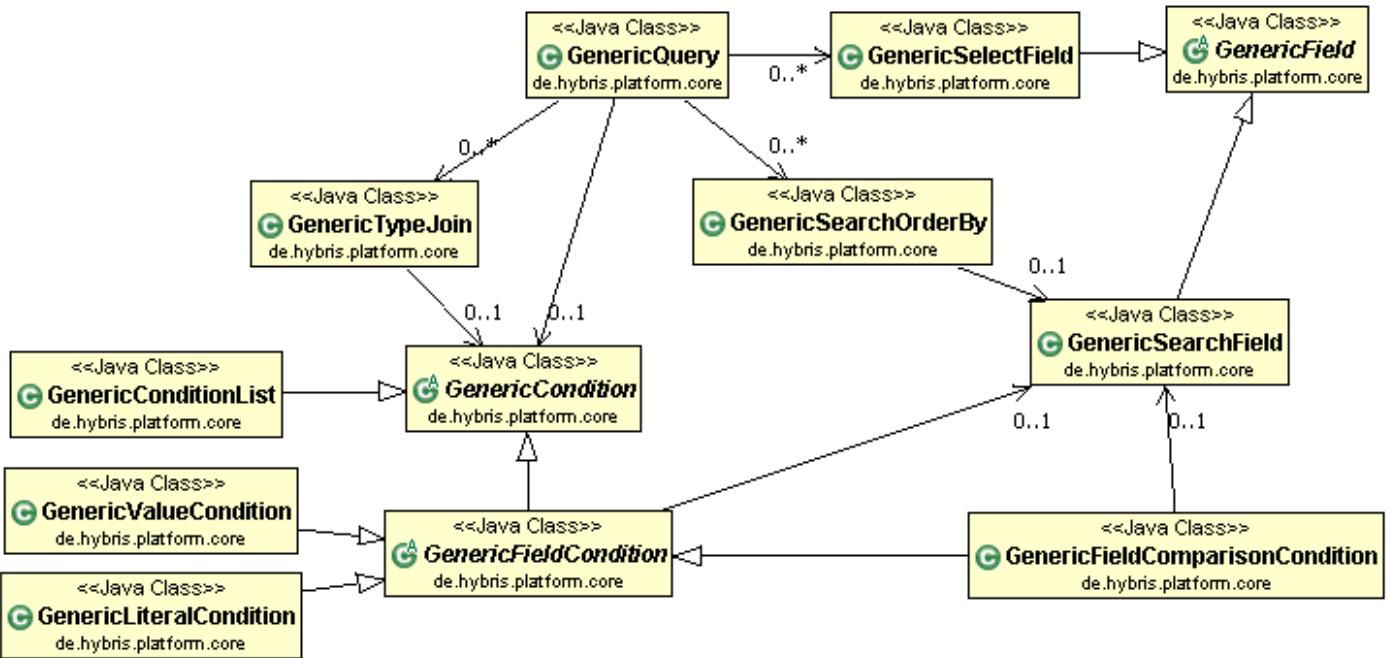
```

public String toFlexibleSearch(final Map<String, Object> valueMap){
    final StringBuilder sb = new StringBuilder();
    toFlexibleSearch(sb, null, valueMap);
    return sb.toString();
}

```

GenericSearch in UML

The UML diagram below illustrates the relationship between the main GenericSearch objects.



Related Information

[How to Implement a Custom Search Provider](#)

<http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/querycriteria.html> ↗

Access Rights

Permissions services for SAP Commerce provide a framework that you can use to define and implement your own access rights policy. By using permissions services, you can define specific access rights for users and the user groups. You can manage the access right for types, attributes, or you can assign them globally by defining access rights for the users without referring to any existing types, items, or attributes.

Further in the document, both users and user groups are also called Principals. Also, the term permissions has the same meaning as user access rights or access rights.

Introduction

Permission is an abstract concept, sometimes also referred to as user right. Permission services don't define what a permission is. This means that the instructions for permissions services and results from permissions services need to be further processed by your own application. The meaning of the permission is something that must be defined by implementing a security policy for a particular system. Such particular security policy covers many more topics than just permissions. It also includes, for example, data encryption, type of credentials used, security auditing, and so on.

Permissions services don't automatically restrict access to items or types, rather they provide a framework for your own application to implement the correct response that reflects your particular access right security policy.

Permissions support you in deciding what kind of access users or user groups can have to the objects. Groups can contain other groups or other users. The groups or users that belong to the higher-level group inherit the access rights from the superior group that has the permissions assigned.

By defining access rights - in other words, by assigning permissions, you can specify a very detailed level of access to the types, items, and their attributes for a particular user or user group. Complex grading of access rights for an individual user or user group can be represented through a hierarchical structure. This is due to the fact that the access rights can be inherited from the groups higher in the hierarchy by the more specific principals lower in the hierarchy.

Permissions system determines how the access rights inheritance is going down through the several hierarchy levels. This way you can assign access rights to a high-level general group from where they're propagated down the hierarchy through the lower-level principals. It also helps you in predicting what kind of effective access rights is granted to the particular user or user group and how to resolve possible conflicts in case of mutually excluding access rights inherited from more than one superior group.

About Permissions Scope

Generally, you can assign permissions to types, items, and attributes. Additionally, you can assign global permissions to users.

Global Permissions

Related to the user or user group and is meant to have the lowest priority. It means that the global permissions assigned to the particular user/group can be overwritten by assigning permissions to the specific item, type, or attribute. The global permissions apply in case there are no other permissions defined.

Type Permissions

Apply to the entire type's data. Access rights to the type can be granted in several stages, for example Read, Change, Create, and Delete. However, permissions framework lets you define and use your own structure of permissions.

Item Permissions

Can be assigned to the particular instance of the previously defined type. Simply speaking, if you can create many items of some type, this could allow you to override type-related access rights and assign permissions for a certain user to the particular item.

Attribute Permissions

Can be granted or denied explicitly for the individual attributes holding the informational content of a type or an item. This can be used for even more refined access control.

It's also important to notice that permission assignments can be either positive (allowed or granted) or negative (disallowed or denied).

About Permissions Stages

There are a few access rights - in other words, permissions stages mentioned in the previous section. These stages used as an example are: Read, Change, Create, and Delete. However, the permissions services aren't limited to those listed stages. The framework is able to handle any kind of permissions that you may have created and implemented into your own application. The names and the number of permissions stages only depend on your particular needs for having a customized access rights policy for your users. Permissions services provide you a framework ready to support you with this task.

Depending on your security policy, it's possible to build a front-end application that can enforce certain limitations in using the available access rights. For example, you may wish to create such application where it isn't possible to perform any action on an attribute that belongs to the type to which the user has been assigned deny permission. In such a case, if user has no access to the type, then it isn't possible to perform any action to its attributes.

Access rights, however, aren't enforced by default by any of the Platform services. As a result, it's up to the particular application that uses permissions services framework to define such restrictions. It's therefore also possible to build such application that doesn't have such limitations.

The choice between these two options depends only on the security policy that the application owner wants to apply for the principals.

Permissions Priority

By creating access right policy, it's important to think about priority and in which order the permissions are applied to the principals: users and user groups. The general idea is that the most general permission has the lowest priority while the most specific permission has the highest priority. For example, if the user has been granted type permission: Read to the type while

attribute permission: Read has been denied, then the user is able to see all the type attributes except the particular one that has been denied.

Another example would be a user that belongs to two different groups, one of which is granted a read permission to a type, while the other group is denied the same permission for the same type. In result of permissions inheritance the user inherits both permissions, however, the effective permission for the user is deny. Simply speaking, while on the same level in the inheritance hierarchy, deny settings rank higher in priority than grant settings.

i Note

Access rights work differently in the Platform and the Backoffice Framework. A key difference is that in the Backoffice Framework, attribute level permissions don't override type-level permissions, which is possible in Platform.

Global Permissions

Global Permissions are assigned to the particular user or user group - in other words, to principals. As they don't relate to any particular type, object, nor attribute, they provide the most general kind of access rights to the users. Any further and more specific permission assignment like type-, item-, or attribute permission overrides the global permissions regarding the particular type, item, or attribute.

Type Permissions

Type-related permissions define the user access rights to the type. The user with assigned permissions to the type also has default access to all type attributes. This is valid unless more specific permissions have been assigned. Type permissions can be assigned to grant or deny certain actions, for example: Read, Change, Create, and Delete. Type-level permissions should be a starting point in building security policy, since they're simple to define and cheap to evaluate (low-performance penalty for checking).

Attribute Permissions

Using attribute permissions allows you to explicitly assign permission to selected attributes of a type. Attribute-related permissions override type-related permissions. For example, if the type permission for a type is set to grant: <Change>, you can deny permission: <Change> to any of the attributes of that type.

Item Permissions

Using item permission assignments allows you to define very fine-grained access control in that you're able to grant or deny permissions to item instances. However, it's best to assign permission at a type level using principal groups as managing and checking permissions at item level can considerably degrade performance if a huge number of items is involved. On the other hand, such mechanism might be useful in many scenarios, such as overriding defaults or securing particularly important item instances.

i Note

Item permissions are based on permissions defined for catalog versions: even if two catalog-version-aware items are of the same type, they might have different item permissions if they belong to different catalog versions with different read or write restrictions.

Assigning Permissions

You can assign permissions - in other words, create access rights for a principal by:

- Using the Permissions Service API: You can use Permissions Service API for enriching your application with user access rights management features.
- Using ImpEx: For details, see [Legacy Permissions in ImpEx Scripts](#).

i Note

Using ImpEx scripts is now restricted only to legacy permissions: Read, Change, Create, and Delete. There's no general-purpose syntax for importing arbitrary permissions. You can overcome this limitation by providing BeanShell scripts in the ImpEx file that can directly call permissions services. However, this solution is neither elegant nor easy to write. Read the next section to investigate the legacy permissions that are compatible with the ImpEx scripts syntax.

Legacy Permissions in ImpEx Scripts

With the permissions services framework, you can design your own access rights names for your front-end application. However, due to the legacy dependency from the ImpEx scripting feature, it's worth to mention the following access right names: Read, Change, Create, Delete. These names were used to provide compatibility with the syntax of the ImpEx scripts used to import data to the SAP Commerce type system.

You can still use the same terminology, however, permissions services system doesn't depend on any particular front-end application and doesn't enforce any particular naming convention for your access rights. It depends entirely on you what kind of access rights names you create to use in your application.

The Read, Change, Create, Delete access rights names are clear and self-describing, and are used in this document as an example.

For more information on user rights in ImpEx, see [ImpEx API](#).

Read

This permission specifies whether the user account is granted access to the respective type. The type's attributes are displayed, but no actions are possible at this access level. If this permission is set to: deny, then the user account cannot:

- See instances of this type
- Read the values for the type's attributes

The type permission: Read applies to all attributes of the type unless explicitly overridden.

Change

This permission specifies whether the user account is allowed change the value of the type - in other words, modify all attributes of the type. The Change permission is different from the Create and Delete. Being assigned the Change permission doesn't allow users to create or delete instances of a type, such as new products or delete them.

You can also deny a user the type-related Change access right. The permission check result provides you with a feedback information that you can use to forbid users the access to the particular type.

The type-related Change permission applies to all attributes of the type unless explicitly overridden.

Create

The create permission assignment to the type specifies whether the user is allowed to create instances of the type, for example, creating individual products, customers, or system languages. A user account with granted Create access right but denied Change access right can create instances of the type, but can't set any of the attributes of the type. In result, the user account can only

create type instances with no attribute values set at all. By consequence, there's usually no reason to assign Create permission to the user but to deny the Change access right at the same time.

Delete

The delete permission specifies whether the user has the right to delete items of the type. For example, using this permission you can decide if deleting a product of a certain type is allowed or not. In some cases, type permission Delete may not suffice to delete SAP Commerce items of that type. Deleting may not be possible due to, for example:

- Other required access rights not being granted
- The user account not having sufficient overall rights
- Restrictions applying to the user account.

For example, to delete a product from the catalog user account needs the Delete access right and Change access right for catalog version.

Change_perm

The change_perm permission specifies whether the user has the right to grant permissions to other users.

Changing Permission Assignment During Runtime Operation

Changing permissions assignment for users and user groups takes immediate effect internally. If your application assigns permissions to a user, then the permissions services immediately return feedback information to you, ready to be handled by your application's user rights management system. As a result, even those users who are logged in into your application by the time of changes are affected and if a user is logged in while the user's access rights are changed, then the changed access rights are immediately effective.

However, if your application has, for example, browser-based user interface, then part of the pages may actually be kept in the browser memory (browser cache). If the user opens such a page before the related permissions have been assigned, then the state of the page may not change automatically, even if the new permissions assignment is already in effect. The similar effect is current for any web-based access interface to the application. After the permissions have been changed some elements may no longer be editable, but it's first clear to the users after they make an attempt to modify attribute values for an item or type through web interface. Usually, refreshing the application to let it apply newly assigned permissions solves this uncertainty.

Generally, it's advisable to avoid confusion and not to modify permission assignments when users are logged in. If you need to change user access rights when SAP Commerce is in live operation, use a scheduled maintenance time window to assign new permissions.

Effective Access Rights for a User Account in a User Group

Users can belong to user groups. If a user group is assigned permissions, then all the users belonging to that group inherit the same permissions. However, it may also happen that the user belongs to several groups having several, sometimes mutually exclusive, permissions. In this case, users inherit these permissions, but the mechanism is in place that tries to resolve possible conflicts and provide the proper information about the effective permissions. Effective permissions aren't those that are assigned but those that apply to the specific user or user group based on the permissions assigned directly and inherited from the groups located higher in the hierarchy.

It's a common approach that the permissions aren't assigned to user accounts directly, but to the user group instead. All members of that user group, both users or other user groups, automatically inherit those access rights. This way access rights can be inherited within a complete user group hierarchy. Subgroups and individual user accounts in such a hierarchy may override their inherited permissions assignments, expanding or restricting the range of effective access rights. Generally, if we consider the

permissions of a user group, the permissions assigned to the groups closest and upwards in the hierarchy will be the ones that are valid for that point.

The final result depends also on what kind of permissions are assigned to the groups higher in the hierarchy. If, for example, access is explicitly granted by one user group and explicitly denied by another user group on the same hierarchy level, then the access is denied for:

- Subgroup inheriting those permissions,
- User that belongs to both groups.

i Note

There are two exceptions in the system of access rights:

- User accounts that belong to the admingroup user group.

This also applies to user accounts that are inherited members of admingroup, that is, members of a subgroup of admingroup.

- User accounts with uid: admin.

A user account that is a member of the admingroup user group or users with uid: admin always have full access rights.

Read more in [Restrictions](#).

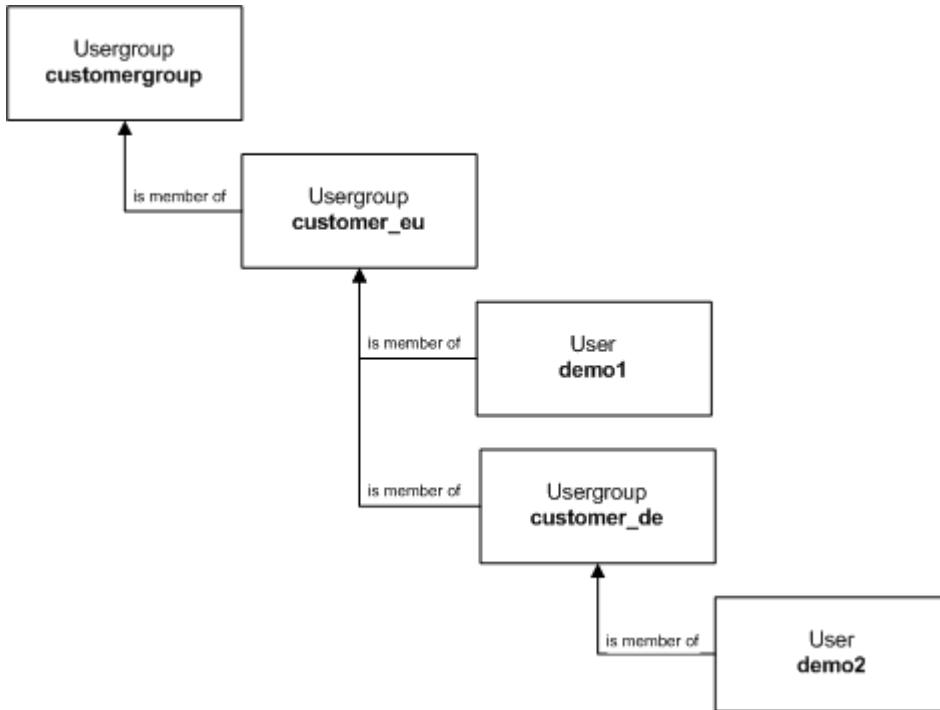
User Account as Member of Vertical User Group Hierarchy

The effective permissions for a particular user are those permissions that are assigned to the user group closest to that user account within a user groups hierarchy:

- If permissions are assigned to an individual user account, then those permissions are considered as closest ones to the user account in the inheritance hierarchy, and therefore are effective.
- If the user group has no permissions assigned, then the permissions assigned to the closest user group one level up in the user group hierarchy are valid.
- If no permission has been assigned throughout the user group hierarchy, then access right defaults to deny.

Example

The user group customergroup contains the user group customer_eu. The customer_eu usergroup contains the User demo1 and the usergroup customer_de. The customer_de usergroup contains the User demo2.



