



Platform, Services, and Utilities

Generated on: 2021-10-20 10:37:13 GMT+0000

SAP Commerce | 1905

PUBLIC

Original content: <https://help.sap.com/viewer/d0224eca81e249cb821f2cdf45a82ace/1905/en-US>

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/viewer/disclaimer>.

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.

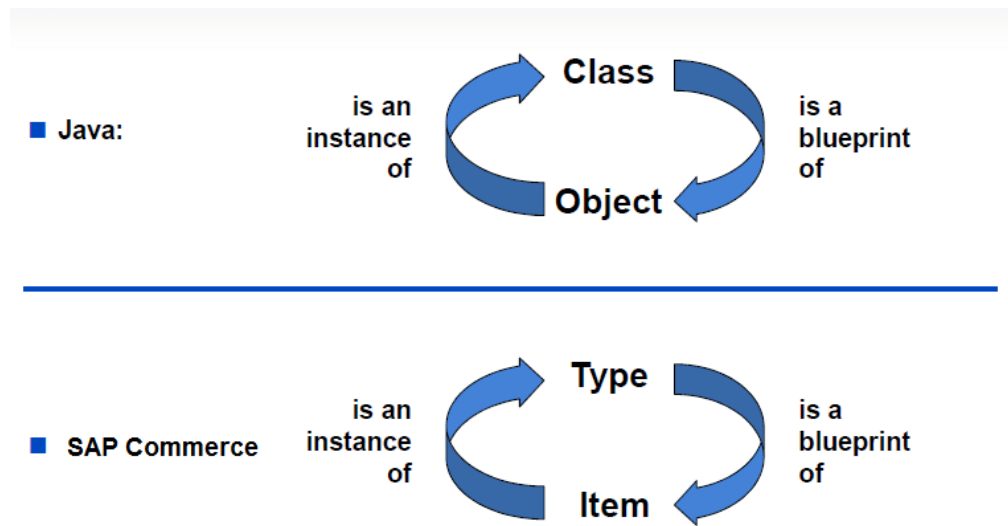
Main Functions of the Type System

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.

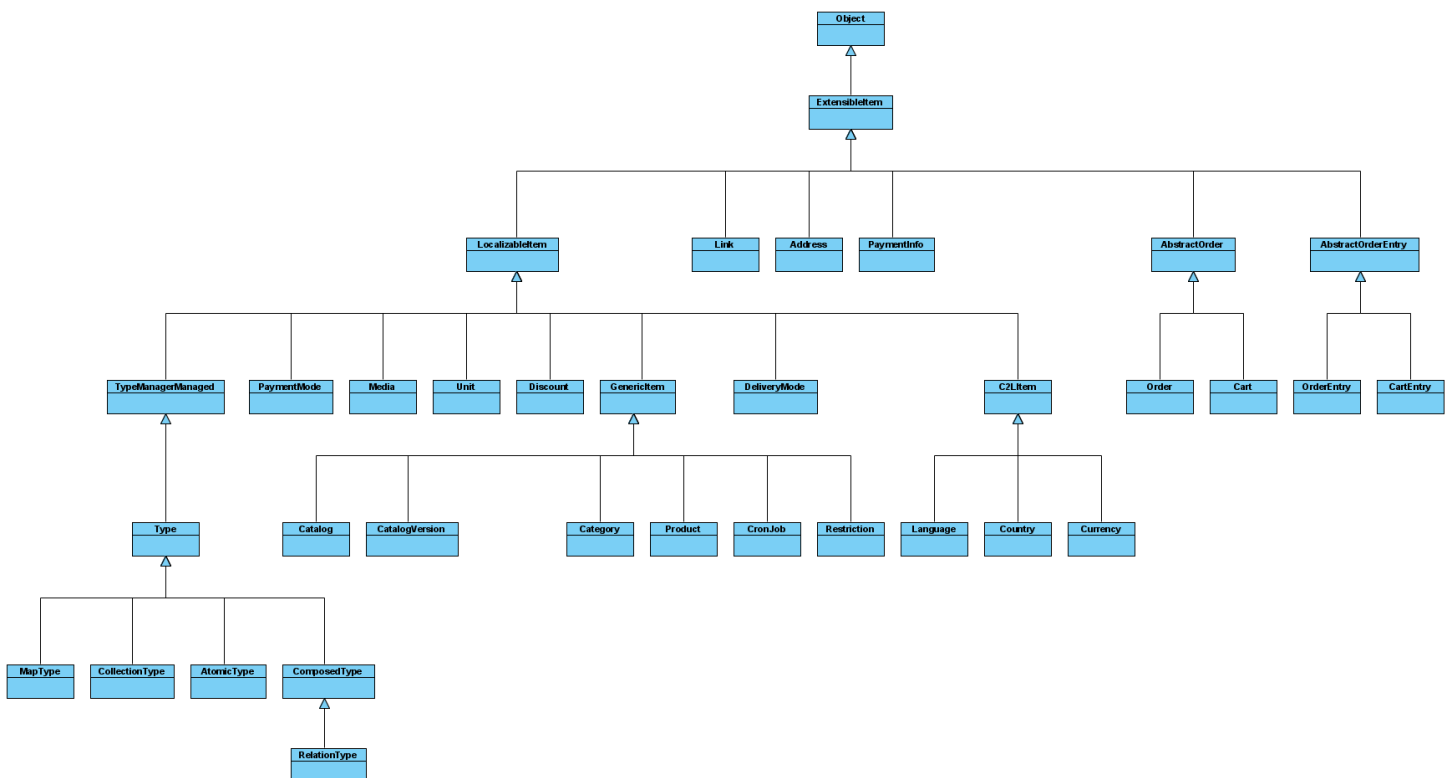
	Data Types	Infrastructure Types
Java:	<div>java.lang.Boolean</div> <div>java.lang.Integer</div> <div>java.lang.String</div> <div>java.util.Date</div> <div>java.util.Map</div> <div>java.lang.Enum</div>	<div>java.lang.Object</div>
SAP Commerce:	<div>java.lang.Boolean</div> <div>java.lang.Integer</div>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