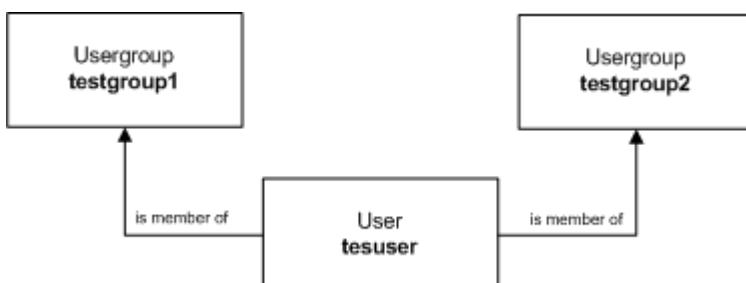
Permission Setting for customergroup	Permission Setting for customer_eu	Permission Setting for customer_de	Effective Access Right for demo1	Effective Access Right for demo2
granted	(no setting)	(no setting)	granted	granted
denied	(no setting)	(no setting)	denied	denied
granted	(no setting)	denied	granted	denied
denied	granted	(no setting)	granted	granted
denied	granted	denied	granted	denied
(no setting)	granted	denied	granted	denied
(no setting)	(no setting)	granted	denied (default)	granted

User Account as Member of Several Individual User Groups

If user account is a member of several user groups not forming a hierarchy, then all permissions assigned to these user groups apply by inheritance to the user account. Again, deny setting for access rights rank higher than grant setting for access rights. It means that if the same access right (for example: read, change) is defined in a conflicting manner by parent user groups being on the same hierarchy level, then the deny setting rank higher than grant setting.

Example

The user tesuser is a member of both testgroup1 and testgroup2.



Permission Setting for testgroup1	Permission Setting for testgroup2	Permission Setting for testuser	Effective Access Right for testuser
granted	(no setting)	(no setting)	granted
denied	(no setting)	(no setting)	denied
granted	(no setting)	denied	denied
denied	granted	(no setting)	denied
denied	granted	denied	denied
(no setting)	granted	denied	denied
(no setting)	(no setting)	(no setting)	denied (default)

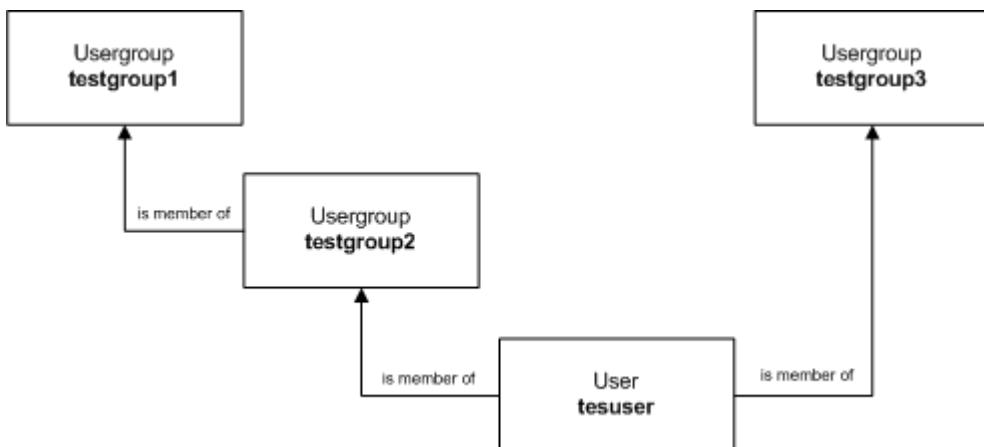
User Account as Member of User Groups Which Form More Than One Hierarchy

In this section, we have a combination of the two following situations:

- User account belongs to several user groups
- Some of the user groups form a hierarchy

Example

The example shows a user account as a member of two user groups, one of which is part of a user group hierarchy. Deny access rights rank higher than grant access rights. If the same access right is defined in a conflicting manner by user groups that are on the same hierarchy level, then the deny access rights rank higher than grant access rights.



Permission Setting for testgroup1	Permission Setting for testgroup2	Permission Setting for testgroup3	Permission Setting for testuser	Effective Access Right for testuser
granted	(no setting)	(no setting)	(no setting)	granted
denied	(no setting)	(no setting)	(no setting)	denied
granted	(no setting)	denied	(no setting)	denied
denied	granted	(no setting)	(no setting)	granted
denied	granted	denied	(no setting)	denied
(no setting)	granted	denied	granted	granted

Permission Setting for testgroup1	Permission Setting for testgroup2	Permission Setting for testgroup3	Permission Setting for testuser	Effective Access Right for testuser
(no setting)	(no setting)	granted	(no setting)	granted
(no setting)	(no setting)	(no setting)	(no setting)	denied (default)

i Note

The inheritance mechanism considers the closest inheritance levels.

Related Information

[Permissions in The Backoffice Framework](#)

Managing and Checking Access Rights

You can use the permission management service to manage access rights in SAP Commerce. Additionally, the permission checking service allows you to check the existing permission assignments on a particular user or user groups.

Managing and Checking Access Rights

The permissions services is a framework that can be used to implement access rights system on the SAP Commerce Platform. By using the permissions services framework you can manage and check the access rights for users and user groups. You can assign, modify or remove permissions related to your users like employees and customers. You can also manage and check access rights assigned for the user groups. The framework has separate services for managing the access rights and checking what kind of access rights are assigned.

For example, you may assign the different access rights for your employees who manage SAP Commerce and for the customers who visit your web shop. The employees who administer SAP Commerce need to assign, modify, or remove permissions. For this task, use the **PermissionManagementService**. On the other hand, customers who visit your web shop do not need to manage permissions and there is no need to give them access to the **PermissionManagementService**. However, you may need to use the service that checks what the customer can see in your web shop. This is the task for **PermissionCheckingService**.

Access Rights should correspond to your security requirements, data model and organizational structure, and must be defined during design of your application, as a part of a security policy.

In other words, you need to design and implement access rights system used by your application.

Permission Services

Permission services described in this document focus on the permission assignment, which in most cases is a relation between:

- Permission: Distinguished by its name
- Principal: User or user group
- Object: Item, type or an attribute to which the permission is assigned

The only exception are the global permissions, that are only assigned for the users. They should apply in those situations when no other, more specific access rights, are assigned to the items, types or attributes.

The usual way to use permissions in the system is:

- Assigning permission to user groups
- Checking permissions for single users

However, permission services allow also to assign permissions for the single users and check permissions for the user groups, if needed. This is because the methods in permission services take a `PrincipalModel` argument, which can be either a user or usergroup.

PermissionManagementService

Responsible for managing permissions and permission assignments. Possible operations supported by this service are:

- Creating new permissions
- Listing available permissions
- Assigning permissions
- Replacing permissions
- Removing permissions

PermissionCheckingService

`PermissionCheckingService` is used to check the effective permission assignments, in other words: effective user access rights.

PermissionCRUDService

This is a convenience service that is a wrapper over `PermissionCheckingService` and provides checking operations for the four basic platform-defined permissions: Create, Read, Update, Delete (since CRUD).

Other

- `PermissionCheckResult`: Provides information whether the permission is granted or denied
- `PermissionCheckValue`: Simple enumeration of defined permissions used by other services
- `PermissionsConstants`: Contains a list of pre-defined permissions
- `PermissionAssignment`: Object used by `PermissionManagementService` to return the information about the permission for the particular user or users group.

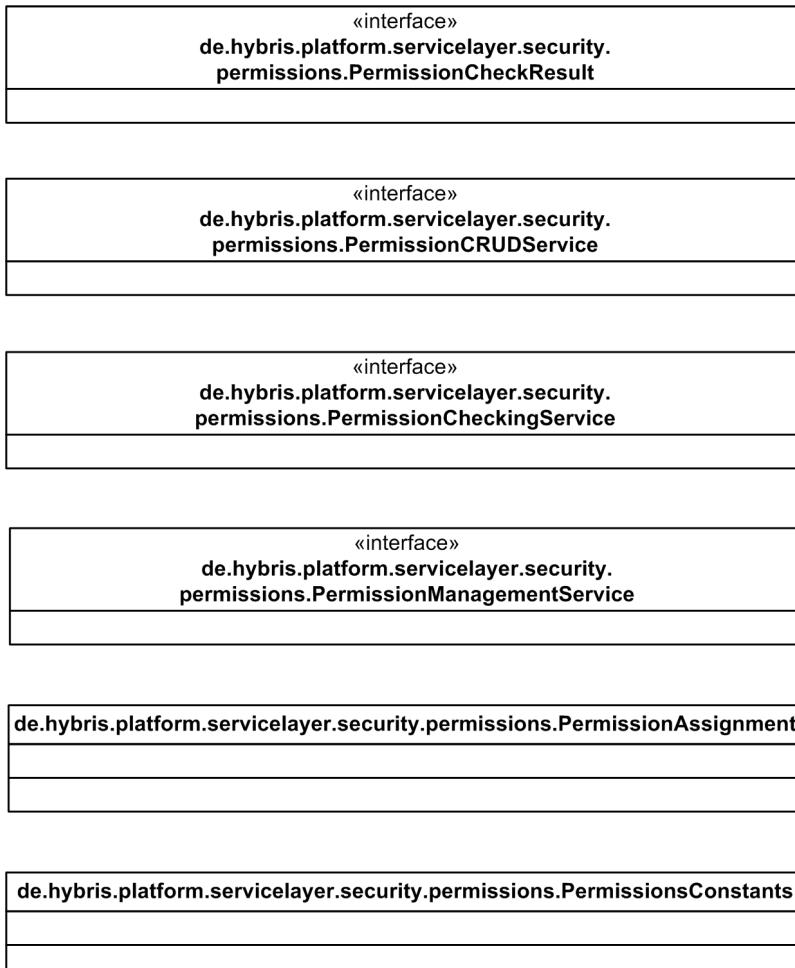


Figure: Permissions Services.

PermissionManagementService

PermissionManagementService is used to define and manage permission assignments. There are two basic operations for manipulating permissions without doing any assignments:

- Create a new permission
- ```
void createPermission(String permissionName);
```
- Return a collection containing names of all defined permissions:
- ```
Collection<String> getDefinedPermissions();
```

There are no means to delete the permission, once it is created. Such an operation would require checking the entire database to ensure that the permission is not used anywhere. This could also lead to a hole in the security system. Since permissions are only distinguished by name, removing a permission and then adding another one with the same name makes the existing security audit logs useless.

Creating Permission Assignments

To create the permission assignment, use **PermissionAssignment** object, which provides information whether the permission is granted for a principal, or denied. The principal is usually a usergroup, but it can be also a user. The following code snippet provides an example on how to grant permission A_PERMISSION to an item aCountry for a group aGroup.

```
//create a permission
permissionManagementService.createPermission("A_PERMISSION");
```

```
<!-- ... -->

final CountryModel aCountry = ...
final PrincipalModel aGroup = ...

//create a permission assignment object which grants A_PERMISSION for aGroup
final PermissionAssignment permissionAssignment = new PermissionAssignment("A_PERMISSION", aGroup);

//create permission assignment to aCountry.
permissionManagementService.addItemPermission(aCountry, permissionAssignment);
```

The next example shows how to deny A_PERMISSION to an item aCountry for an user anUser. The second constructor of **PermissionAssignment** class is used here to take explicit value for denied flag.

```
//create a permission
permissionManagementService.createPermission("A_PERMISSION");

<!-- ... -->

final CountryModel aCountry = ...
final PrincipalModel anUser = ...

//create a permission assignment object which denies A_PERMISSION for anUser
final PermissionAssignment permissionAssignment = new PermissionAssignment("A_PERMISSION", anUser, true);

//create permission assignment to aCountry.
permissionManagementService.addItemPermission(aCountry, permissionAssignment);
```

There is also a **addItemPermissions()** method that enables to add multiple permission assignments to an item in one invocation.

In the same way permission can be assigned to:

- Attribute descriptors: By using **addAttributePermission()** or **addAttributePermissions()** methods
- Item types: By using **addTypePermission()** or **addTypePermissions()** methods

Global permission assignments are even simpler. To create a global permission there is no need for any object. In the code sample, note **PermissionAssignment** constructor that takes explicit value of denied flag as false which means that permission is granted.

```
//create a permission
permissionManagementService.createPermission("A_PERMISSION");

<!-- ... -->

final PrincipalModel aGroup = ...
//create a permission assignment object which grants A_PERMISSION for aGroup
final PermissionAssignment permissionAssignment = new PermissionAssignment(TEST_PERMISSION_1, aGroup, false);

//assign the permission.
permissionManagementService.addGlobalPermission(permissionAssignment);
```

Retrieving and Removing Permission Assignments

The methods for getting and removing permission also use **PermissionAssignment** object. When retrieving this object provides the information about existing permission assignments. When removing, this object provides information which assignment is to be removed.

For details about the available methods, see API documentation for **PermissionManagementService** class.

PermissionCheckingService

Permission checking service provides the answer to questions of the following form:

- Is a permission to an object granted for a principal?
- Is a permission granted for a principal globally?

The answers is returned for:

- Given permission
- Principal
- Object
- Item
 - Item type
 - Attribute descriptor

The answer is calculated out of the existing permission assignments created with the `PermissionManagementService` using permission checking rules. This way the answer provides information about actual permission assignments, that means, the permissions that were directly assigned.

The combination of actual permission assignments and permission checking rules produce effective permission assignments. Effective permission assignments might not physically exist in the system, but are the effect of using checking rules. For example, if there is an actual permission assignment to a user group, then every member of the group has that permission effectively assigned. This happens because user group permissions are inherited by users who belong to the group. Permission checking rules take into account permission inheritance within principal group hierarchy and inheritance within type and attribute descriptor hierarchy. The rules also prioritize different kind of permission assignments. For example, global assignments have the lowest priority. To see more detailed description, check the API documentation for `PermissionCheckingService`.

The result of checking permission assignment can be one of the following constants defined by `PermissionCheckValue`:

ALLOWED

Permission is granted for a principal

DENIED

Permission is explicitly denied for a principal

NOT_DEFINED

No actual permission assignment was found for the given permission checking operation

CONFLICTING

Situation when there are both DENIED and ALLOWED equal-priority assignments for a principal

The methods of `PermissionCheckingService` do not directly return `PermissionCheckValue` contants. When checking, it is much more convenient to get a single boolean yes or no answer. For that purpose a `PermissionCheckResult` interface is used as a result type. This interface gives acces to the raw checking results represented by a `PermissionCheckValue` enumeration. However, it also provides two convenience methods: `isGranted()` and `isDenied()`, which map `PermissionCheckValue` constants onto a boolean value. This mapping can be changed by using different `PermissionCheckValueMappingStrategy`. By default this strategy assumes that:

- Only ALLOWED maps to true
- Other values map to false

Use Cases

This section contains examples on checking permission for some most expected use cases. To fully understand these examples, check API documentation for `PermissionCheckingService` to find more details about permission checking rules.

Assume that:

- There is a permission `READ_CATALOG` defined
- User `user1` is a member of group1
- The following objects are defined:

```
CatalogModel catalog1 = ...
CatalogModel catalog2 = ...
PrincipalModel user1 = ...
PrincipalModel group1 = ...
```

Checking Item Permissions within Principal Group Hierarchy

Code snippet shows how permissions to an item within principal group hierarchy can be checked.

```
boolean result;

//The actual result is NOT_DEFINED, since no permission assignments has been done so far.
result = permissionCheckingService.checkItemPermission(catalog1, user1, "READ_CATALOG").isGranted()
result = permissionCheckingService.checkItemPermission(catalog2, user1, "READ_CATALOG").isGranted()

/*
 * Let's deny "READ_CATALOG" globally for group1
 */
permissionManagementService.addGlobalPermission(new PermissionAssignment("READ_CATALOG", group1, true))

//The actual result is now DENIED because of global "READ_CATALOG" denial for group1 and principal
result = permissionCheckingService.checkItemPermission(catalog1, user1, "READ_CATALOG").isGranted()
result = permissionCheckingService.checkItemPermission(catalog2, user1, "READ_CATALOG").isGranted()

/*
 * Let's grant "READ_CATALOG" for group1 to catalog1
 */
permissionManagementService.addItemPermission(catalog1, new PermissionAssignment("READ_CATALOG", group1, true))

//The result is now ALLOWED, although there is global denial for group1. This is because global ass
//have lower priority.
result = permissionCheckingService.checkItemPermission(catalog1, user1, "READ_CATALOG").isGranted()

//The result is still DENIED, because no permission assignment has been done to catalog2 to overrid
// "READ_CATALOG" denial for group1.
result = permissionCheckingService.checkItemPermission(catalog2, user1, "READ_CATALOG").isGranted()
```

Assigning Type Permissions

Building on the example in the previous section, permission assignments to the type can be used as shown in the code sample below.

```
ComposedTypeModel catalogType = ...

permissionManagementService.addTypePermission(catalogType, new PermissionAssignment("READ_CATALOG", true))

//The result is still ALLOWED, because of permission assignment to catalog1. Type permission assignm
// of lower priority,
// so it does not change anything.
result = permissionCheckingService.checkItemPermission(catalog1, user1, "READ_CATALOG").isGranted()

//The result is now ALLOWED, because permission assignment to catalog type overrides global "READ_
//denial for group1.
result = permissionCheckingService.checkItemPermission(catalog2, user1, "READ_CATALOG").isGranted()
```

For more sophisticated examples, check the `PermissionCheckingServiceTest`, where the permission checking rules are tested for various scenarios.

Best Practices and Tips

- Permissions are not automatically enforced. This means that your code must invoke a method or two from `PermissionCheckingService` and then act accordingly in order to have a permission-based security.
- By default, everything is denied.
- Assign permissions for the user groups. Every member of the group inherits the permission assignments. This works across the group hierarchy.
- Assign permissions to the types. Every item of that type will inherit this permission assignments. This works across type hierarchy.
- Use attribute descriptor assignments for fine-grained control of the attribute values.
- Use global permission assignments as a fall-back or default assignment. The global permissions have the lowest priority and are always overridden by the item, type, or attribute permissions assignments.
- Design your permissions schema and keep it simple.

Known Limitations

- You cannot search for the effective permission assignments using Flexible Search like: return all items that have a permission XYZ granted for principal P. The reason is that the effective permission assignments are calculated, and there is no easy way for the database server to perform such calculations when retrieving item rows from the database. This also means that permissions cannot be used for filtering rows in the data access layer.
- Since permission assignments are currently not represented by models, you cannot directly import them using the ImpEx scripts. The only way is to use `PermissionManagementService` from ImpEx using scripting.