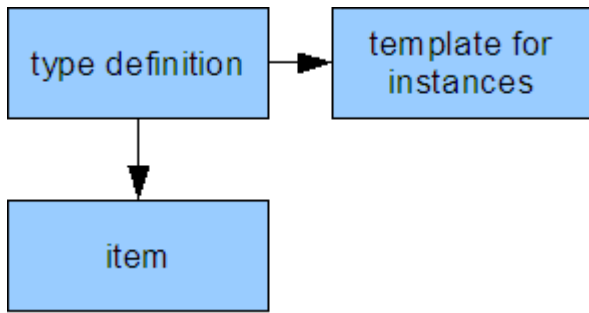
1. System-related types make up or extend the type system itself and deal with internal data management:

- 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.
- 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.

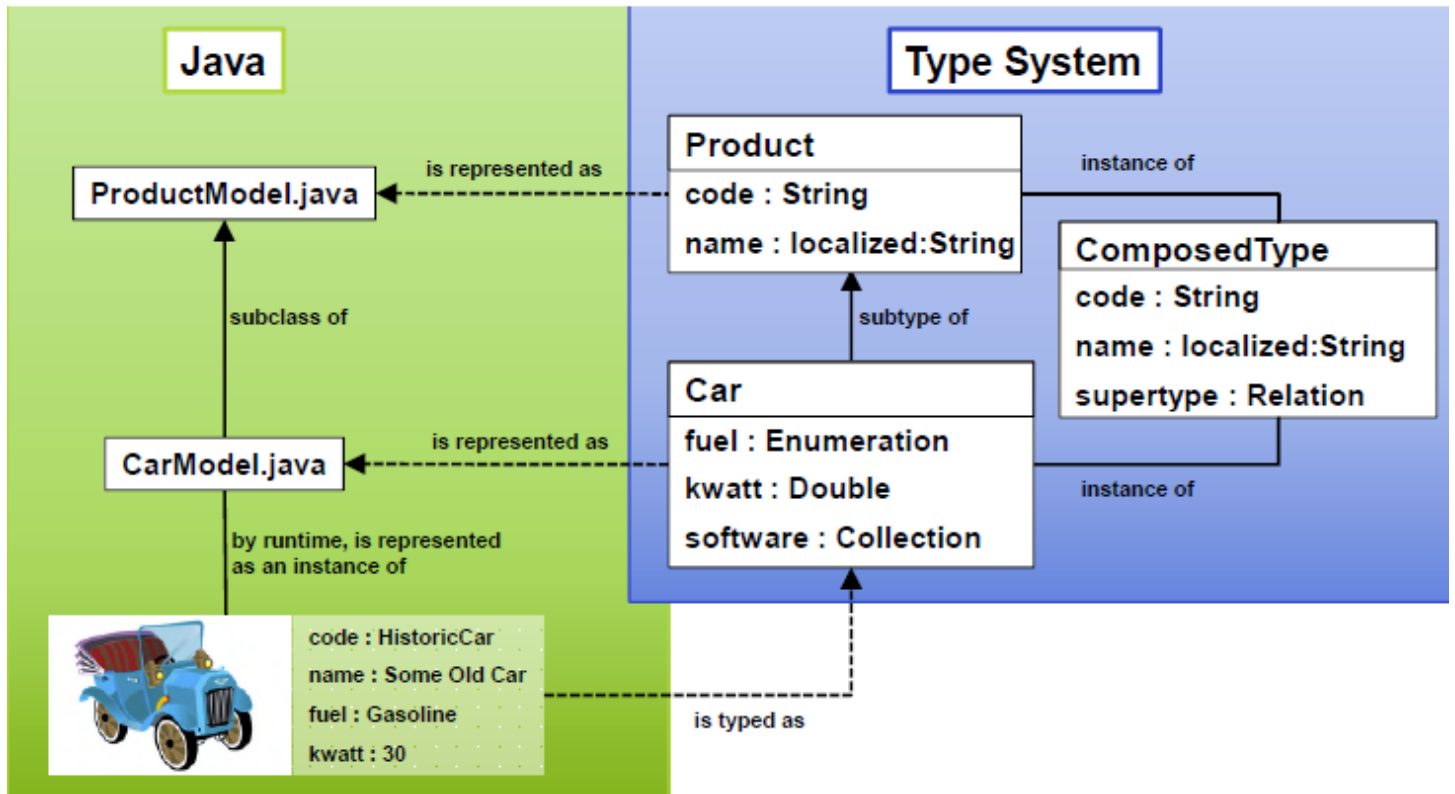
The following diagram gives you an overview on a part of the type hierarchy defined by an out-of-the-box SAP Commerce installation. We provide the diagram only as a general example that doesn't show details, and as an example reference. For the actual structure of the type system, see the `core-items.xml` 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`.

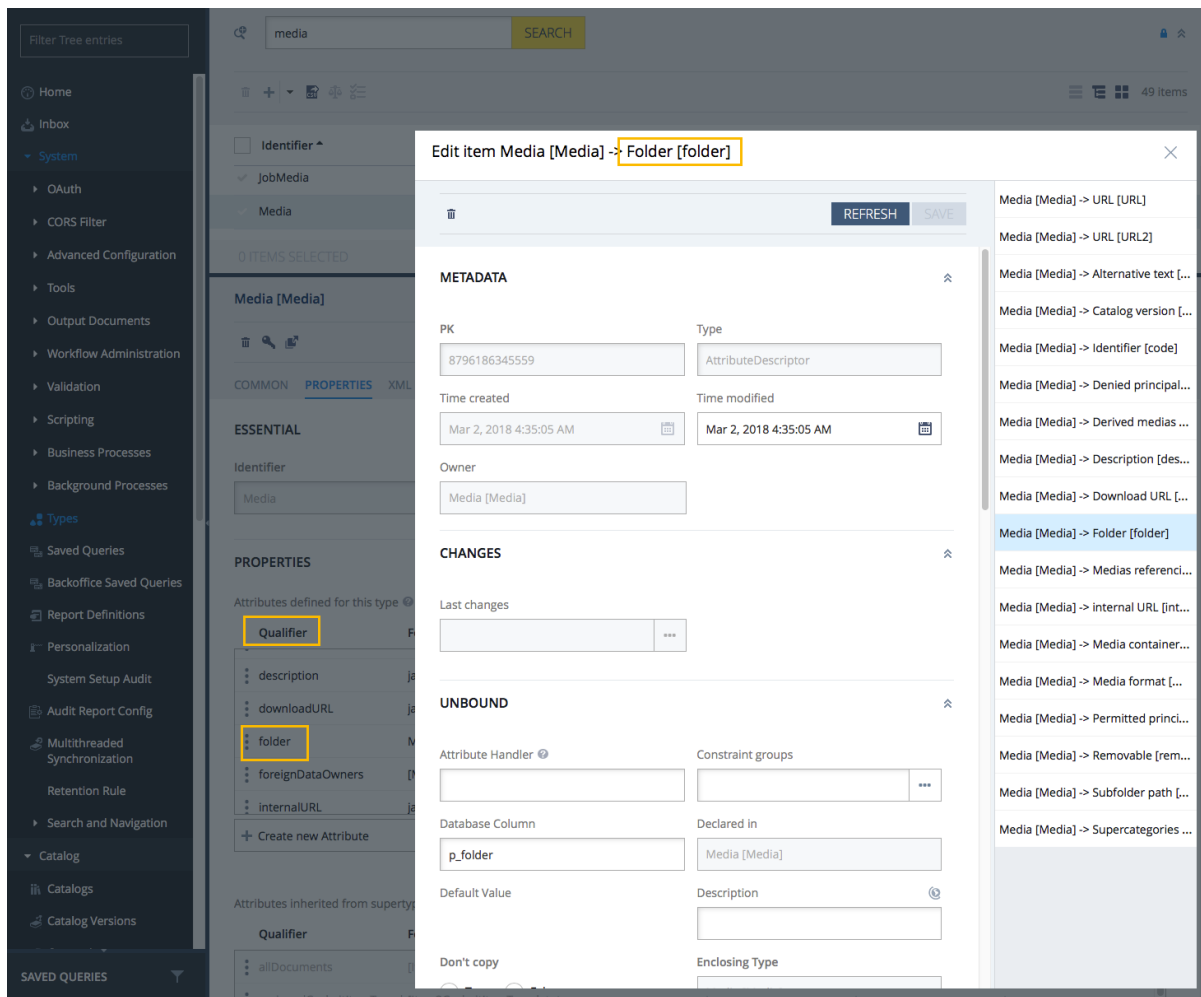
```
<item code="SpecialProduct"      metatype="SpecialComposedType"  extends="Product">
```

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.