Related Information

[Users in Platform](#)

Cross-Origin Resource Sharing Support

SAP Commerce supports the Cross-Origin Resource Sharing mechanism. The CORS mechanism defines a way for a browser and a server to decide which cross-origin requests for restricted resources can or cannot be allowed.

Imagine a script on `http://www.example.com/somepage.html` wanting to access resources from `https://www.example.com:87/resources`. Your browser considers these two addresses as two different origins and it will prevent the `/somepage.html` script from fetching the resources from `/resources`. It is possible to safely relax this limitation by preparing CORS configurations in such a way that the `/somepage.html` request will be accepted.

Enabling CORS Support in SAP Commerce Extensions

To enable CORS support in SAP Commerce extensions, include these sections in the `[extname]-web-spring.xml` files of chosen extensions.

`extname-web-spring.xml`

```
<bean id="extnamePlatformFilterChain" class="de.hybris.platform.servicelayer.web.PlatformFilterCha:
<constructor-arg>
<list>
<ref bean="corsFilter"/>
```

```
[...]
</list>
</constructor-arg>
</bean>
```

Configuring CORS in SAP Commerce Web Applications

Global Cluster Configuration

The CORS configuration is stored in the database inside `CorsConfigurationProperty` items. It is global and applies to all nodes connected to a cluster.

To edit your configuration, in Backoffice go to **System > CORS Filter > CorsConfigurationProperty** and create or modify instances of the `CorsConfigurationProperty` type.

Create New CorsConfigurationProperty

ESSENTIAL

Web Application: corsnode1

Key: allowedMethods

Value: POST GET PUT DELETE

CANCEL DONE

As a context use the name of your extension - we use `webAppName` in this document as an example. As a key, use one of the available properties.

CORS Header	CorsConfigurationProperty	Default Value	Description
Access-Control-Allow-Origin	allowedOrigins	null	List of hosts that can receive CORS response from your extension.
Access-Control-Allow-Methods	allowedMethods	GET HEAD	List of the supported HTTP methods.
Access-Control-Allow-Headers	allowedHeaders	null	The names of the supported author request headers.
Access-Control-Expose-Headers	exposedHeaders	null	List of the response headers other than simple response headers that the browser should expose to the author of the cross-domain request.
Access-Control-Allow-Credentials	allowCredentials	null	Indicates whether the user credentials, such as cookies, HTTP authentication, or client-side certificates, are supported.

CORS Header	CorsConfigurationProperty	Default Value	Description
Access-Control-Max-Age	maxAge	null	Indicates how long the results of a preflight request can be cached by the web browser, in seconds. If -1 - unspecified.
Access-Control-Allow-Patterns	allowedOriginPatterns	false	Alternative to allowedOrigins that supports more flexible origins patterns with "*" anywhere in the host name in addition to port lists.

i Note

It is not allowed to use the combination of the `allowCredentials=true` and `allowedOrigins=*` attributes. Use `allowedOriginPatterns` instead.

Use XSS filter to set the header

Add `Access-Control-Allow-Origin` in the response header as,

```
xss.filter.header.Access-Control-Allow-Origin=<#domain name#>
```

You must replace the domain name with the actual domain. Enter * to allow access from all domains.

Local Node Configuration

To configure CORS in a SAP Commerce extension, use these properties:

project.properties

```
corsfilter.webAppName.allowedOrigins=http://localhost:8080
corsfilter.webAppName.allowedMethods=GET PUT POST DELETE
```

⚠ Caution

Properties from the database take precedence before local properties. If a property with the same context and key is configured both in the database and the `properties` file, the value is taken from the database.

Default Configuration

When no CORS properties are defined, all CORS requests are rejected.

HTTPS Traffic Certificate

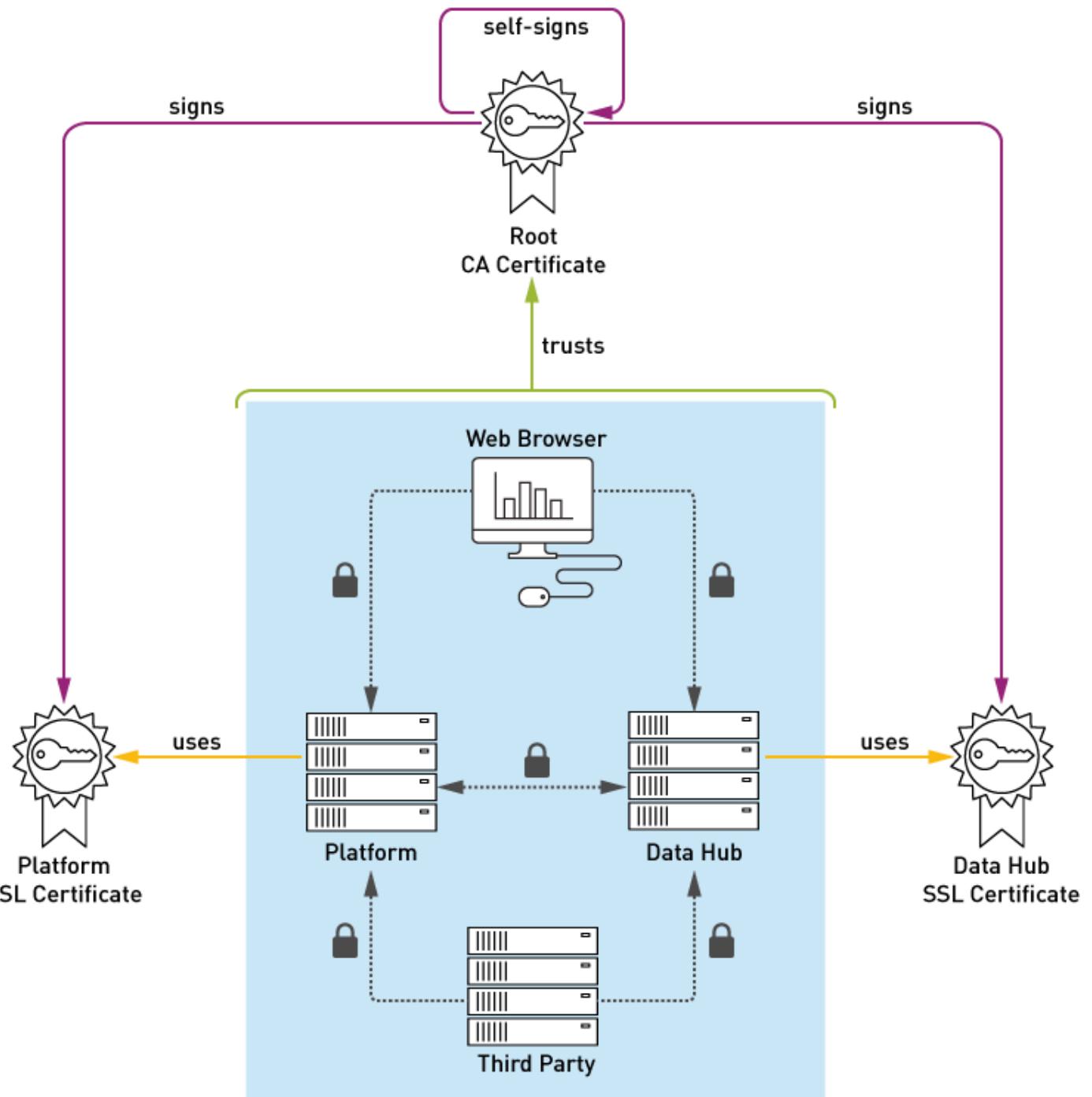
SAP Commerce handles the HTTPS traffic using a certificate signed by a self-signed RootCA.

⚠ Caution

Since the private key is known, this approach is intended only for the development environments.

Platform trusts this RootCA by default so it is possible to access an HTTPS resource from Platform itself.

The following diagram shows how the approach works:



Sample SSL Certificates for Developer Environments

Sample SSL certificates are provided for developer environments and allow you to use localhost only. Those certificates should be disabled in production environments. You can disable them by setting empty values to the following properties in your `local.properties` file:

```
additional.javafx.net.ssl.trustStore=
additional.javafx.net.ssl.trustStorePassword=
```

Implementing Encoding Algorithms with the Generic Password Encoder

With the Platform, you can implement additional **PasswordEncoder** interfaces by using the **GenericSaltedPasswordEncoder** class. This feature enables you to add additional encoding algorithms into the Platform without the need to manually create a custom class for your algorithm.

Introduction

By default, the Platform implements the **MD5**, **SHA-256** and **SHA-512** algorithms. If you want, you can use the new feature to add any of the Java supported **MessageDigest** algorithms. For details, see [Java Cryptography Architecture Standard Algorithm Name Documentation](#).

Implementing MessageDigest Algorithms

To implement an algorithm, in the **core-spring.xml** file configure the **GenericSaltedPasswordEncoder** class as a spring bean with the **algorithm** parameter. Use **genericPasswordEncoder** as a parent for that bean. As the parameter's value set one of the available **MessageDigest** algorithms.

Use the following example of implementing the **SHA-384** algorithm to implement other **MessageDigest** algorithms:

1. In the **platform/ext/core/resources/core-spring.xml** file, declare a bean and set "SHA-384" for the **value** attribute :

```
<bean id="sha384PasswordEncoder" parent="genericPasswordEncoder">
    <property name="algorithm" value="SHA-384"/>
</bean>
```

2. Add this entry into the **encoders** map in the **core.passwordEncoderFactory** bean:

platform/ext/core/resources/core-spring.xml

```
<bean id="core.passwordEncoderFactory" class="de.hybris.platform.persistence.security.PasswordEncoderFactory">
    <property name="encoders">
        <map>
            <!-- ... -->
            <entry>
                <key>
                    <value>sha-384</value>
                </key>
                <ref bean="sha384PasswordEncoder" />
            </entry>
        </map>
    </property>
</bean>
```

You must restart the Platform to apply the changes.

Caution

The MD5 Message-Digest Algorithm is not considered strong anymore.

Password Autocomplete

The autocomplete function gives you suggestions while you type your login and password into the field. It allows you to quickly provide the credentials for different users, for example while testing your software.

Password Autocomplete in SAP Commerce

The login and password autocomplete feature is disabled on a global level in SAP Commerce. However, on the cockpit level it is turned on, meaning that the global settings are overwritten by these ones specified in the extension `project.properties` file. You can find an example of login and password autocomplete setting enabled in the `project.properties` file for SAP Commerce Product Cockpit. The `project.properties` file is located in the `${HYBRIS_BIN_DIR}/modules/cockpit-applications/deprecated/productcockpit/` folder.

```
productcockpit.default.login=<login>
productcockpit.default.password=<password>
```

Disabling Password Autocomplete

For security reasons, you might want to disable Password Autocomplete. Do it on the cockpit level by deleting the values for the following properties in your extension `project.properties` file:

```
# Default login and password for logging into your cockpit:
<yourcockpitname>.default.login=
<yourcockpitname>.default.password=
```

Password Change Auditing

The password change auditing feature lets you register all the changes introduced to your passwords.

You can track changes such as the time a password changed, or who changed it. Additionally, password change auditing lets you store the hash and the encoding type of the previous password. Data about changes is stored in `UserPasswordChangeAudit` items.

Enabling Password Change Auditing

To enable password change auditing, set the `user.audit.enabled` property to true:

```
# enables user audit of password changes
user.audit.enabled=true
```

On the model service side, it is the `UserPasswordChangeAuditPrepareInterceptor` that provides the logic responsible for recording password changes. On the Jalo side, the logic is executed inside the `setPassword` method of a `User` object.

Extending Scope of Audit

To extend the scope of audited information, you need to extend the `UserPasswordChangeAudit` class with the additional attributes you want to store. To set values for these attributes, you can use the `processAfterCreation` extension point of `UserAuditFactory`. The first step is to extend `de.hybris.platform.servicelayer.user.impl.UserAuditFactory`, and override the `processAfterCreation` method. Then, you must overwrite the `userAuditFactory` bean.

Password Security Policies

SAP Commerce Platform provides customizable extension points that enable you to have fine control of password handling. With password security policies, you can define requirements that your passwords must meet before you can set or change them. This feature tremendously contributes to increasing Platform security.

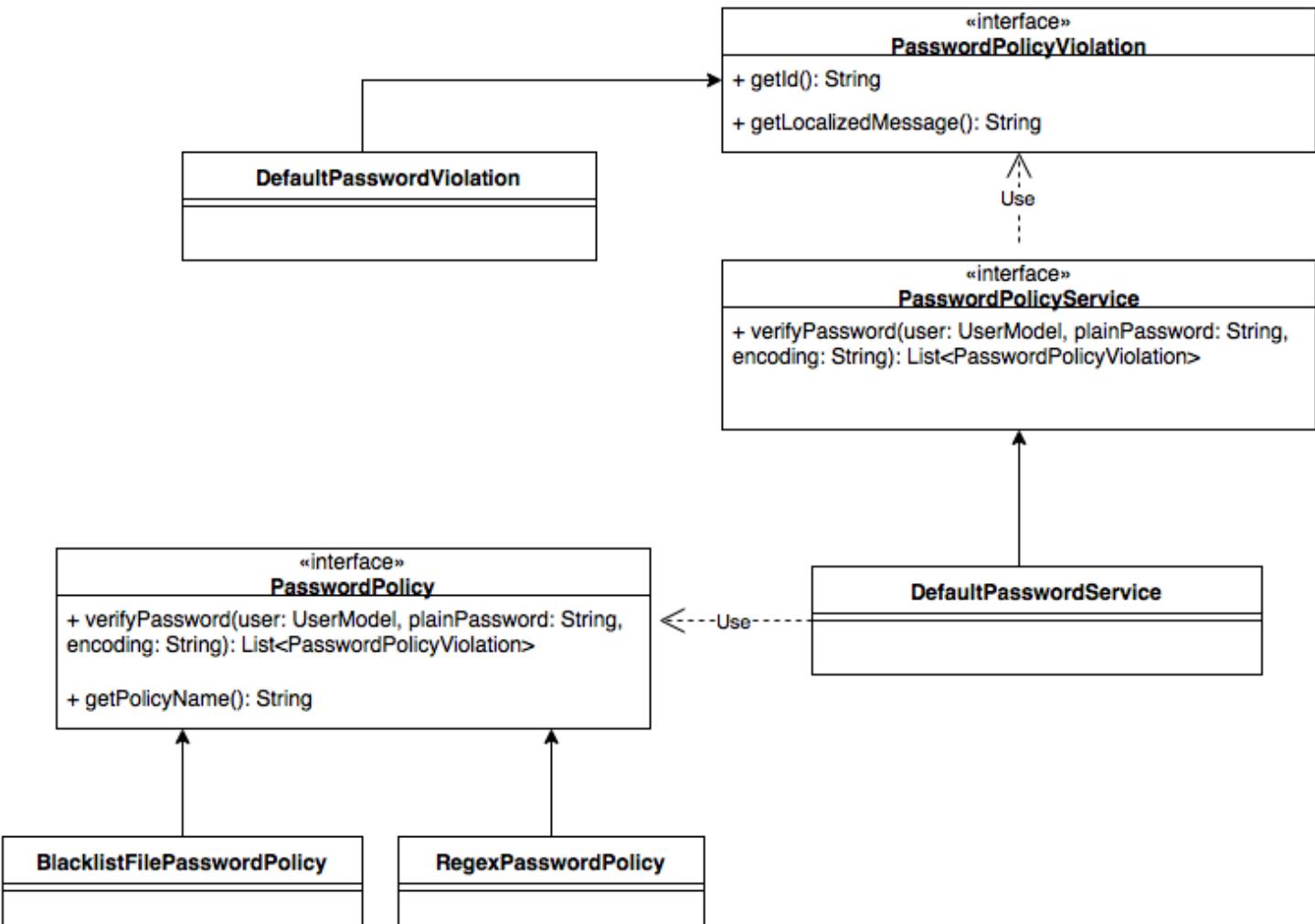
Regex and **blacklist** are two out-of-the-box password security policies delivered with Platform. Regex prevents users from using passwords that could be too easy to crack. Blacklist bans the most common passwords. You can implement your own policies too.

i Note

SAP Commerce Platform automatically handles the password security policies for Backoffice. If you want to enable this feature for customers, implement appropriate password policy handling in your storefront.

Implementation

The following diagram shows classes used to implement password security policies:



The **PasswordPolicyService** interface provides the **verifyPassword** method that checks whether your password matches defined policy requirements. If your password doesn't violate the requirements, an empty collection is returned. If it does violate, **verifyPassword** returns a list with **PasswordPolicyViolation** objects. Each violation has its unique id. You can obtain those ids calling the **getId** method.

PasswordPolicyService is used every time you set a new password, no matter if via Service Layer, or Jalo. In Service Layer, you change a password by calling the **setPassword** method on the **UserService** bean. **DefaultUserService** delegates a check to **PasswordPolicyService**, and throws **PasswordPolicyViolationException** if a new password doesn't fulfill requirements. The same check takes place inside the **setPassword** method on a **User** Jalo object. Nevertheless, the recommended way to validate user input for a new password is to call **verifyPassword** explicitly, and set the password only if no violations are returned.

The `PasswordPolicyViolation` interface provides the `getLocalizedMessage` method that returns localized violation messages. You can use these messages to provide feedback for users in the view layer.

Assigning Password Policies to User Groups

Password policies aren't enabled for any user group unless you configure them. To enable an example policy named `foo` for user groups `group1` and `group2`, set the following property as shown:

```
password.policy.foo.groups.included=group1,group2
```

You can also use the `*` character as a wildcard - it matches all user groups and thus enables a given policy for everyone in the system. To exclude users when using a wildcard, use the `excluded` property:

```
# match all groups except excluded_group for foo policy
password.policy.foo.groups.included=*
password.policy.foo.groups.excluded=excluded_group
```

Available Password Policies

Platform comes with two out-of-the-box implementations of password policies:

Password policy name	Implementation class	Bean	Description
regex	de.hybris.platform.servicelayer.user.impl.RegexPasswordPolicy	regexPasswordPolicy (user-spring.xml)	Matches a set of required or regular expressions defined against a password
blacklist	de.hybris.platform.servicelayer.user.impl.BlacklistFilePasswordPolicy	blacklistPasswordPolicy (user-spring.xml)	Checks password against a prohibited passwords defined available on classpath - property <code>password.policy.blacklist</code> in platform/project

Regex Password Policy

The regex password policy allows you to configure a set of rules based on regular expressions. For example, you can decide that your passwords are at least 8 characters long and include at least one digit. These rules can either be required or disallowed:

Type	Property	Description
Required	<code>password.policy.regex.required.[rule_id]</code>	Regular expression must match against a new password
Disallowed	<code>password.policy.regex.disallowed.[rule_id]</code>	Regular expression must not match against a new password

Replace the `[rule_id]` placeholder with an assigned unique password violation id. This text is returned by the `getId` method. It's also used to find proper localization text for the `getLocalizedMessage` method.