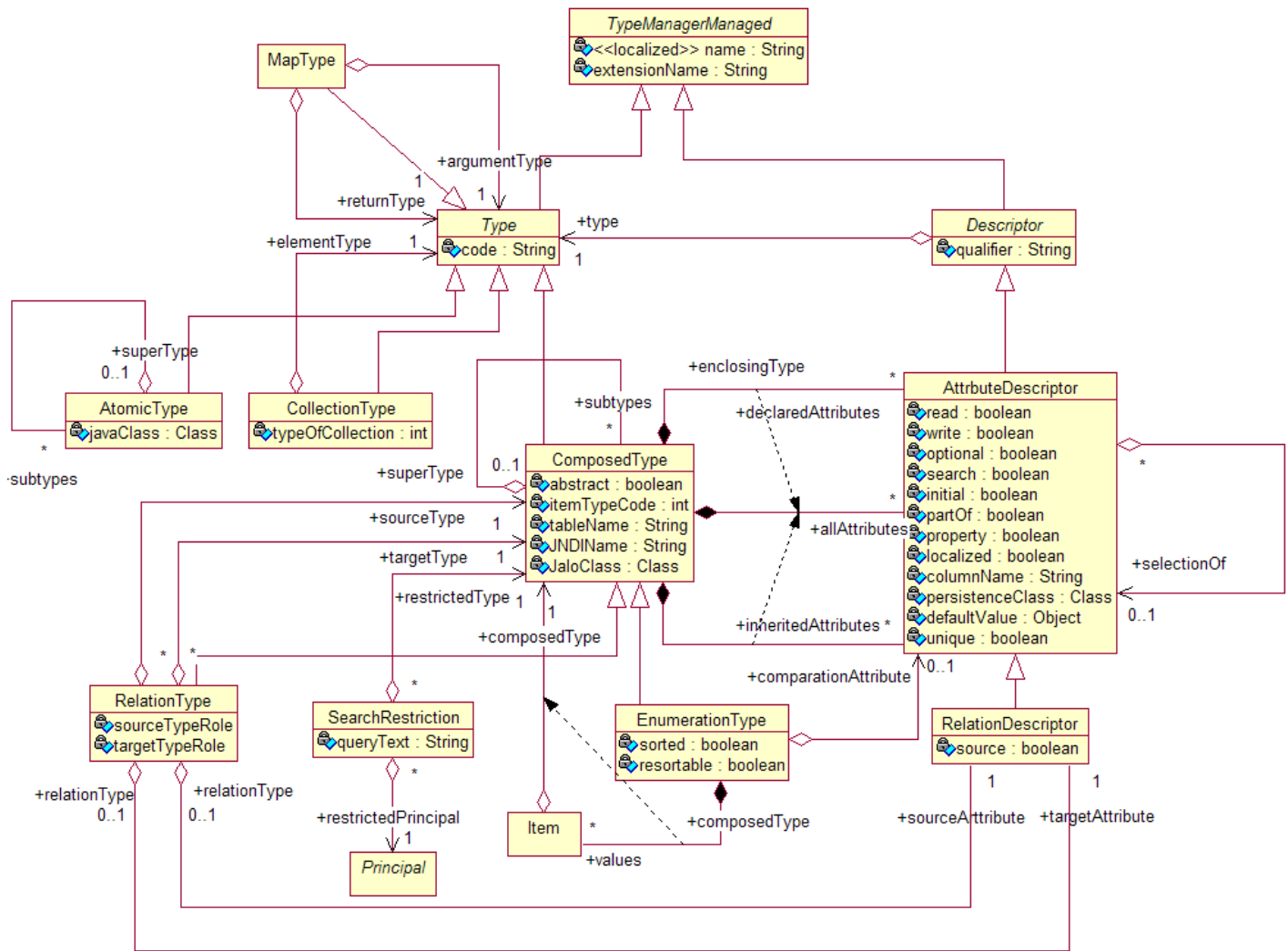


Configured Types vs Runtime Types

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`; they are only defined in runtime in Backoffice.

i Note

SAP strongly discourages using Runtime Types for production environments because of:

- 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.

Since Runtime Types have these technical limitations, SAP strongly recommends exporting Runtime Type definitions into an `items.xml` file.

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, please refer to the documentation about the `items.xml` file, [items.xml](#).

i Note

As mentioned before, SAP strongly discourages using Runtime Types for production environments. Whenever you initialize SAP Commerce, all Runtime Types and added attributes are gone.

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.