See the example of defining regex rules:

```
# enforces passwords to be between 6 and 128 characters
password.policy.regex.required.certainsize=.{6,128}

# enforces passwords to contain digit
password.policy.regex.required.mustcontaindigit=.*\d+.*

# prohibits from setting '12345678' as a password
password.policy.regex.disallowed.not12345678=12345678
```

To provide localization texts for these rules, create `password.policy.violation.regex.required.[rule_id]` or `password.policy.violation.regex.disallowed.[rule_id]` keys with translation in `resources/localization/[extension]-locales_en.properties` bundle.

See how you can provide localization for rules defined in the previous example:

```
password.policy.violation.regex.required.certainsize=Password must contain between 8 and 128 characters.
password.policy.violation.regex.required.mustcontaindigit=Password must contain at least 1 digit.
password.policy.violation.regex.disallowed.not12345678=Using '12345678' as the password is not allowed.
password.policy.violation.blacklist.blacklistedpassword=Selected password is a blacklisted.
```

Blacklist Password Policy

Blacklist password policy allows you to specify a file that contains a list of passwords (one password per line) that aren't allowed. To specify a blacklist file name, set the `password.policy.blacklist.file` property with a name of the file available on the classpath.

For example, this setting allows you to use the `password-blacklist.txt` file stored in an extension resources folder:

```
password.policy.blacklist.file=password-blacklist.txt
```

Creating Custom Password Policies

To create a custom password policy, create a bean that implements the `PasswordPolicy` interface. `DefaultPasswordPolicyServices` uses autowiring by type so it detects a custom implementation available to use. Assign your policy to appropriate groups. A policy name is a string returned by the `getName` method. Remember to obey the contract and provide password policy violations with correct translations.

Password Storage Strategies

When saving user passwords, it is important to protect them against unauthorized access. For that reason, Platform always stores passwords in an **encoded** format.

You can choose from multiple encoding strategies (known as **password encoders**) available. In addition, you can implement your own strategies.

Change the Default

Instead of requiring you to explicitly specify the strategy, some APIs allow you to set passwords using a **default** strategy. The default strategy in Platform is . You can change it if necessary through the `<default.password.encoding>` property.

```

#
# The code of the password encoder to use as default. These are the available options (from core-spring-boot)
#
# '*' .. legacy 'default' mapping to 'plain' (not changeable before 5.7 - do not use anymore)
# 'plain' .. plain text
# 'sha-256' .. SHA 256
# 'sha-512' .. SHA 512
# 'md5' .. MD5 (for legacy reasons - do not use in production )
# 'pbkdf2' .. PBKDF2 (strong and configurable)
# 'bcrypt' .. (strong and configurable; a current default for FIPS compliance purposes)
# 'scrypt' .. (strong and configurable; one of the OWASP-recommended strategies)
# 'argon2' .. (strong and configurable; most recommended by OWASP)
#
#default.password.encoding=bcrypt
default.password.encoding=argon2

```

i Note that changing the default strategy this way preserves previously stored passwords (using the **default** strategy) so that the user can still log in. **⚠️** In case the previous strategy was considered to be unsafe, all previous passwords remain to be unsafe as well!

Configure New Generic Strategy

Adding a new password hashing strategy is now made easy by providing a **generic** implementation that just requires you to specify the hashing algorithm.

Refer to [Implementing Encoding Algorithms with the Generic Password Encoder](#) to find out how to do it.

Implement Your Own Strategy

It is also possible to implement and add a custom password hashing strategy:

```

public class DummyEncoder implements PasswordEncoder
{
    @Override
    public String encode(final String uid, final String plain)
    {
        return Integer.toString(plain.hashCode());
    }
    @Override
    public boolean check(final String uid, final String encoded, final String plain)
    {
        return encoded.hashCode() == plain.hashCode();
    }
    @Override
    public String decode(final String encoded) throws EJBCannotDecodePasswordException
    {
        throw new EJBCannotDecodePasswordException(new Throwable("Dummy encoded passwords cannot be decoded"));
    }
}

```

Now add the implementation via Spring:

```

<bean id="dummy" class="foo.bar.DummyEncoder"/>
<bean id="core.passwordEncoderFactory" class="de.hybris.platform.persistence.security.PasswordEncoderFactory">
    <property name="encoders">
        <map>
            <!-- Attention: Ensure to preserve the built-in encoders.
If you remove or change them, the existing passwords will be lost !!!
-->
            <entry key="*" value-ref="${default.password.encoder}" />
            <entry key="plain" value-ref="core.plainTextEncoder" />
            <entry key="sha-256" value-ref="sha256PasswordEncoder" />
            <entry key="sha-512" value-ref="sha512PasswordEncoder" />
            <entry key="md5" value-ref="core.saltedMD5PasswordEncoder" />
            <entry key="pbkdf2" value-ref="pbkdf2PasswordEncoder" />

            <entry key="argon2" value-ref="argon2PasswordEncoder"/>
            <entry key="scrypt" value-ref="scryptPasswordEncoder"/>
            <entry key="bcrypt" value-ref="bcryptPasswordEncoder"/>
            <!-- the new one -->
            <entry key="dummy" value-ref="dummy" />
        </map>
    </property>
</bean>
```

See above to find out how to change the default so you can use the new strategy.

Localize Labels For Encoders

To localize a Backoffice label for a newly created encoder, add an appropriate entry to the `labels.properties` file located in the `<YOUR_EXTENSION>/resources/<YOUR_EXTENSION>-backoffice-labels` directory. This entry should match the following pattern:

```
hmc.encrypt_<encoder>=<localized encoding name>
```

The encoder should match the key property in the `core.passwordEncoderFactory` bean in the `ext/core/resources/core-spring.xml` directory, for example:

```
hmc.encrypt_bcrypt=BCrypt
```

If you need a quick and dirty solution, you can put this entry into the `resources/platformbackoffice-backoffice-labels/labels.properties` file in the `platformbackoffice` extension, however, we don't recommend modifying this file.

One-Time Password

SAP Commerce supports the functionality of creating and validating one-time passwords, which is a form of two-factor authentication (2FA). These passwords can be used in various scenarios.

Customer Login with One-Time Verification Token

You can enhance security using the one-time password functionality. To enable one-time passwords for customers when logging into composable storefronts:

- Update your composable storefront codebase and enable the one-time password functionality for customers. For more information on how to do this, see [Patch Upgrade Notes](#).

When one-time verification token for customer login is enabled, the Customer User token can be retrieved from the Commerce Authorization Server only with the one-time password *<Token Id>* provided as *<username>* and one-time password *<Token Code>* provided as *<password>*. To be able to fetch a user token you first have to create a one-time login password for that Customer. After fetching one-time password *<Token Id>* from the OCC API and receiving *<Token Code>* in an email, you can request a Customer Token with the following command:

```
curl --location --request POST 'https://<COMMERCE_CLOUD_BACKOFFICE_URL>/authorizationserver/oauth/1'
--header 'Accept: application/json, text/plain, */*' \
--header 'Authorization: Basic <BASE64_ENCODED_OAUTH_CLIENT_CREDENTIALS>' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--header 'X-Requested-With: XMLHttpRequest' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'scope=basic' \
--data-urlencode 'username=<TOKEN_ID>' \
--data-urlencode 'password=<TOKEN_CODE>'
```

i Note

One-time password for customer login functionality only affects Customer type Users. Employee type Users are not affected and should authenticate/fetch tokens in a standard way using their credentials.

Preventing Brute Force Attacks

Using the `otp.customer.login.token.max.verification.attempts` property, you can set the maximum number of failed login attempts for a one-time token. If the maximum number of failed login attempts is reached, the specific one-time token can no longer be used to authenticate as it was removed. By default, the token is removed after the third failed attempt. You can override it in `local.properties`, as shown in the example:

```
otp.customer.login.token.max.verification.attempts = 5
```

The Customer login one-time password functionality can be adjusted by overriding the default configuration properties, described below:

```
# Specifies if the one time password (OTP) functionality is enabled for customer login
otp.customer.login.enabled = false
# Specifies the token encoder to be used for one time password code encoding functionality, see the
otp.customer.login.token.code.encoder = pbkdf2
# When OTP is enabled allows to specify the length of verification tokens for OTP login functional:
# the minimum allowed value is 6 characters, maximum allowed value is 34 characters
otp.customer.login.token.code.length = 8
# When OTP is enabled allows to set a specific character set with which the code for the OTP login
# The minimal charset length should be 10 characters. If the provided charset is shorter, the defau
otp.customer.login.token.code.alphabet = 1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
# When OTP is enabled allows to specify time to live in seconds for the OTP login functionality
# the maximum allowed value is 900 seconds (15 minutes), minimum allowed value is 60 seconds
otp.customer.login.token.ttlseconds = 300
# Specifies the maximum number of failed verification attempts for a single token, after which the
# 1 means that the token is removed after the first failed verification attempt ( no retries allowe
otp.customer.login.token.max.verification.attempts = 3
# Number of bits used to generate the random part of the token id for the one time password (OTP)
# The minimum allowed value is 128 bits. Its recommended to use at least 256 bits. Max value is 102
otp.customer.login.token.id.generator.bits = 256
# This short name is used to quickly identify the tokens generated for the OTP login functionality.
# The token id format is <{shortName}{randomPart}>
# The short name should be unique for initial verification purpose and kept short (max 10 character)
```

```
# If the short name exceeds 10 characters, it will be fallback to default 'OTH' (other) short name.
otp.customer.login.token.purpose.short.name = LGN
```

Leveraging One-Time Password Feature

One-time passwords can be easily reused for other scenarios. To do so, you can define the new purpose of the one-time password by:

- Extending `de.hybris.platform.core.enums.SAPUserVerificationPurpose` in `*-items.xml` with a new purpose, for example `REGISTRATION`, `ORDER` or other.
- Use the `de.hybris.platform.servicelayer.user.UserVerificationTokenService` to create, lookup, and handle one-time token verification in your business logic.

Each `de.hybris.platform.core.enums.SAPUserVerificationPurpose` can have a custom set of additional configuration properties defined. Here's an example of a sample set of custom properties for the new `REGISTRATION` verification purpose type:

```
## The recipe for configuring new OTP purpose functionality is to create a set of properties according to the SAP User Verification Purpose enum.
## The same safe defaults as for customer login OTP are used for other OTP functionalities.
## Lets see this on example set of properties for customer registration OTP after REGISTRATION code
#otp.customer.registration.enabled = false
#otp.customer.registration.token.code.encoder = pbkdf2
#otp.customer.registration.token.code.length = 8
#otp.customer.registration.token.code.alphabet = 1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
#otp.customer.registration.token.ttlseconds = 300
#otp.customer.registration.token.max.verification.attempts = 3
#otp.customer.registration.token.id.generator.bits = 256
#otp.customer.registration.token.purpose.short.name = REG
```

If some property value isn't provided for a new verification purpose type, the default value is applied.

Related Information

[Flows](#)

[Verification Token](#)

Protecting Against Cross-Site Scripting

Inadequately protected web applications are exposed to cross-site scripting (XSS) attacks.

XSS attacks are aimed at end users and are possible after a malicious code is sent to and executed by end users' web browsers. This code can be sent in the form of a web page link or through a web application into which it was previously injected. For example, an attacker can inject a malicious code into an unprotected online store and its database by leaving a comment about a product. If not stripped or encoded, the code will be executed after being displayed in end users' web browsers. Depending on the code's purpose, it might enable the attacker to access the end users' private information such as cookies or passwords they use in a given web application.

The XSS Encoder

The `security.core.server.csi-1.00.8.jar` library encodes different kinds of content and helps prevent XSS attacks from malicious data being stored inside the Platform database.

The library was added to the `hybris ext/core/lib` folder. The following methods are available to prevent XSS.

Context	Method
HTML / XML	out = XSSEncoder.encodeHTML(in) and XSSEncoder.encodeXML(val)
JavaScript	out = XSSEncoder.encodeJavaScript(val)
URL	out = XSSEncoder.encodeURL(val)
CSS	out = XSSEncoder.encodeCSS(val)

For more information, see:

- [SAP Encoding Functions for AS Java and JavaScript](#)
- [Output Encoding Contexts](#)

Web Security XSSFilter

XSSFilter is a simple generic cross-site scripting protection filter for the Platform.

Under normal operation, enabling the filter adds an average overhead of 50 ms per page requests in a performance testing environment. When the number of requests reaches the limit of the system capacity, the response time may increase significantly.

XSSFilter can be enabled or disabled. By default, the filter is enabled, and SAP recommends that it remains enabled.

i Note

XSSFilter doesn't handle HTTP request bodies. It filters request parameters and headers only.

Configuring XSSFilter

You can find a set of modifiable properties included in `platform/project.properties`:

```
#####
# WEB-SECURITY SETTINGS #####
#
# Here web related security settings can be found.
#
#####
# enable globally
xss.filter.enabled=true

# define action on violation matching globally
# STRIP .. strips all text occurrences which match the patterns below but allow
#           processing the request (default)
# REJECT.. if any pattern matches the whole request gets rejected with the  BAD REQUEST
#           error code
xss.filter.action=STRIP

# our default rules
xss.filter.rule.script_fragments=(?i)<script>(.*)</script>
xss.filter.rule.src=(?ims)[\\s\\r\\n]+src[\\s\\r\\n]*=[\\s\\r\\n]*'(.*)'
xss.filter.rule.lonely_script_tags=(?i)</script>
```

```
xss.filter.rule.lonely_script_tags2=(?ims)<script(.*)>
xss.filter.rule.eval=(?ims)eval\\((.*?)\\)
xss.filter.rule.expression=(?ims)expression\\((.*?)\\)
xss.filter.rule.javascript=(?i)javascript:
xss.filter.rule.vbscript=(?i)vbscript:
xss.filter.rule.onload=(?ims)onload(.*)=
```

Disabling and Enabling XSSFilter

The filter can be disabled or enabled globally with just one parameter:

```
xss.filter.enabled=true
```

You can disable the filter if you have a web application firewall in front of SAP Commerce or if you have some other means of dealing with malicious input.

You can also disable or enable the filter per extension. Just add the **extension name** as a prefix to the parameter:

```
hac.xss.filter.enabled=false
```

Action on Match

The default reaction on matching any rule is that each occurrence of these text fragments is stripped, and the request proceeds:

```
xss.filter.action=STRIP
```

For testing purposes, you may want to change that setting to reject the request with the BAD REQUEST error code:

```
xss.filter.action=REJECT
```

Again, you can set the action for a specific extension:

```
hac.xss.filter.action=REJECT
```

Rules

i Note

Default rules are just a basic set and you should expand them within your projects.

Rules are globally defined as properties. A rule property must start with `xss.filter.rule` in its name, followed by a specific rule ID, which is used for information purposes only:

```
xss.filter.rule.script_fragments=(?i)<script>(.*)</script>
xss.filter.rule.src=(?ims)[\\s\\r\\n]+src[\\s\\r\\n]*=[\\s\\r\\n]*'(.*)'
xss.filter.rule.lonely_script_tags=(?i)</script>
xss.filter.rule.lonely_script_tags2=(?ims)<script(.*)>
xss.filter.rule.eval=(?ims)eval\\((.*?)\\)
xss.filter.rule.expression=(?ims)expression\\((.*?)\\)
xss.filter.rule.javascript=(?i)javascript:
```

```
xss.filter.rule.vbscript=(?i)vbscript:  
xss.filter.rule.onload=(?ims)onload(.*)?=
```

You may override rules globally or even per extension. Make sure to match the exact rule ID:

```
hac.xss.filter.rule.vbscript=
```

Configuration in web.xml

To enable the filter, change `web.xml` files for each web application. Make sure that the filter:

1. matches all requests
2. comes first in the overall filter chain

```
<filter>  
  <filter-name>XSSFilter</filter-name>  
  <filter-class>de.hybris.platform.servicelayer.web.XSSFilter</filter-class>  
</filter>  
  
<filter-mapping>  
  <filter-name>XSSFilter</filter-name>  
  <url-pattern>/*</url-pattern>  
</filter-mapping>
```

Sorting Filters and Rules

`XSSFilter` contains a list of patterns against which it filters dangerous content. With the default `XSSMatchAction.STRIP` action, the order of the items in the pattern list is important because it affects the filtering outcome. See these examples:

```
//Sorting Order 1  
input: >><script>abc</script><<  
patterns: {  
(?i)<script>(.*)?</script>  
(?i)</script>  
(?ims)<script(.*)?>  
}  
  
output: >><<
```

```
//Sorting Order 2  
input: >><script>abc</script><<  
patterns: {  
(?i)</script>  
(?ims)<script(.*)?>  
(?i)<script>(.*)?</script>  
}
```

```
output: >>abc<<
```

You can see that the patterns are the same in both lists but the outcomes differ because the patterns are ordered differently. Without sorting, pattern definitions are provided in an undefined order.

You can, however, order filters and rules in your list by sorting them alphabetically. See the example:

```
xss.filter.rule.001_script_fragments=(?i)<script>(.*)</script>
xss.filter.rule.002_src=(?ims)[\s\r\n]+src[\s\r\n]*=[\s\r\n]*'(.*)'
xss.filter.rule.003_lonely_script_tags=(?i)</script>
xss.filter.rule.004_lonely_script_tags2=(?ims)<script(.*)>
```

In this way you make sure you get the outcome you aim for.

Ordering filter rules alphabetically is disabled by default. To enable it, set:

```
xss.filter.sort.rules=true
```

To override it per extension, set:

```
[extension_name].xss.filter.sort.rules=false
```

Servlet 3.0

In Servlet 3.0 containers (Tomcat 7) there may be servlets making use of asynchronous processing. In that case make sure that the filter is configured to support that by adding `<async-supported>true</async-supported>`:

```
<filter>
  <filter-name>XSSFilter</filter-name>
  <filter-class>de.hybris.platform.servicelayer.web.XSSFilter</filter-class>
  <async-supported>true</async-supported>
</filter>
```

Libraries

The filter is part of the platform global classpath. Therefore, you do not need to add any library to your platform web application.

Injecting Static HTTP Response Headers

You can configure SAP Commerce to inject static headers into HTTP response.

To inject headers globally, that is for all SAP Commerce web applications, use:

```
xss.filter.header.[header name]=[header value]
```

This property can also be set for a **specific** web application only. To set it, prefix the parameter with a web application name.

```
[webapp name].xss.filter.header.[header name]=[header value]
```

For example, you may want to add X-Frame-Options to all SAP Commerce web applications to prevent clickjacking. The X-Frame-Options header is used only as an example - it is already set to SAMEORIGIN by default. You need to add the following parameter

to the `local.properties` file.

```
xss.filter.header.X-Frame-Options=SAMEORIGIN
```

However, if you would like to add this header only for a web application "foo", you would have to add the parameter with the key prefixed with the application name.

```
foo.xss.filter.header.X-Frame-Options=SAMEORIGIN
```

Remember Me Feature

Platform supports the Remember Me feature so that users can log on to the application without the need to provide their credentials every time they access it.

Platform uses a cookie-based login token (`LoginToken`) to authenticate users whose credentials have been stored by the Remember Me feature. The token uses the `CoreRememberMeService` class that implements the Spring `RememberMeServices` class interface.

Enhanced Login Token Generation

Using cookie-based login tokens has been enhanced as the following information is added to their value:

- their TTL (Time to Live) timestamps that are further verified on the server side. If the value of the timestamp has expired, the user isn't authenticated.
- a randomly generated salt that is used for password rehashing.
- a randomly generated value that is stored in the database for each user

The randomly generated value can be revoked. As a result of such revocation, all cookie-based login tokens generated for users become invalid and users can't be authenticated through the old token. To revoke user tokens, use the `de.hybris.platform.jalo.user.TokenService` interface and the dedicated `revokeTokenForUser(final String userId)` method. The default implementation (`DefaultTokenService`) replaces random values with new ones.

If you want to turn off the default behavior of the enhanced login token generation, add the following property to your `local.properties` file:

```
login.token.extended=false
```

Caution

Due to security reasons, it's not recommended to disable this behavior.

Accepting Login Tokens as URL Parameters

Login tokens aren't accepted as URL parameters. However, if you want to enable such login tokens, add the following property to your `local.properties` file:

```
login.token.url.enabled=true
```

Caution

Due to security reasons, it's not recommended to enable this behavior.

Disabling Authentication Through Login Tokens

You can disable authentication through login tokens by adding the following property to your `local.properties` file:

```
login.token.authentication.enabled=false
```

If you want to disable login token authentication for given web applications, use the `login.token.authentication.[extensionName].enabled=false` property, for example:

```
login.token.authentication.backoffice.enabled
```

If the property is not set for given web applications, the value of the `login.token.authentication.enabled` is used. See an example of a setting that enables authentication with login tokens for Backoffice and disables it for Administration Console:

```
login.token.authentication.hac.enabled=false
login.token.authentication.backoffice.enabled=true
login.token.authentication.enabled=false
```

Setting Basic Authentication in Core Plus Services

This document explains how to set up basic authentication between SAP Commerce and Core+ services, and, more specifically, between SAP Commerce and Core+-based REST clients.

Suggested Reading

For more information on basic authentication, refer to the documentation provided on the Tomcat website at <http://tomcat.apache.org/tomcat-7.0-doc/realm-howto.html>.

Transparent Attribute Encryption (TAE)

To encrypt sensitive data transparently, SAP Commerce supports you in declaring a `String-Attribute` as encrypted just by adding the modifier `encrypted="true"` in the `items.xml` file of your extension.

Why Encryption?

You built a secure system, encrypted the most sensitive data, and built a firewall around the database servers. But the thief took the easy approach: They took the backup tapes of your database.

Protecting the database data from such theft is not just good practice; it's a requirement for compliance with most laws, regulations, and guidelines. Consequently, encryption lets you protect your database from this vulnerability.

How It Works

The first layer of defense is the firewall around the whole information infrastructure of your organization, which keeps outsiders from accessing any of the information sources inside your company.

If an intruder gets past the external firewall, that person will be required to supply a password to access the server or perhaps be asked to provide other authentication credentials such as security certificates. This is the second layer of security.

After being authenticated, the legitimate user must only be allowed to access those assets that person is supposed to access.

If a user gets into the database but has no authority to see any table, view, or any other data source, the information is still protected. This mechanism is the next layer of security.

Some security-related compliancy regulations stress that it is possible for an intruder to somehow defeat all of the protective measures and get to the enterprise data. From a planning perspective, this possibility must be accepted, analyzed, and accounted for.

The only option left for defending against an intruder at this point, the last layer of security, is to alter the data, via a process known as encryption, in such a way that the intruder will not find it useful.

Encryption alters data to make it unreadable to all except those who know how to decipher the information.

In a symmetric cipher, the key for encryption and decryption is the same. The Caesar cipher is a simple example of such a symmetric cipher.

Modern symmetric cryptographic algorithms are much more advanced than the Caesar cipher, but they operate in similar fashion - one passes the plaintext into a cipher with a given key and out comes the ciphertext. The same procedure in reverse produces the original (plaintext) message.

Encryption and Decryption Process

Encrypting a value involves passing the original data and the encryption key to the encryption algorithm to create encrypted data.

Element	Sample
Original Data	1234 1234 1234 1234
Key	nKZkqm!3~!;r^L#td,t!h1+;!vwM]lpy"IEZ+{kKA:UkjYn_-:M)IDclj)NYm ^1
Encryption Algorithm	AES
Encryption Data	LFGjqOi4quc=K8p+AIJJBoIQTpbh7yaSojr97rHES7YaTkY06SqBA/M=

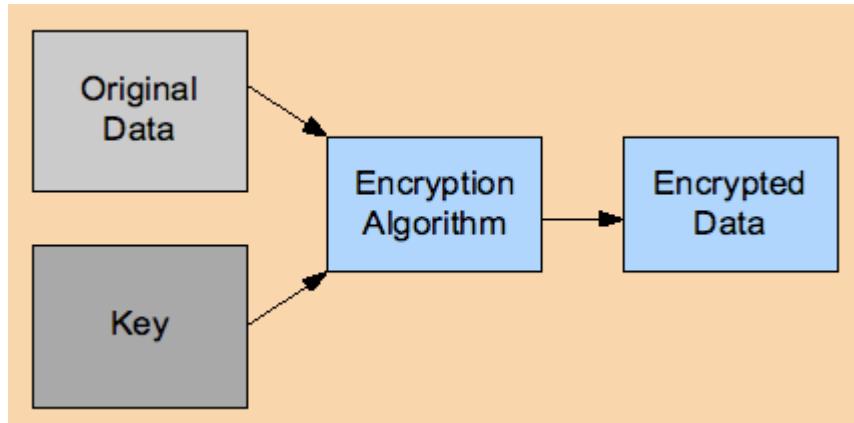


Figure: Encryption mechanism.

During decryption, the logic is reversed, producing the original value. As explained above, since the same key is used to encrypt and decrypt, this scheme is also known as symmetric encryption.

Encryption Algorithm Used by TAE

TAE uses the AES (Advanced Encryption Standard) as default encryption algorithm, which was chosen in October 2000 by the National Institute of Standard and Technology (NIST). They selected an algorithm called Rijndael, developed by Joan Daemen and Vincent Rijman. AES allows key lengths of 128, 192 and 256 bits.

Limitations of Symmetric Encryption

All symmetric algorithms will work in roughly the same fashion. The same key is used to both encrypt and decrypt messages. This means that in order to send a message to someone you must agree on a key beforehand.

This is the problem with symmetric, or secret-key encryption - the key needs to be kept secret, and any exposure of the key compromises the secrecy of any ciphertext, such as the encrypted credit card number created with that key.

Key Generation

You can find an AES key generator for 128, 192 and 256 bits key in SAP Commerce Administration Console.

1. Open SAP Commerce Administration Console.
2. Go to the **Maintenance** tab and select **Encryption Keys** option.
3. The **Encryption Keys** page in the **Generator** tab displays.

For more information, see [Maintenance Tab](#).

The screenshot shows the SAP hybris administration console interface. At the top, there's a blue header bar with the title '(v) hybris administration console'. To the right of the title is a search bar with a magnifying glass icon and the placeholder 'Type here...'. Below the header, there's a dark blue navigation bar with tabs for 'Platform', 'Monitoring', 'Maintenance', and 'Console'. The 'Maintenance' tab is currently selected. On the left side of the main content area, there's a sidebar with a tree view. The main content area has a title 'Encryption Keys'. Below the title, there are three tabs: 'Generation' (which is selected), 'Migration', and 'Credit cards encryption'. Under the 'Generation' tab, there are two configuration sections: 'Key size' (set to '128') and 'Output file' (set to 'Generated-KEYSIZE-Bit-AES-Key.hybris'). At the bottom of this section is a blue 'Generate' button. The overall interface is clean and modern, typical of SAP's administration tools.

Encryption Keys page in the Generation tab

TAE and Salt

Encryption is all about hiding data, but sometimes it is easier to guess the value of encrypted data if there is repetition in the original plain text value of the data. (This type of attack is called Known Plaintext Attack). For instance, a salary information table may contain repeated values. In that case, the encrypted values will be the same, too, and an intruder could determine all entries with the same salary.

To prevent such an occurrence, an additive, also called salt, is added to the data. It makes the encrypted value different, even if the input data is same. TAE, by default, applies a salt.

TAE stores the salt with the data that is encrypted. Each time a new piece of data is encrypted, a new salt is generated. This means that the same text encrypts to a different value each time it is encrypted, even if the same key is used every time. To decrypt, the salt must be extracted from the encrypted data, and then combined with the key to create the decryption key.

Key Management and Key Rotation

Key management is the foundation of any solid encryption implementation. Unless an organization establishes a systematic approach to generate, rotate and store its keys, its encryption activities will be largely futile.

Unfortunately, while data encryption itself can be reasonably easy to achieve, efficient management of encryption keys across their lifecycle continues to be a problem.

Annual rotation of encryption keys is required by data security regulations such as PCI DSS while security best practices indicate that rotation should be performed far more frequently. (The PCI Security Standards Council has indicated more frequent rotations will be required in a soon-to-be released revision of the standard).

Caution

Key Rotation: SAP Commerce TAE supports re-encryption of all historical data with the new key on the fly. The sample below explains how this works. A poorly implemented rotation process can create new data security vulnerabilities and may make critical data inaccessible even to authorized users. We strongly recommend that you never ever delete any encryption key you used for securing your productive data and that you will keep track of the related master password (symmetric.key.master.password).

project.properties/local.properties

```
# all keys have to be stored under: ${platform_config_dir}/security/
symmetric.key.file.1=weak-symmetric.key
symmetric.key.file.default=1
symmetric.key.master.password=w427tg3uy73uioomc1fohx1w6pew00n124mlt8ksplpm6ynz55z6305w2nwtj22
```

In the sample above all sensitive attributes (items.xml, encrypted=true) will be encrypted with the key that is specified by the setting `symmetric.key.file.default`.

In our sample this will be the key `weak-symmetric.key` (key id = 1). To make this encryption key unique for a specific SAP Commerce installation this one is protected by the `symmetric.key.master.password`.

Note

The format of the key definition is: `symmetric.key.file.<key id> = <name of the key file>`

Caution

All key files have to be stored in `${platform_config_dir}/security/` and every key file which uses `.hybris` as its file extension is expected to be encrypted with the configured master password. It is recommended that you use our AES key generator because this one will place the file in the right place, create secure random keys and use the proper file extension, too. For more information, see section Key Generation.

As a result of this, configuration encrypted attributes will be stored like:

1:LFGjqOi4quc=K8p+AIJJBoIQTpbh7yaSojr97rHES7YaTkYO6SqBA/M=

Here the prefix '1:' indicates that the key id '1' will be used for decrypting this value.

Changing the Used Encryption Key

In the new sample configuration shown below we specify a new encryption key (id=2) and by setting `symmetric.key.file.default=2` this key will be used for all new encryption operations.

`project.properties/local.properties`

```
# all keys have to be stored under: ${platform_config_dir}/security/
symmetric.key.file.1=weak-symmetric.key
symmetric.key.file.2=Generated-256-Bit-AES-Key.hybris
symmetric.key.file.default=2
symmetric.key.master.password=w427tg3uy73uiocomc1fohx1w6pew00n124mlt8ksplpm6ynz55z6305w2nwtj22
```

As a result of this configuration encrypted attributes will be stored like:

2:m2FF0RUDs04=3QiJ+1QJG89AWB3sIRZwqBrP65SUh/gOglsZESrXuKs=

Here the prefix '2:' indicates that the key id '2' will be used for decrypting this value.

At this time we will have the following encrypted entries in our database:

1:LFGjqOi4quc=K8p+AIJJBoIQTpbh7yaSojr97rHES7YaTkYO6SqBA/M=

2:m2FF0RUDs04=3QiJ+1QJG89AWB3sIRZwqBrP65SUh/gOglsZESrXuKs=

At the point of time when an already encrypted entry will be rewritten, the key with id=2 will be used for this operation.

So an entry like

1:LFGjqOi4quc=K8p+AIJJBoIQTpbh7yaSojr97rHES7YaTkYO6SqBA/M=

will be (re)stored like

2:yJ2jg1s37Js=i3/Lwy6PXHrlN4LL3dUxIWaqW1/JTVY5d4CtyvP75Dc=

And now the encrypted database entries will look like:

2:yJ2jg1s37Js=i3/Lwy6PXHrlN4LL3dUxIWaqW1/JTVY5d4CtyvP75Dc=

2:m2FF0RUDs04=3QiJ+1QJG89AWB3sIRZwqBrP65SUh/gOglsZESrXuKs=

Here encrypted attributes will only be re-encrypted with the new key if the corresponding instance will be loaded and restored. To encrypt all encrypted attributes with the new key, use SAP Commerce Administration Console. A poorly implemented rotation process can create new data security.

See also [Encryption Keys Migration](#).

More about Encryption Keys

Platform uses AES symmetric algorithm to encrypt any value for an attribute that is configured as `encrypted=true` in `items.xml`. If not configured differently by the user, Platform uses the default AES key named `default-128-bit-aes-key.hybris` located in the `bin/platform/ext/core/resources/security` directory. This file is pre-generated with use of the default salt 1234567.

→ Remember

Change this salt to something unique and longer than the default value, and generate your own unique AES key.

The whole encryption key configuration is stored in properties and follows a simple schema:

```
symmetric.key.file.<ID>=filename
symmetric.key.file.default=<ID>
```

The `symmetric.key.file.<ID>` property points to the file name of the key. The `<ID>` is an ID of the key. You can have more than one key configured and have them stored under unique IDs that are integer values. The `symmetric.key.file.default` property points to the ID of the key considered as the default one in the system, and every instance of an Item that is created uses this key to encrypt data. Take a look at example configuration:

```
symmetric.key.file.1=default-128-bit-aes-key.hybris
symmetric.key.file.default=1
```

Here is another configuration with more than one key:

```
symmetric.key.file.1=default-128-bit-aes-key.hybris
symmetric.key.file.2=my-own-256-bit-aes-key.hybris
symmetric.key.file.default=2
```

The above example shows a bit more complicated configuration where you have two keys. The key with ID=2 is the default one but the data is stored in the database encrypted with key ID 1 and key ID 2 may be used simultaneously. It results from how encrypted data is stored in the database.

How encrypted data is stored in the database

Encrypted data is stored in the database using following pattern:

```
keyID:encrypted data
```

The key ID before the colon is the ID of the key that is used to encrypt data that you see after the colon. This way you can have data in your database encrypted by many keys. One condition must be fulfilled though - you can't shuffle your keys in the configuration. Otherwise underlying logic may choose different key to decrypt data from the one which was used initially to encrypt it.

Encryption Keys Migration

Encryption key migration importance and frequency depends on your security policy. It may also become necessary as an emergency measure in the case where existing keys have become unsecure or were compromised.

Context

Procedure

1. Open SAP Commerce Administration Console.
2. Go to the **Maintenance** tab and select **Encryption Keys** option.
3. The **Encryption Keys** page in the **Generator** tab displays.

The screenshot shows the hybris administration console interface. At the top, there's a blue header bar with the title '(v) hybris administration console'. To the right of the title, it says 'You're Administrator - logout' and has a search bar with placeholder text 'Type here...'. Below the header, there are four tabs: 'Platform', 'Monitoring', 'Maintenance', and 'Console'. The 'Console' tab is selected. In the main content area, there's a sub-header 'Encryption Keys'. Below it, there are three tabs: 'Generation', 'Migration', and 'Credit cards encryption'. The 'Migration' tab is selected. Under 'Encryption Keys', there's a table with two rows:

ID	Key File
1	Generated-KEYSIZE-Bit-AES-Key.hybris
2 (default)	Generated-derberg-Bit-AES-Key.hybris

Below this, there's a section titled 'Encrypted Attributes' with a table:

Selection	Type	Attribute	Instances
<input checked="" type="checkbox"/>	CreditCardPaymentInfo	number	0

At the bottom of this section is a blue button labeled 'Migrate'.

Figure: [Encryption Keys](#) page in the Migration tab.

If you did not configure your migration key in `project.properties` or `local.properties` file, then instructions on how to do it would display. Otherwise you can see below fields:

- Encryption Keys: List of the configured encryption keys
- Encrypted Attributes: All types with their encrypted attributes are listed. You see the total count of encrypted instance (#15) and how often every key is used (id=1, #15). Every selected row is encrypted with the new default key after clicking the [Migrate](#) button.