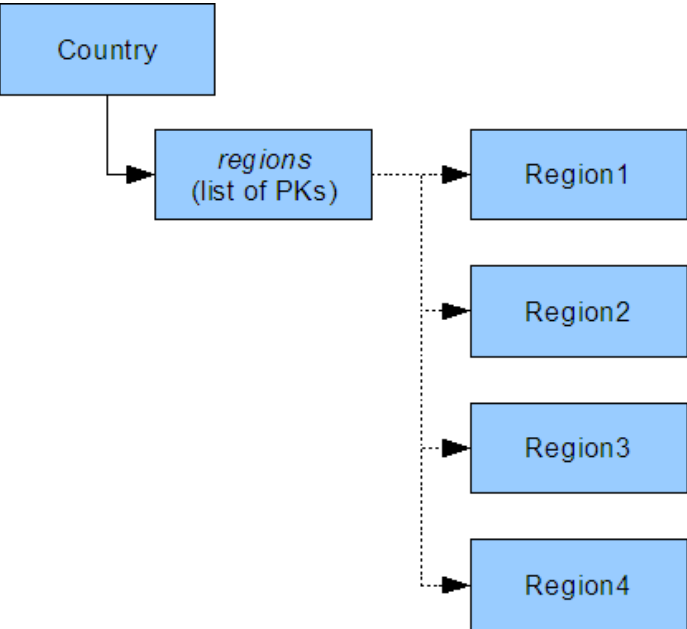
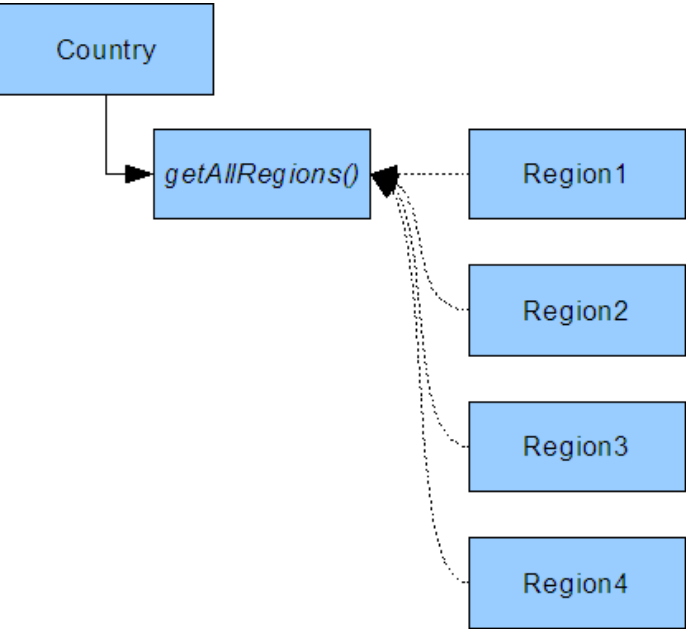
As you need to define the type of items that are stored in a `CollectionType` from the very start, it is impossible to store any other type of items in the same `CollectionType`. In the example code snippet, the `StringCollection` type is restricted to `String` type items only. SAP Commerce blocks any 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)
-----------------------------	-----------------------------

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>  <p>In the graphic, the Country type has an attribute regions 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> <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. 	<p>Store the attribute values at the respective target items and have a getter method at the source type to retrieve the values.</p>  <p>In the graphic, Country has a getter method called getAllRegions() that runs a FlexibleSearch statement on the database to find all instances of Region.</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. • 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.

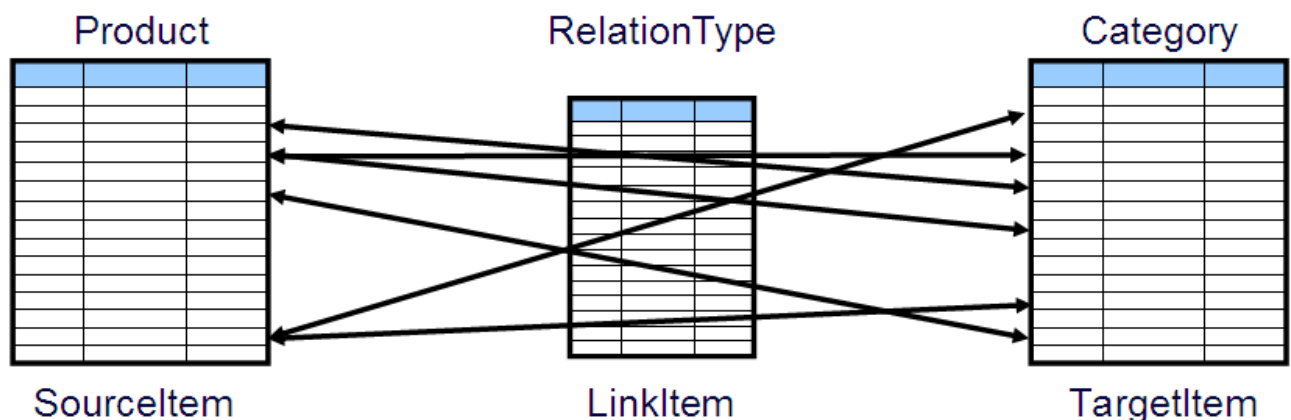
i Note

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 `sampletype`, then its localized values are stored in the table `sampletypelp`. 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"
```

This is custom documentation. For more information, please visit the [SAP Help Portal](#)

```

        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.jalo"
    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 hybrisservice.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

This is custom documentation. For more information, please visit the [SAP Help Portal](#)

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, isActivate() );
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

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, isActivate() );
params.put( HelloWorldWizardCronJob.INTERVAL, getInterval() );
params.put( HelloWorldWizardCronJob.CODE, "HelloWorldWizardCronJob" + String.valueOf( jobNum ) );
params.put( HelloWorldWizardCronJob.JOB, hwwj );
HelloWorldWizardCronJob hwwcj = HelloWorldWizardManager.getInstance().createHelloWorldWizardCronJob
...
```

`ComposedType.newInstance()` 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 )
    {
    }
}
```

10/20/2021

```
{
    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 " + allAttribute

    // 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
-------------	-----------------------

Method Name	Description / Comment
protected Item createItem(SessionContext ctx, ComposedType type, ItemAttributeMap allAttributes) throws JaloBusinessException	The actual creation method. Implement it so that its return value is the new item. Marked abstract in the Item type.
protected ItemAttributeMap getNonInitialAttributes(SessionContext ctx, ItemAttributeMap allAttributes)	Implement this so that it filters allAttributes and removes values that were added during createItem(. . .) . If you override this method, be sure to call the super class' implementation of this via super.getNonInitialAttributes(ctx,allAttributes) otherwise you would have to implement the removal values the default implementation would have removed already.
public void setNonInitialAttributes(SessionContext ctx, Item item, ItemAttributeMap nonInitialAttributes) throws JaloBusinessException	This method has to set all (non-initial) values returned by the getNonInitialAttributes(. . .) in other words, all values that have not yet been set yet.
getInitialProperties()	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 setNonInitialAttributes() 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 (true) or whether the null value is treated as a missing value (false). Defaults to false.

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 allAttribu
{
    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
    }
}

```



```

        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 allAttribu
{
    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
    }
    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 technically, 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

Kind of Relation	SAP Commerce Type
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">
  <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" init="true"/>
    </attribute>
  </attributes>
</itemtype>
```

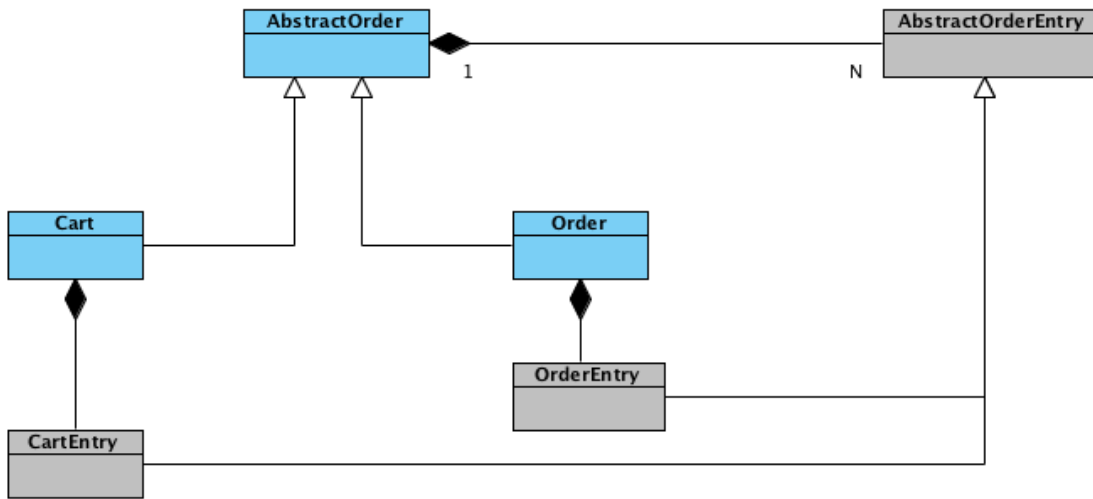
1:n Relations - One-to-Many vs Many-to-One

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="AbstractOrder">
    <modifiers read="true" write="true">
      <custom-properties>
        <property name="ordering.attribute">entryNumber</property>
      </custom-properties>
    </modifiers>
  </sourceElement>
  <targetElement type="AbstractOrderEntry">
    <modifiers read="true" write="true">
    </modifiers>
  </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.commerce.core.order.Order"
  autocreate="true"
  generate="true">
    <deployment table="Orders" type="Order">
      <attributes>
        <attribute autocreate="true">entryNumber</attribute>
      </attributes>
    </deployment>
  </itemtype>

```

3. Additionally, the following collection type is needed:

```
<collectiontype code="OrderEntryCollection" elementtype="OrderEntry" autocreate="true" generate="true">
```

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="OrderEntry"
    <attributes>
    <attribute autocreate="true"
    <modifiers read="true" write="true"
    </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

With the latest patch, you can switch off ordering of one-to-many relations in OneToManyHandler.

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 attribute is used to order the items of the **-many** side when they are retrieved from the database. When you create a new instance of the item of the **-many** side, for example ProductFeature for Product2FeatureRelation, this attribute is assigned a new value. 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. When you create multiple items of the **-many** side, for example during impex import or synchronization, these SQL statements become expensive from a performance perspective. You can disable them through a property that uses the following naming convention:

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

Here is a use example:

```
relation.Product2FeatureRelation.reordered=false
```

You can set the property to these values:

- `false` is the default setting where the SQL statement that retrieves the current highest value of an ordering attribute gets executed
- `sync` means that only when a **sync job** is active, the SQL statement doesn't get 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 (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`)

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" i
  </sourceElement>
  <targetElement type="Address" cardinality="many" qualifier="address"
    <modifiers read="true" write="true" search="true" optional="true" p
  <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`.

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

AttributeDescriptor Modifiers

Modifier	Description / Comment
qualifier	The identifier of this Attribute

Modifier	Description / Comment
autocreate	If set to true, a new Attribute gets created when the Platform creates the type system during initialization. Set this to false 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.createAttributeDesc
AttributeDescriptor.READ_FLAG + AttributeDescriptor.WRITE_FLAG + At
AttributeDescriptor.SEARCH_FLAG );
AttributeDescriptor usedAsFollowUpForAttr = productType.createAttri
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( "followU
Collection usedAsFollowUpFor = (Collection)myProduct.getAttribute(
```