Tips and Pitfalls

Here are some tips and pitfalls related to TAE.

Encrypting Search-Relevant Values

As it's impossible to decrypt any values at run time, TAE shouldn't be used for fields that need to be searched for, for example first name or email address.

`dontOptimize=true`

Because you will not be able to search for encrypted values it could be a good idea to add the modifier "`dontOptimize=true`" in your `items.xml` for the encrypted attribute.

DB Persistence Type

Encrypting a string value increases its length, so be sure that you have chosen the right persistence type (**VARCHAR**, **CLOB**, **TEXT**) for storing your encrypted values.

Cluster Configuration

For guaranteeing the integrity of your data, you have to be sure that every node is using the same encryption key.

Since version 4.03 a general default key will be part of the release and is stored at
\$platformhome/bin/platform/ext/core/resources/security/default-128-bit-aes-key.hybris.

But we highly recommend, that you replace this weak and unsecure keyfile by your own one, which has to be placed in
\$platform_config_dir/security/.

Related Information

[Administration Console](#)

User Account

SAP Commerce Platform enables you to manage the user account and its security with special properties.

Preventing Password Brute-Force Attacks

It is possible to set a maximum number of unsuccessful login attempts in SAP Commerce to prevent brute-force attacks. After exceeding this number, the user isn't able to log in to the system anymore.

You can set the maximum number of unsuccessful login attempts on per-group basis. If the user belongs to many groups that have this property set to different values, the minimum (most strict) value is used.

Configuration

You can set your maximum number of unsuccessful login attempts for a given group in Backoffice, in the  User Groups tab. After you select your target group, you have access to the **Max brute force login attempts** property field in the Administration tab.

Logs

When a user from a target group enters invalid credentials on the login screen, the following log is recorded:

```
INFO [hybrisHTTP36] [DefaultUserAuditLoginStrategy] user:foo failed 1 logins. Max failed logins all
```

and if they exceed the maximum number of login attempts:

```
INFO [hybrisHTTP37] [DefaultUserAuditLoginStrategy] user:foo has reached the max number of failed 1
```

As a result, the user cannot log in to the account anymore.

Details

The number of unsuccessful login attempts for each user is stored in the database in **BruteForceLoginAttempts** items. Those items are persisted between system restarts and work correctly in the cluster environment. When the number of

unsuccessful login attempts exceeds your configured value, the `User.loginDisabled` field is set to `true`. This prevents the user from logging in to the system until the value is reverted to `false`. In addition, a `BruteForceLoginDisabledAudit` item is created to mark the fact that the user account has been disabled because of too many unsuccessful login attempts.

You can customize this mechanism by implementing `de.hybris.platform.servicelayer.user.UserAuditLoginStrategy` and registering a `userAuditLoginStrategy` bean.

When disabling user accounts, the configuration also takes into account the number of unsuccessful authentication attempts in OAuth flows where user credentials are provided in request parameters.

Access tokens generated for user accounts disabled as part of brute force attack prevention can no longer be used to access resources as they get deleted from the database. The credentials of disabled users can also no longer be used in OAuth flows, which means that it's impossible to issue a new access token or refresh an existing one by using these credentials in the request. To get a new access token for disabled user accounts, re-enable the accounts first.

For disabled user accounts, every new request for an access token, even with correct client and user credentials, fails with a 400 HTTP "Bad request" error response:

```
{
  "error" : "invalid_grant",
  "error_description" : "User is disabled"
}
```

Deactivate User Accounts

The `User.deactivationDate` property allows you to deactivate a user account. If the deactivation date you set is earlier than the current date and time, you deactivate the account instantly. If the deactivate date is in the future, the account is active up to this date and time, and then it gets deactivated.

Re-Enabling User Accounts

User accounts that have been disabled can be re-enabled through Backoffice in the [Password](#) tab for each user.

Client and User Brute Force Attack Prevention

Brute force attack prevention is configured and performed separately for users and OAuth clients. However, in OAuth flows that require both an OAuth client and user credentials, user authentication is not performed in case of unsuccessful client authentication and brute force attempts are counted only for the client, not for the user.

For information on client brute force attack prevention, see [OAuth2](#).

Users in Platform

Users and user groups in SAP Commerce all descend from the generic `Principal` type that is the foundation for all other user-related, more specific sub-types.

The `Principal` type is a base for the `user` type and, indirectly, the `usergroup` type. These are the starting points for you to use factory default `user` and `usergroup` accounts or to create your own user accounts and usergroup accounts.

You can create your own **employee** sub-types to reflect the company structure and the roles your employees play or structure. You can also create accounts for customers, or create the front end application that let your customers create and manage their own **customer** accounts.

Overview on Principals

Principal is the main abstract class for **user** and **usergroup** types. However, in common informal use both user and user groups are also referred to as principals. The following diagram gives an overview of **Principal** types in SAP Commerce.

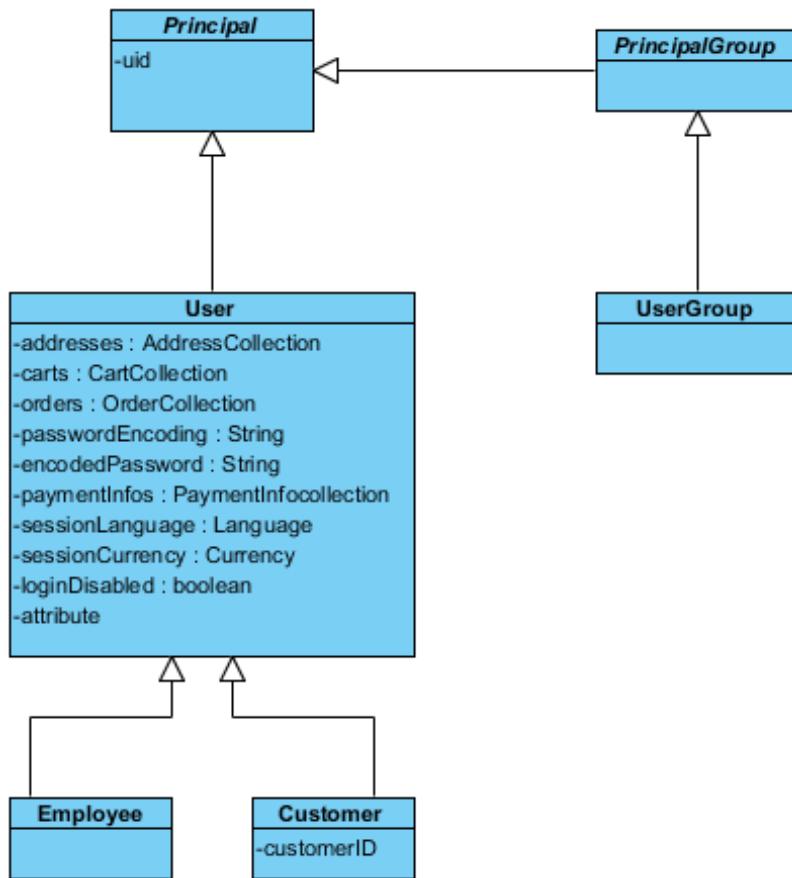


Fig. Principal types hierarchy

i Note

Please, bear in mind that the information presented here may differ when it comes to the Backoffice Application. To know more, see [Business Roles in Backoffice](#)

Unique Identifier

In the **Principal** class you can see **uid**. This stands for Unique Identifier that must be defined for every single **user** and **usergroup** item. The main purpose for this is to provide a differentiation factor for users and groups, and to identify each item. As a result, the **uid** must be **globally unique** for all types that descend from **Principal**. This means you cannot have the user and group with the same name. Also, as attribute **uid** is unique both for customers and for employees, employee and customer cannot have the same identifier.

System Accounts

There are three special system user items, that are essential to the platform and cannot be modified nor deleted. These are:

- **employee**: admin
- **customer**: anonymous

- **usergroup:** admingroup

Both the **anonymous** and **admin** users are crucial for the internal processes of SAP Commerce. Therefore, both accounts are protected from removal and against renaming. SAP Commerce blocks all attempts of removing or renaming these user accounts.

You can also add other users to **admingroup**. Those users would have the same rights as **admin** user. The only difference is that the Rule Framework is evaluated for all users, while it is not evaluated for **admin** user, as administrator has full access rights to everything within the system by default.

i Note

Please, bear in mind that the information regarding users may differ when it comes to the Backoffice Application. To know more, see [Business Roles in Backoffice](#)

Security

The security system concept in SAP Commerce is based on permissions that can be assigned to users and user groups, as well as global unique identifiers used to identify and authenticate users. Additionally, encryption mechanisms are in place to protect the user passwords. Using available tools you can create the complex security system based on, for example:

- Type related permissions applied to catalogs, products
- Roles of your employees and restrictions

Users

You can use the **user** type to create more specific types that define accounts for your employees and customers. Technically it is also possible to create an item of a **user** type, however from functional point of view it is pointless and should be avoided.

customer Type

Extended from a **user** type, the **customer** type comes with SAP Commerce. This is designed to be used for customers who visit your front end application like web shop, create their accounts for placing orders, manage their profiles, add and modify payment informations, and address data. Depending on your business context customers can create their own accounts, and have full or limited access to all their data.

Example of such situation can be when **customer** creates a new account, adds name and other information together with address and payment information. By creating a new account user also creates **uid**, that is a unique login, for further authorization and authentication purposes. As a result customer can modify the name or family name, address data or payment information, however, the login, the global identifier remains the same. You may configure your front end application to allow users changing data or to prevent them from modifying certain information.

Crucial **customer** data for ordering process, like address or payment information, are kept safe by being duplicated to the **Order** item. When customer makes an order, the delivery address and payment information are copied to the **Order** item and are stored separately to secure that the shipment arrives at the correct address and is correctly paid. Even, if a customer changes payment information or delivery address after the order was made, the shipment is paid and delivered according to the information stored with the **Order** item.

You can also use customer data to provide a support for localized service in your front end. For example, when a customer logs on to the system and starts a session, you could provide a localized version of the page for your customer.

The **customer** type has one extra attribute not present in other types descending from Principal: **customerID**. This mainly serves as an example how the **customer** type can be extended with new attributes to better reflect your business approach for collecting, using, and storing customer data.

Customers can be member of:

- **Usergroup:** For example, cockpit usergroup, VIP customergroup, frequent buyer group
- **Tax Group:** For details, see [europe1 Extension](#)
- **Price Group:** For details, see [europe1 Extension](#)
- **Discount Group:** For example, discount 5%. For details, see [europe1 Extension](#)

Customers can be affected by:

- **Restrictions:** For details, see [Restrictions](#).
- **Access Rights:** Permissions services framework defines access rights for users to the catalogs and other content. For details, see [Access Rights](#).

i Note

Customers are not Allowed to Manage SAP Commerce.

However, it is technically possible to grant **customer** account access to SAP Commerce management tools, this is not recommended and should not be allowed.

By factory default, SAP Commerce has one customer: **anonymous**. Other customers may be created as the need emerges. You can let the customers create their own account through the front end application, or you can do it by yourself, manually or by using ImpEx script to automate the process. For details on ImpEx, see [ImpEx Import - Best Practices](#).

employee Type

Extended from **user** type. This user can usually do the same as the **customer**. Additionally, the **employee** user can have access to the back office management tools and can perform several management actions, defined by the system of permissions and other regulations.

User **employee** is a representation of a member of your company. This account is used to - depending on assigned permissions - manage objects in SAP Commerce.

The special case of employee user is **admin**. This account cannot be removed, renamed, or restricted. Access rights framework checks if the user is **admin**, if yes, then no evaluation rule is checked to defined permissions for **admin**. User **admin** has access to everything within SAP Commerce. The **admin** user is special in that extent that it is member of the **admingroup** user group. This groups has no factory default limitations, nor restrictions.

You can create other employee users later as the need arises, manually or by using ImpEx scripts. For details on ImpEx, see [ImpEx Import - Best Practices](#). Also, by installing new cockpits, you also receive some factory default users that come together with particular cockpit extensions.

Employees can be affected by:

- **Restrictions:** For details, see [Restrictions](#) documentation.
- **Access Rights:** Permissions services framework defines access rights for users to the catalogs and other content. For details, see [Access Rights](#).

There are several factory default employee accounts that come with the SAP Commerce system. For few examples of such user accounts, see [Factory Default User Accounts](#).

User Groups

Unlike the user accounts, **usergroup** cannot be used to authenticate in SAP Commerce. The **uid** serves mainly as a group name. The group contain the collections of user accounts that belong to the group. The main purpose of that is to assign the permissions to the user group, that are further inherited to all group members. This enables easier permissions management, as they do not need to be assigned to particular user but to the set of users groups by the user groups.

Member of a user group inherits all settings from that user group, unless explicitly overridden. That way, you can combine permissions to the types, items, and attributes for a large number of users in one single place. All the users who are members of a certain user group will be affected by these settings. If the exception is needed, you can than override groups settings by assigning specific permissions to the user account in the group.

Groups can also contains other user groups. This let you build the complex hierarchical structures for easier management of your users and better reflecting of your company structure.

For a basic discussion of how inheritance with user groups in SAP Commerce works, see [Access Rights](#) documentation.

company Type

The user group **company** is a particular case of a **usergroup** type that by factory default comes together with SAP Commerce. This is a representation of commercial entity. You can add here other user groups and **employee** accounts in order to cover your company structure in the required scope.

Profiles and Roles

Customer can be defined by a broad set of data like name, address, age, title, language, sex, etc. There are some attributes in the **customer** type that are factory default. However, you can also extend the type in order to store more details about your customers and let them create personalized profiles. Such extended profiles may serve as a digital representation of your customer personality enabling you better understanding of customers behavior, needs, requirements, and expectations. Customer profile can help you to identify some characteristics about people who actually come to your web shop, create customer accounts, visit several pages, browse products, fill in the carts, and eventually make orders. The knowledge acquired from user profiles can be applied to predict the preferences of certain customer or customer groups. This is the step toward personalization of interactions between the customers and your company.

Advanced Personalization Module can serve as an example, where such kind of information can be very useful to enrich your business relations with customers. By extending the customer type with new attributes to hold customer data, you can collect information about your target groups for marketing and sale purposes. Depending on the customer group certain customer qualifies to, you could display different content, special offers, or hide certain areas of your web shop in order to prevent customers from accessing them. Static user profiles based on the data provided by customers can be extended with the information based on the customers behavior and shopping history in order to better adapt to customer needs and provide additional tools like suggestions or preferences.

Employee can also be treated as a personalized unit. Unlike the customer, the data related to the employee is rather back-office oriented and describes the administrative role that the employee plays within SAP Commerce. There is no way to predict those roles, as they depend on your specific business context. However, the SAP Commerce system is very flexible and allows you to create several employee accounts, assign different permissions to them or even combine permissions for **employee** users and **company** user group on an item or attribute level. Additionally, there are several factory default employee accounts that come with several cockpits, for details see the [Overview of Factory Default Employee Accounts](#) document.

Related Information

[ServiceLayer Security](#)

[Restrictions](#)

[Access Rights](#)

Managing Users and User Groups

Users and **user groups** in SAP Commerce have the same origin: **Principal**. They can be created, updated, and removed.

The process is similar for both users and user groups. There are some issues to take into consideration, such as uniqueness of the **uid**, or **system accounts** that cannot be removed.

Working with Users and User Groups

If you need to work with the user accounts, you can create them by instantiating a **UserModel** and saving it with the **ModelService**. For details, see [Models](#).

For most actions, you need to get the unique user identifier, **uid**, which also serves as unique login for users.

```
UserModel getUserForUID(String userId);
<T extends UserModel> T getUserForUID(String userId, Class<T> class);
```

The **uid** is usually used as the login for a user. If you know the user's **uid**, you can check if the user of the specified login already exists in the system.

```
boolean isUserExisting(String uid);
```

You can get the user group by a specific **uid**, which in this case is the unique name of the group, **getUserGroupForUID**. You can have many groups in your system defined. The groups may belong to other groups. Also, they may contain other groups as well. This is how to create a hierarchical tree structure. Normally, a user is a leaf in such a structure. However, the group not containing other groups can also be considered a leaf.

If you want to check the groups to which a user belongs, you can use the following methods:

```
Set<UserGroupModel> getAllUserGroupsForUser(UserModel user);
getAllUserGroupsForUser(UserModel user, Class<T> class);
boolean isMemberOfGroup(UserModel member, UserGroupModel groupToCheckFor);
boolean isMemberOfGroup(UserGroupModel member, UserGroupModel groupToCheckFor);
```

You can specify a **Title** for a user. For example, Mr., Ms., or Dr. The **Title** itself is a separate type. Because it is a minor type related to the user and does not usually change, the **Title** is also handled in the **UserService** by the **getTitleForCode** method.

There is a set of methods to help you in checking the system user accounts, such as **admin**, **anonymous**, and **admingroup**.

```
EmployeeModel getAdminUser();
UserGroupModel getAdminUserGroup();
CustomerModel getAnonymousUser();
boolean isAdmin(UserModel user);
boolean isAnonymousUser(UserModel user);
```

To check if a user is an admin or has the administrator right, which it inherited from the **admingroup**, you can generate a list of all **admin** users or check all groups for the specific user to verify if any of them is **admingroup**.

You can use **UserService** methods to check the user in session and then, based on the result, to perform certain actions, for example:

```
UserModel currentUser = userService.getCurrentUser();
if(userService.isAnonymousUser(currentUser))
{
```

```
System.out.print("You need to login to proceed");
}
```

Unique Identifiers for Users and User Groups

Users and user groups are identified by a unique identifier, a **uid**. The **uid** is an attribute of the **Principal** class; it is used in any type that extends **Principal**. The **uid** is used to uniquely identify all principals in the SAP system. It is used to identify users such as **customer** and **employee**, or user groups such as **company**.

The **uid** must be unique. It cannot be duplicated. You can avoid duplicating a **uid** by adding the suffix "group" to every **uid** that is used in the **usergroup** type, for example, **admin** and **admingroup**.