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

Creating an EnumType

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

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, At
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](#)

[ImpEx](#)

[Initializing and Updating SAP Commerce](#)

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 GeneratedMy
[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 GeneratedMy
[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 GeneratedMy
[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 GeneratedMy
[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 occur
/opt/hybris/bin/platform/resources/ant/antmacros.xml:385: The following error occur
/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 occur
/opt/hybris/bin/platform/resources/ant/antmacros.xml:472: The following error occur
/opt/hybris/bin/platform/resources/ant/antmacros.xml:570: The following error occur
/opt/hybris/bin/platform/resources/ant/antmacros.xml:591: The following error occur
/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.

Addresses

Addresses:

	Postal Code	Town	Street Name
	12345	n/a	Jalo Road
	23456	n/a	Payment Avenue

Default Payment Address:

Payment Avenue - 12 - 23456 - n/a


Default Shipment Address:

Jalo Road - 5 - 12345 - n/a

The Address Tab

Unbound

Base DN:

[mirroredAddress] Jalo Road - 5 - 12345 - n/a 

[orderProcess]:

	Code	Job	Current status
The list is empty.			

Copies

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: the [ServiceLayer](#) and the 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">

  <atomictypes>
    ...
  </atomictypes>

  <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-item
[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-item
```

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.train
<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>
    (....)
```

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="CartEntryCollec
      <modifiers read="true" write="true" search="true" removable="true" optional="true" p
    </attribute>
    <attribute type="java.lang.String" qualifier="sessionId">
      <persistence type="property"/>
      <modifiers read="true" write="true"/>
    </attribute>
  </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.

This is custom documentation. For more information, please visit the [SAP Help Portal](#)

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>CLOB</value>
  </columntype>
  <columntype database="sap">
    <value>NCLOB</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.

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

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

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

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
Nilable	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>atomicType</atomicType> [0..*]	An atomicType represents a simple Java object. (The name 'atomic' just means 'non-composed' objects.)
</...>	

Complex Type: atomicType

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

<...>	
-------	--

<code>generate="boolean" [0..1]</code>	Whether getter and setter methods for the model representation of the attribute will be generated. Default is 'true'.
<code>></code>	
<code><getter>modelMethodType</getter> [0..*]</code>	Allows to configure alternative getter methods at generated model.
<code><setter>modelMethodType</setter> [0..*]</code>	Allows to configure alternative setter methods at generated model.
<code></...></code>	

Complex Type: attributesType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Configures a list of attributes.

XML Instance Representation

<code><...></code>	
<code><attribute>attributeType</attribute> [0..*]</code>	Defines a single attribute.
<code></...></code>	

Complex Type: attributeType

Parent type:	None
Sub-types:	None
Abstract	no
Documentation	Defines an attribute of a type.

XML Instance Representation

<code><...></code>	
<code>redeclare="boolean" [0..1]</code>	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'.
<code>qualifier="string" [1]</code>	Qualifier of this attribute. Attribute qualifiers must be unique across a single type.
<code>type="string" [1]</code>	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.
<code>metatype="string" [0..1]</code>	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:

This is custom documentation. For more information, please visit the [SAP Help Portal](#)

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: columntypeType

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 'hsq'. 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

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: itemTypeType

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

<...>	
<matype>matypeType</matype> [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]	<p>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> <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>columnType</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.
autocreate="boolean" [0..1]	If 'true', the item will be created during initialization.
generate="boolean" [0..1]	Deprecated. Will have no effect for relations.
>	
<description>string</description> [0..1]	Provides possibility to add meaningful description phrase for a generated model class.
<deployment>deploymentType/deployment> [0..1]	Configures deployment information for this relation (table name and type code).
<sourceElement>relationElementType/sourceElement> [1]	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.
<code>/></code>	

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 <i><pattern></i> = ([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 <= <i><value></i> <= 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" collect
    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>
</attribute>
<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>
```

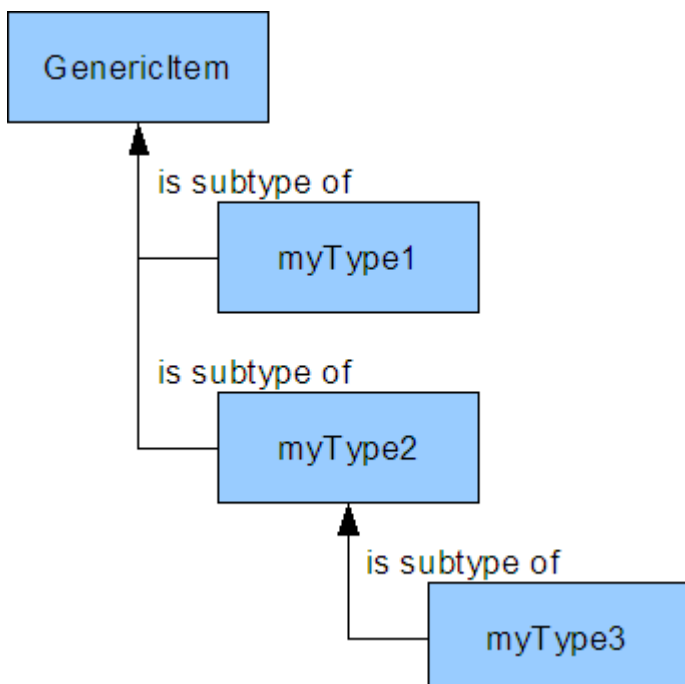
Otherwise it would be easy to forget to specify the deployment, and, as a result, some types would go to the **GenericItem** table. The build failure reminds you that deployment is not specified for some types. To change this default behaviour, set the property **build.development.mode** to **false** 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" />
```