Users

Creating Users

Creating user is a process similar to creating any other model instance in the service layer architecture. To create a user, you call specific methods, as shown in the following code samples.

- Create an employee user

```
final EmployeeModel empl = new EmployeeModel();
empl.setUid("test");
modelService.save(empl);
```

- Create a customer user

```
final CustomerModel cust = new CustomerModel();
cust.setUid("test");
modelService.save(cust);
```

In the previous examples, we used the user called test. However, the user **uid** that serves as login must be globally unique within the system. You cannot have a **customer** and **employee** with the same **uid**. If a **uid** is already used, the system throws a model exception and you receive information about **uid** duplicate similar to the example shown below. You must change the **uid** and save it again.

```
de.hybris.platform.servicelayer.exceptions.ModelSavingException:  
ambiguous unique keys {uid=test} for model CustomerModel (<unsaved>) - found 1 item(s) using the sa
```

How to name the methods depends on your business context. For example, your customers can create new accounts by visiting your web shop. In the web shop, you can implement a business application that:

- Handles customer data
- Calls UserService on the SAP Commerce side
- Creates an account for a customer

During this process, the customer is prompted to provide a unique **login** that serves as **uid**. If the login is already taken by another user, the SAP system returns the relevant information. You can use this information to display a more meaningful message to your user and prompt the user to pick another login. The login could be a string or email address. You can also assign specific permissions to anonymous users that allows them to view selected parts of your content or allows them to perform actions that do not require a unique login.

Generating uid with Key Generator

Another method of obtaining a unique identifier for your customers within the system is to use a number series key generator. This method creates a unique customer ID for each customer. To use the key generator, go to the **servicelayer/user-spring.xml** file and enable the following beans:

- **customerIDGenerator**: This is the key generator that creates the key for the user found in the **customer_id** field. In the code example, it defines an eight-digit alphanumeric key, starting from 00000000. To generate a key with only numerical characters, you must set the **numeric** property to **true**.
- **customerIDPrepareInterceptor**: This prepares the interceptor. The interceptor is activated only for the object with the number that is defined as the uid for the user.
- **InterceptorMapping**: This connects the interceptor with the particular customer. For more information on interceptors, see the [Interceptors](#) document.

```
<!--
<bean id="customerIDGenerator" class="de.hybris.platform.servicelayer.keygenerator.impl.PersistenceKeyGenerator"
      init-method="init" >
  <property name="key" value="customer_id"/>
  <property name="digits" value="8"/>
  <property name="start" value="00000000"/>
  <property name="numeric" value="false"/>
</bean>

<bean id="customerIDPrepareInterceptor" class="de.hybris.platform.servicelayer.user.interceptor.PrepareCustomerIDInterceptor"
      >
  <property name="keyGenerator" ref="customerIDGenerator"/>
</bean>

<bean class="de.hybris.platform.servicelayer.interceptor.impl.InterceptorMapping" >
  <property name="interceptor" ref="customerIDPrepareInterceptor"/>
  <property name="typeCode" value="Customer"/>
</bean>
-->
```

Updating Users

Customer Type

Action	Description
Updating Address	The UserModel, as well as the Cartmodel and the OrderModel, can have an address. You use the AddressService to create the address for a user. If the customer creates a cart, the address is cloned to the cart related to the customer. If the cart changes to order during a later step, then the order also keeps the user address. The address in the cart and order are copies made by cloning the customer address value. This can be useful if, for example, a user places the order, but later changes the address before the order is placed. In such a situation, the shipment from the order is to be delivered to the address created when the order was created.
Updating Payment Info	Each customer has his/her own PaymentInfo . It holds the payment details for user.
Saving Carts	For a specific user, you can save the cart-related data. This allows the user to stop browsing the web shop any time and return later to continue from the point the user left the web shop.
Deactivating User	You can modify the loginDisabled attribute to deactivate the user account. SAP recommends that you deactivate user accounts instead of removing them, as it is possible that there are still orders holding the address for the users. Once you have removed a user

Action	Description
	account, there is no way to link the existing order with the user who placed it. To deactivate a user, use <code>user.setLoginDisabled(true);</code>

Employee Type

Action	Description
Changing User Roles	<p>Users of the employee type can play different roles within a company. You can define roles by creating specific user groups and assigning users to them. To change the role of your employee, you remove the user from one group and assign it to another group.</p> <p>For example, by assigning a user to the admingroup, you define administrator as the role for this user . From a functional point of view, such a user has the same rights as an admin user. As one group can contain many users, and one user can belong to many groups, it is possible to define several roles for one user.</p>

Removing User

You can remove any user account, except system users such as **admin** and **anonymous**. Removing a user account is done in the same as any other model. To remove a user account, you must first get the current user and then use a method that removes the user account.

```
UserModel user = UserService.getUserForUID("theUserId");
ModelService.remove(user);
```

If the user you want to remove is not found in the system, then the **UnknownIdentifierException** exception is thrown. If you try to remove a system user account, you also get an exception because you cannot remove a system user account.

Historical User Data

Once the order is created, the **Address** and **paymentInfo** are cloned from the user type to the order. Cloning makes a copy of the data. This is done because the user can modify the payment information after placing an order, but the order should use the address given at the time of placing the order. The same is valid for the information about payment: **paymentInfo**.

After a user has been removed, historical data related to user, order, and payment can remain in the system. For example, the user's address is stored in two locations. If you remove a user, you can have an order with an address but without a user name. This can happen because orders are not removed with the user. If the user is removed, then the order loses its reference to the user. This is why SAP recommends that you deactivate the user instead of removing the user account. Deactivated users are not automatically removed from the system. If you want to remove a deactivated user account from the system, you must do so manually.

User Groups

Usergroup types are similar to the **user** types. Both **user** and **usergroup** types are extended from the **Principal** type. You can create user groups the same way as you would create the users. For more information, see the API Documents.

Creating User Groups

User groups are containers that hold users and user groups. This allows you to create complex hierarchical structures. When you create user groups, you must ensure that no cyclic references are added in the user groups structures.

For example, you can create group A that contains group B, that holds group C. However, you cannot create a cyclic reference in which group A contains group B, that contains group C, that contains group A.

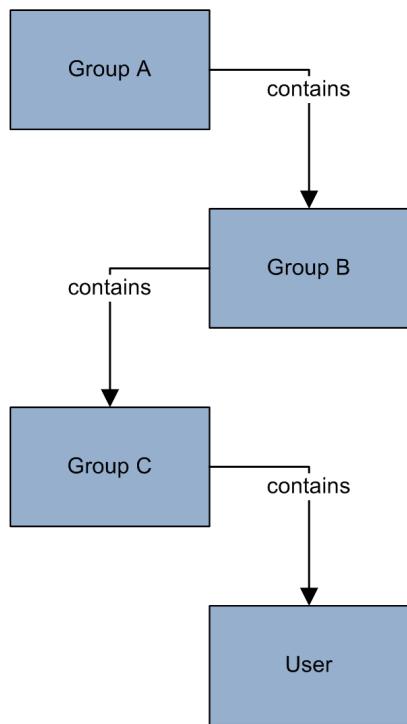


Fig. User Group Structure Allowed

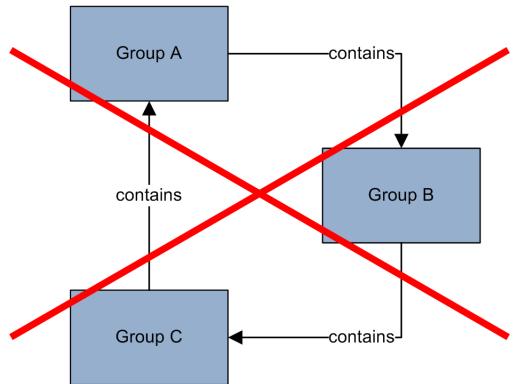


Fig. User Group Structure Not Allowed

The purpose of a user group can be:

- **Roles:** With the group hierarchy you can have a representation of the roles that your users play in the system.
- **Permissions management:** You can use the user groups to assign access rights to groups of your employees. User groups make it easier to assign and remove access rights to a number of users.
- **Marketing targeting:** You can define your target groups and assign users to the groups that reflect this kind of assignment. Customers could then have a different type of access to particular areas of your web shop content. Depending on the target group, you could show or hide the content from some customer groups.
- **Different access layers:** Generally, you can use the groups to separate those users who can perform management functions like **employee** users from those who can access your web shop without any management rights like **customer** users.
- **Cockpit customization:** Several cockpits come with the pre-defined user accounts that can have a specific access right scope to the particular cockpit management functions.

- **Catalog visibility:** With the assigning specific permissions to the different groups you can allow your users to see different catalogs, different products or different catalog versions depending on several criteria. For example, you can gather all users from a particular country into one group and display the product offer that is specifically designed for that group.

Updating User Group

You can update the attributes of the **usergroup**. For example, you can add users to the group or remove users, or change other attributes. Updating the usergroup can be done the same way as updating any other model in the Service Layer architecture. You need to get the user by the **uid**, and then proceed with other actions depending on what kind of information you want to update. For details on the operations on models, see [Models](#) documentation.

```
UserGroupModel usergroup = UserService.getUserGroupForUID("theUserGroupId");
```

It is also possible to change the **uid** of the user group. As the **uid** is the same as the user group name, it just means that you change the name of the user group.

Removing User Group

It is not possible to remove the system user group: **admingroup**. Assuming you have the sufficient permissions assigned you can remove all other groups easily:

```
UserGroupModel usergroup = UserService.getUserGroupForUID("theUserGroupId");
ModelService.remove(usergroup);
```

Related Information

[Removing Model Instance](#)

[Creating Model Instance](#)

[How to Display a User](#)

[Models](#)

Factory Default User Accounts

There are several factory default user accounts that come with the SAP Commerce system.

The following table list all the default user accounts included in SAP Commerce.

Factory Default Employee Accounts

User Name
aarav.devi@acme.com
aaron.customer@hybris.com
abra.christensen@sapfsa.com
abraham.mclane@acme.com
acctmgra
acctmgrb
acctmgrc
acctmgrd

User Name
admin
adrian.kent@hybris.com
aiko.abe@sapfsa.com
akiro.nakamura@pronto-hw.com
akiro.nakamura@rustic-hw.com
aladdin.gentry@sapfsa.com
alan.martin@hybris.com
albert.decastro@acme.com
alda.kamaka@acme.com
alejandro.navarro@rustic-hw.com
alistair@hybris.com
amanda.smith@stateofrosebud.com
amelia.hill@acme.com
amos.adkins@sapfsa.com
analyticsmanager
anamaria.coots@acme.com
andrea.customer@hybris.com
andrew.customer@hybris.com
anette.customer@hybris.com
angelyn.lobaugh@acme.com
anil.gupta@pronto-hw.com
anil.gupta@rustic-hw.com
annabel.golder@acme.com
anne.customer@hybris.com
anonymous
anthony.customer@hybris.com
anthony.lombardi@rustic-hw.com
antonio.ferrari@sapfsa.com
arjun.sewant@acme.com
arnold.customer@hybris.com
asagent

User Name
asagentmanager
asagentsales
aubrey.baxter@sapfsa.com
axel.krause@rustic-hw.com
ayumi.nakamura@acme.com
BackofficeIntegrationAdministrator
BackofficeIntegrationAgent
BackofficeProductAdministrator
BackofficeProductManager
BackofficeWorkflowAdministrator
BackofficeWorkflowUser
BedfordWarehouseAgent
BerlinDomWarehouseAgent
BerlinHospitalWarehouseAgent
BerlinMuseumWarehouseAgent
BerlinZooWarehouseAgent
bernard.customer@hybris.com
bernardo.coelho@acme.com
blossom.welch@sapfsa.com
bobby.customer@hybris.com
brandon.leclair@acme.com
brian.customer@hybris.com
bridget.customer@hybris.com
burton.franco@sapfsa.com
burtonlover@hybris.com
byung-soon.lee@rustic-hw.com
CajonWarehouseAgent
calvin.citizen@stateofrosebud.com
cameralenslover@hybris.com
canonlover@hybris.com
carla.torres@rustic-hw.com

User Name
CarltonWarehouseAgent
carol.citizen@stateofrosebud.com
carson.shepherd@sapfsa.com
chandni.devaraju@acme.com
chelsie.steck@acme.com
chen.gao@acme.com
chris.rumple@ehost.com
christmascustomer@hybris.com
cmseditor
cmsmanager
cmsmanager-apparel-de
cmsmanager-apparel-uk
cmsmanager-electronics
cmsmanager-electronics-de
cmsmanager-electronics-eu
cmsmanager-electronics-euzone
cmsmanager-electronics-uk
cmsmanager-electronics-us
cmsmanager-powertools
cmspublisher
cmsreader-apparel
cmsreviewer
cmstranslator
cmstranslator-Annette
cmstranslator-Seb
colt.ford@sapfsa.com
csagent
customer.support.1@sap.com
customer.support.2@sap.com
customer.support.3@sap.com
customer.support.4@sap.com

User Name
customer.support.5@sap.com
customer.support.6@sap.com
customer.support.7@sap.com
customer.support.8@sap.com
customer.support@chicago.com
customer.support@ichikawa.com
customer.support@nakano.com
customer.support@sanfrancisco.com
CustomerSupportAdministrator
CustomerSupportAgent
CustomerSupportManager
cxmanager
cxmanager-apparel-de
cxmanager-apparel-uk
cxmanager-electronics
cxmanager-electronics-de
cxmanager-electronics-eu
cxmanager-electronics-euzone
cxmanager-electronics-uk
cxmanager-electronics-us
cxmanager-powertools
cxuser
cxuser-apparel-de
cxuser-apparel-uk
cxuser-electronics
cxuser-electronics-de
cxuser-electronics-eu
cxuser-electronics-euzone
cxuser-electronics-uk
cxuser-electronics-us
cxuser-powertools

User Name
dagmar.fischer@acme.com
daisy.smith@irc.uk
dan.cameron@siteb.com
daniel.customer@hybris.com
daniele.sorber@acme.com
darrin.hesser@acme.com
deacon.fuller@sapfsa.com
dean.barton@sapfsa.com
debera.spiller@acme.com
DefaultWarehouseAgent
diana.best@sapfsa.com
dietrich.brand@acme.com
dionne.siguenza@acme.com
dipti.customer@hybris.com
diptiman.customer@hybris.com
donna@moore.com
dorthy.geoghegan@acme.com
dot.cohan@acme.com
elena.bulav@internet.ru
elena.petrova@sapfsa.com
elizabeth.juhlin@acme.com
elizabeth.miller@stateofrosebud.com
emily.bennett@acme.com
etta.berg@hybris.com
eusebio.scharff@acme.com
evangeline.jefferson@sapfsa.com
fern.henline@acme.com
francie.wildman@acme.com
fsintegrationadmin
gi.sun@pronto-hw.com
gi.sun@rustic-hw.com

User Name
GlasgowWarehouseAgent
glen.hofer@acme.com
granny.citizen@stateofrosebud.com
h.williams@peabody.ca
hac_editor
hac_viewer
hanna.andresen@acme.com
hanna.schmidt@pronto-hw.com
hanna.schmidt@rustic-hw.com
harold.wine@asite.org
hedley.mayer@sapfsa.com
hermelinda.cusick@acme.com
homer.citizen@stateofrosebud.com
importmanager
InboundConsentTemplateUser
IndianapolisWarehouseAgent
indira.duffy@sapfsa.com
IntegrationAdministrator
IntegrationAgent
integrationapi_adminuser
integrationapi_createuser
integrationapi_viewuser
integrationmonitoringtestuser
integrationservicetestuser
isabella.jackson@acme.com
isha.customer@hybris.com
jack.smith@stateofrosebud.com
james.bell@pronto-hw.com
james.bell@rustic-hw.com
james.davis@acme.com
jamey.sowa@acme.com

User Name
jane.citizen@stateofrosebud.com
janetta.estep@acme.com
jason.citizen@stateofrosebud.com
JerseyWarehouseAgent
jill.citizen
joana.oliveira@sapfsa.com
joey.citizen
john.citizen@stateofrosebud.com
john.li@sapfsa.com
john.miller@sapfsa.com
john.russel@hybris.com
josephine.citizen@stateofrosebud.com
jules.hasson@acme.com
juliane.tickle@acme.com
kadeem.gamble@sapfsa.com
kai.ratliff@sapfsa.com
karleen.holub@acme.com
kathy.liu@sapfsa.com
keenreviewer0@hybris.com
keenreviewer1@hybris.com
keenreviewer10@hybris.com
keenreviewer11@hybris.com
keenreviewer12@hybris.com
keenreviewer13@hybris.com
keenreviewer14@hybris.com
keenreviewer15@hybris.com
keenreviewer16@hybris.com
keenreviewer17@hybris.com
keenreviewer18@hybris.com
keenreviewer19@hybris.com
keenreviewer2@hybris.com

User Name
keenreviewer20@hybris.com
keenreviewer21@hybris.com
keenreviewer22@hybris.com
keenreviewer23@hybris.com
keenreviewer24@hybris.com
keenreviewer25@hybris.com
keenreviewer26@hybris.com
keenreviewer27@hybris.com
keenreviewer28@hybris.com
keenreviewer29@hybris.com
keenreviewer3@hybris.com
keenreviewer30@hybris.com
keenreviewer4@hybris.com
keenreviewer5@hybris.com
keenreviewer6@hybris.com
keenreviewer7@hybris.com
keenreviewer8@hybris.com
keenreviewer9@hybris.com
keita.tanaka@acme.com
kelsie.spencer@sapfsa.com
kirti.customer@hybris.com
KotoWarehouseAgent
kritika.customer@hybris.com
lael.garibay@acme.com
lars.bauer@rustic-hw.com
latisha.latimer@acme.com
lavone.dupler@acme.com
LeedsWarehouseAgent
linda.wolf@pronto-hw.com
linda.wolf@rustic-hw.com
liu.yang@acme.com

User Name
lucas.kowalski@rustic-hw.com
mai.matsumoto@acme.com
manager
mara.martino@acme.com
marco.rossi@sapfsa.com
maria.stevens@hybris.com
marie.dubois@rustic-hw.com
marie.kempner@acme.com
mark.farrel@hybris.com
mark.rivers@pronto-hw.com
mark.rivers@rustic-hw.com
marketingmanager
marvel.fargo@acme.com
matheu.silva@rustic-hw.com
MatsudoWarehouseAgent
matthew.miller@stateofrosebud.com
matthew.zhao@sapfsa.com
maycustomer@hybris.com
men@hybris.com
menover30@hybris.com
menshortslover@hybris.com
menvipbronze@hybris.com
menvipgold@hybris.com
merchantcontentmanager
merchantproductmanager
merchantvendormanager
michael.adams@sapfsa.com
michael.barton@sapfsa.com
michael.clarke@sapfsa.com
miguel.rodriguez@sapfsa.com
mingmei.wang@pronto-hw.com

User Name
mingmei.wang@rustic-hw.com
MisatoWarehouseAgent
monique.legrand@sapfsa.com
MunichMuseumWarehouseAgent
NakanoWarehouseAgent
natsumi.takahashi@acme.com
nishi.customer@hybris.com
noah.jenkins@acme.com
oliver.baker@acme.com
olivia.ann@hybris.com
ossie.dilks@acme.com
patricia.anderson@sapfsa.com
pedro.dasilva@sapfsa.com
piedad.holdren@acme.com
powerdrillslover@pronto-hw.com
pradeepthi.customer@hybris.com
pritika.customer@hybris.com
productmanager
punchout.customer@punchoutorg.com
punchout.customer2@punchoutorg.com
ravi.pandey@sapfsa.com
RegApproverA
reggy.ray@hybris.com
revenueCloudCustomerApiUser
reviewer1@hybris.com
reviewer10@hybris.com
reviewer11@hybris.com
reviewer12@hybris.com
reviewer13@hybris.com
reviewer14@hybris.com
reviewer15@hybris.com

User Name
reviewer16@hybris.com
reviewer17@hybris.com
reviewer18@hybris.com
reviewer19@hybris.com
reviewer2@hybris.com
reviewer20@hybris.com
reviewer21@hybris.com
reviewer22@hybris.com
reviewer23@hybris.com
reviewer24@hybris.com
reviewer25@hybris.com
reviewer26@hybris.com
reviewer27@hybris.com
reviewer28@hybris.com
reviewer29@hybris.com
reviewer3@hybris.com
reviewer30@hybris.com
reviewer4@hybris.com
reviewer5@hybris.com
reviewer6@hybris.com
reviewer7@hybris.com
reviewer8@hybris.com
reviewer9@hybris.com
rewati.customer@hybris.com
richard.martin@acme.com
richard.wilson@sapfsa.com
richard@wilson.com
ronnie.ray@hybris.com
sade.mcdougall@acme.com
salome.levi@rustic-hw.com
sandesh.patwary@acme.com

User Name
sapCpqQuoteApiUser
sapInboundB2BCustomerUser
sapInboundB2CCustomerUser
sapInboundClassificationUser
sapInboundMDMB2CCustomerUser
sapInboundOMMOrderUser
sapInboundOMMReturnRequestUser
sapInboundOMSOrderUser
sapInboundPriceUser
sapInboundProductUser
sapInboundRCConfigUser
sapInboundSubscriptionPriceUser
sbgadmin
screwdriverslover@pronto-hw.com
searchmanager
SecondaryNakanoWarehouseAgent
selfserviceuser2@hybris.com
selfserviceuser3@hybris.com
selfserviceuser4@hybris.com
selfserviceuser5@hybris.com
selfserviceuser6@hybris.com
sheilah.duffin@acme.com
ShinbashiWarehouseAgent
shortslover@hybris.com
shun.watanabe@acme.com
stefan.bosch@sapfsa.com
summercustomer@hybris.com
surabhi.customer@hybris.com
TacomaWarehouseAgent
tag.demph@sapfsa.com
takahiro.suzuki@sapfsa.com

User Name
TampaWarehouseAgent
temeka.meekins@acme.com
teresa.citizen@stateofrosebud.com
teresa.ruiz@sapfsa.com
test-user-with-coupons@ydev.hybris.com
test-user-with-orders@ydev.hybris.com
thanksgivingcustomer@hybris.com
thomas.schmidt@sapfsa.com
tilda.prisbrey@acme.com
tim.james@hybris.com
tom.ziebarth@acme.com
TranslatorGSW
tualInboundSimpleProductOfferingUser
Tulsa1WarehouseAgent
Tulsa2WarehouseAgent
Tulsa3WarehouseAgent
Tulsa4WarehouseAgent
Tulsa5WarehouseAgent
ula.barragan@acme.com
ulf.becker@rustic-hw.com
ulysses.head@sapfsa.com
vada.rahm@acme.com
vendor1vendoradministrator
vendor1vendorcontentmanager
vendor1vendorproductmanager
vendor1vendorwarehousestaff
vendor2vendoradministrator
vendor2vendorcontentmanager
vendor2vendorproductmanager
vendor2vendorwarehousestaff
vendor3vendoradministrator

User Name
vendor3vendorcontentmanager
vendor3vendorproductmanager
vendor3vendorwarehousetaff
vendor4vendoradministrator
vendor4vendorcontentmanager
vendor4vendorproductmanager
vendor4vendorwarehousetaff
vendor5vendoradministrator
vendor5vendorcontentmanager
vendor5vendorproductmanager
vendor5vendorwarehousetaff
vendor6vendoradministrator
vendor6vendorcontentmanager
vendor6vendorproductmanager
vendor6vendorwarehousetaff
vendor7vendoradministrator
vendor7vendorcontentmanager
vendor7vendorproductmanager
vendor7vendorwarehousetaff
vendor8vendoradministrator
vendor8vendorcontentmanager
vendor8vendorproductmanager
vendor8vendorwarehousetaff
vipbronze@hybris.com
vipgiold@hybris.com
vipgold@hybris.com
vipsilver@hybris.com
vjdbcReportsUser
wang.lei@acme.com
WarehouseAdministrator
WarehouseAgent

User Name
WarehouseEAgent
WarehouseManager
WarehouseNAgent
WarehouseSAgent
WarehouseWAgent
wei.liu@homemail.ch
wfl_marketing
wfl_marketing_DE
wfl_marketing_EN
wfl_marketing_ES
wfl_marketing_FR
wfl_marketing_GB
wfl_marketing_GSW
wfl_marketing_IT
wfl_marketing_SWE
wfl_productApproval
wfl_productManagement
wfl_purchase
wfl_translator_DE
wfl_translator_EN
wfl_translator_ES
wfl_translator_FR
wfl_translator_GB
wfl_translator_GSW
wfl_translator_IT
wfl_translator_SWE
william.hunter@pronto-hw.com
william.hunter@rustic-hw.com
women@hybris.com
womenvipgold@hybris.com
womenvipsilver@hybris.com

User Name
xaviera.crawford@sapfsa.com
yan.shehorn@acme.com
yformsmanager
yoshie.dority@acme.com
yu.yamamoto@acme.com
yuka.kobayashi@acme.com
yuri.chandler@sapfsa.com
zhang.wei@acme.com

Related Information

[Users in Platform](#)

Visibility Control

You can configure SAP Commerce to allow or deny access to items, catalogs, and features. Some of these controls take effect in the front end, while others are specific to Backoffice Administration Cockpit.

There are three types of visibility controls in SAP Commerce:

- Those that are effective all across SAP Commerce, such as restrictions
- Those that are effective in a web front end only, such as product approval status
- Those that are effective in Backoffice only, such as access right settings

i Note

The following table only lists controls that are available in SAP Commerce by default. If you have built custom extensions, some of these controls may not take effect, or may work differently from what is described here.

Available controls in the shop frontend	Available controls in Backoffice
<ul style="list-style-type: none"> • Restrictions. • Category version visibility. • Product approval status. • Catalog version active attribute. <p>For more information, see Front End Visibility Controls.</p>	<ul style="list-style-type: none"> • Restrictions. • Category version visibility. • Backoffice access rights. • Usergroup-specific Backoffice configuration. <p>For more information, see Backoffice Visibility Controls.</p>

Restrictions

Restrictions affect Flexible Search results. If restrictions are in place, a user may not receive all search results that would be available without those restrictions.

Duration of Effect

Visibility rights are bound to a session. If you change a visibility setting and the change appears to have no effect, try closing the session and logging in again. Some visibility settings are valid for the entire session lifetime and are not affected retroactively by changes made when the session has already been created. For an overview of catalog-related setting lifetimes, see [Catalog Guide](#).

Visibility Control Checklist

Quickly find answers to problems related to the visibility of items in the storefront or Backoffice Administration Cockpit.

The following table lists a number of possible visibility issues and their potential causes. The potential causes are discussed in more detail in the related topics.

Problem	Location	Potential cause
Changes I have made seem to have no effect.	Front end, Backoffice	Session lifetime has expired.
Not all catalog versions are visible.	Front end	Restrictions are preventing visibility. The catalog version is not set to active. No category is visible for the current user.
No categories, or not all categories are visible.	Front end	Restrictions are preventing visibility. The user or user group has no read access to the category.
Individual product is not visible.	Front end	Restrictions are preventing visibility. The product status is not set to approved.
No catalogs are visible.	Backoffice	Restrictions are preventing visibility. Usergroup-specific Backoffice configuration is preventing visibility for this group. Appropriate Backoffice access rights are not granted for this group.
Not all catalog versions are visible.	Backoffice	Restrictions are preventing visibility.
An attribute is invisible or not present on the expected tab.	Backoffice	Appropriate Backoffice access rights are not granted for this group. Attribute has no explicit Backoffice configuration and is therefore moved to the Unbound section.
I get an "Attribute not readable" error.	Backoffice	Appropriate Backoffice access rights are not granted for this group.
Greyed-out editor.	Backoffice	Appropriate Backoffice access rights are not granted for this group.

Problem	Location	Potential cause
Greyed-out Delete button.	Backoffice	Appropriate Backoffice access rights are not granted for this group.
Greyed-out Create context menu.	Backoffice	Appropriate Backoffice access rights are not granted for this group.
An entry (node) in the Explorer Tree is missing.	Backoffice	Usergroup-specific Backoffice configuration is preventing visibility for this group. Appropriate Backoffice access rights are not granted for this group.
A tab is not showing. .	Backoffice	Usergroup-specific Backoffice configuration is preventing visibility for this group.
A section is not showing.	Backoffice	Usergroup-specific Backoffice configuration is preventing visibility for this group.
Non-default attribute editor.	Backoffice	Usergroup-specific Backoffice configuration is preventing visibility for this group.

Front End Visibility Controls

SAP Commerce includes tools to control the visibility of items for customers in the storefront.

If a customer can not see the full list of items you expect (only a number of products, for example), there may be an issue with either your custom extension, or a visibility setting. A good means of finding out whether there is a visibility issue or a source code problem is by assigning the **admin** employee to the session. For more information, see [Restrictions](#).

- If the full list of items appears, it is a visibility issue.
- If only a part of the list of items appears, it is not a visibility issue. Instead, the cause lies somewhere else.

If it is a visibility issue, you can check the following possible causes.

Catalog Version Visibility

By default, only the catalog version marked **active** is visible in the frontend.

Category Visibility

Each category has an attribute that specifies the users or user groups who are allowed to see and browse the category. You can find this under **Category Visibility** on the **General** tab in Backoffice Administration Cockpit, or **allowedPrincipals** in the type system). Users who are not specified for this attribute are not allowed to access the category. Hierarchically structured usergroups inherit visibility settings from their parent.

If no category of a catalog version is visible to a user, then the catalog version will not be visible either: Having a catalog version with no category displayed would result in a catalog version with nothing useful to display in the web front end.

Product Approval Status

Only products with an approval status of **approved** are visible by default in the front end. Products whose status is **unapproved** or **check** are not visible.

Backoffice Visibility Controls

Backoffice has settings to control the visibility of items and features to certain groups within Backoffice Administration Cockpit.

An admin user can restrict the visibility of Backoffice features and functions to employee users and user groups. Such restrictions have no effect on customer users or user groups.

Catalog Version Visibility

You can assign read and write permissions to a catalog version in the Permissions tab of the catalog version details in Backoffice.

A catalog version is accessible to a user if the user has read access to the catalog version. Whether the catalog version is active or not has no effect on this visibility. The user account who creates a catalog version is given read and write access by default. Read or write permissions must be set explicitly for all other users and groups.

Backoffice Access Rights

You can grant or deny user accounts access to Backoffice.

Usergroup Specific Backoffice Configuration

Backoffice allows setting up a specific configuration of features for individual user groups. This makes it possible to hide Backoffice elements from user groups, both in the Explorer Tree and in the Organizer. This does not affect access rights.

Unbound Attributes

Backoffice moves attributes that are not explicitly placed anywhere onto the **Administration** tab of the Type details. Such movement will result if no explicit configuration for the attribute was specified.

Related Information

[Restrictions](#)

[Catalog Guide](#)

[FlexibleSearch](#)

Web Application Endpoint Control

Platform allows you to disable endpoints within web apps. This functionality may be useful especially for REST-oriented applications.

The example below shows how to disable endpoints in a web app.

The example web app offers the following endpoints:

```
/platform
/platform/init
/platform/update
/platform/system
/console/impex/export
/console/impex/import
```

Assume that you want to set up Platform to permanently disable the /hac/platform/init and /hac/platform/update endpoints. You can do it through a configuration that uses appropriate properties.

The configuration property responsible for disabling endpoints uses the following naming convention:

`endpoint.extensionName.endpointPath.disabled=true|false`, where:

- `endpoint` is a constant part and must be always included
- `extensionName` is the name of the extension that provides the web app; in the example, it is the `hac` extension
- `endpointPath` is a dotted version of the endpoint path, for example, `foo.bar.baz` is the dotted version of the `/foo/bar/baz` path.
- `disabled` is a constant part and must be always included

Here is the property configured to disable the /hac/platform/init and /hac/platform/update endpoints:

```
endpoint.hac.platform.init.disabled=true
endpoint.hac.platform.update.disabled=true
```

i Note

Any configuration for this functionality is only read during system startup. It is not possible to change it during runtime.

ResourcesGuardService

`ResourcesGuardService` enables you to check whether an endpoint in a web app is configured to be disabled or enabled. It also enables you to manage configuration propagation for endpoint paths and subpaths. You can inject it through Spring by using the `ResourcesGuardService` ID.

Checking Whether Endpoints Are Disabled

`ResourcesGuardService` provides two public methods that enable you to check whether endpoints are disabled or enabled:

- `boolean isResourceDisabled(String extensionName, String resourcePath)`
- `boolean isResourceEnabled(String extensionName, String resourcePath)`

This example shows how to use them:

```
// assuming resourceGuardService was injected by Spring
boolean platformInitDisabled = resourcesGuardService.isResourceDisabled("hac", "/platform/init");
boolean platformUpdateDisabled = resourcesGuardService.isResourceDisabled("hac", "/platform/update"

// assertions
assertThat(platformInitDisabled).isTrue();
assertThat(platformUpdateDisabled).isTrue();
```

Propagating Configuration for Paths and Subpaths

By default, each configuration respects the parent-child hierarchy of paths and subpaths. It means that a configuration set for a given path propagates into all subpaths of that path.

For example, the `/platform` path has these three subpaths: `/platform/init`, `/platform/update`, and `/platform/system`. Disabling only the `/platform` path makes all the three subpaths disabled too - by default and without any further configuration:

```
endpoint.hac.platform.disabled=true
```

To prevent your configuration from propagating from a path into its subpaths, explicitly configure those subpaths. Also, configure `ResourcesGuardService` by setting the `endpoints.guardservice.respect.parents` property value to `false`.

For example, here are some available endpoints:

```
/platform
/platform/init
/platform/update
/platform/system
```

And here is some configuration:

```
endpoints.guardservice.respect.parents=false
endpoint.hac.platform.disabled=true
endpoint.hac.platform.update.disabled=false
```

As a result:

- `/platform/init` and `/platform/system` endpoints inherit the configuration settings from their top-level parent, `/platform`; these endpoints are disabled because their paths aren't explicitly configured otherwise
- you prevent the `/platform` configuration settings from propagating into the `/platform/update` subpath; the `/platform/update` endpoint is not disabled

ResourcesGuardFilter

You need a tool to use all the presented configurations and settings to actually make a particular endpoint disabled. Platform provides a new `Filter` class called `ResourcesGuardFilter`. You can inject it into the standard SAP Commerce filter chain. Configure it on the web app level as in this example:

```
<bean id="myWebAppResourcesGuardFilter" class="de.hybris.platform.servicelayer.web.ResourcesGuardFilter">
    <property name="resourcesGuardService" ref="resourcesGuardService"/>
    <property name="extensionName" value="myWebApp" />
</bean>

<bean id="myWebAppFilterChain" class="de.hybris.platform.servicelayer.web.BackOfficeFilterChain">
    <constructor-arg>
        <list>
            <ref bean="myWebAppResourcesGuardFilter"/>
            <!-- Possible other filters here... -->
        </list>
    </constructor-arg>
</bean>
```

By default, this filter sends the HTTP 404 code whenever it finds that a particular request URI matches a configured endpoint and that endpoint is disabled. It may also send a redirect to a particular page or endpoint if you configure it to do so, for example:

```
<bean id="myWebAppResourcesGuardFilter" class="de.hybris.platform.servicelayer.web.ResourcesGuardFilter">
    <property name="resourcesGuardService" ref="resourcesGuardService"/>
    <property name="extensionName" value="myWebApp" />
    <property name="reditectTo" value="/404.jsp" />
</bean>
```

Hiding Tabs in SAP Commerce Administration Console

If you disable endpoints that are URLs to Administration Console tabs, the tabs become invisible.

Considerations

The provided tools read a configuration and help you determine whether some endpoints in your web app are configured to be disabled. Your web app, however, may have, for example, navigation links to those endpoints on some jsp pages. You need to implement your own logic to use `ResourcesGuardService` if you don't want the end user to see such links.