- The deployment attribute specifies the table name into which the instances of the type are written, such as `table="mytype_deployment"`
- 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)):YDeployme
de.hybris.platform.persistence.europe1_DiscountRow::((europe1)):YDeployment[europe1.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 not be performed.

- If there was another deployment defined before:

Modification of the deployment for type <TYPECODE> from <OLDDEPLOYMENT> to <NEWDEPLOYMENT> will not be performed.

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" persistence-
            <attribute-mapping database="oracle" persistence-name="VALUESTRING1" persistence-
            <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>

    </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="LONGVARCHAR" />
  <type-mapping type="String" persistence-type="LONGVARCHAR" />

  <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.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.

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:

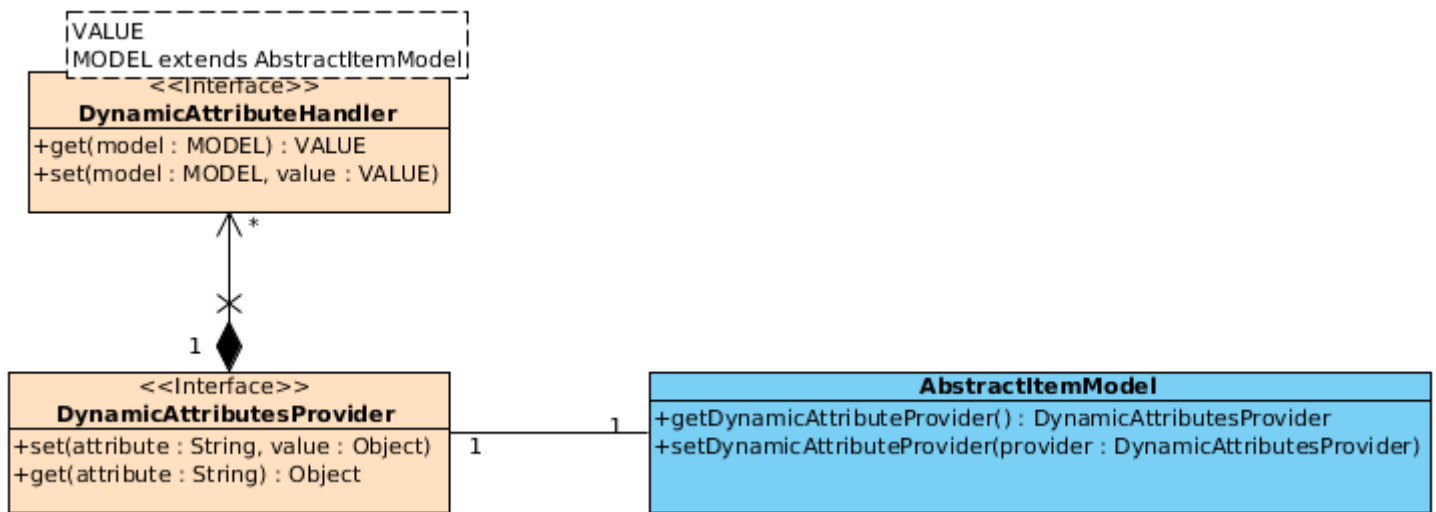
```
<itemtype code="Accommodation" autocreate="true" generate="true" jaloclass="de.hybris.platform.test
  <attributes>
    <attribute type="int" qualifier="daysHumanReadable">
      <persistence type="dynamic" attributeHandler="myOwnSpringBeanId" />
      <modifiers read="true" write="false" />
    </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
  <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, ClientNameMod
{
    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();
    }
}

```

10/20/2021

```
@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.DynamicAttribu
```

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 `Generated ItemTypeCode` 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 `Generated ItemTypeCode` 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"
  jalo="de.hybris.platform.cms2.jalo.restrictions.AbstractRestriction"
  extends="CMSItem"
  autocreate="true"
  generate="true"
  abstract="true">
  <attributes>
    <!-- ... -->
    <attribute qualifier="description" generate="true" autocreate="true" type="localized:java.l
      <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.!
    <persistence type="dynamic" />
    <modifiers write="false" />
  </attribute>
...
```

2. Define separate Dynamic Attributes for each type that extends `AbstractRestriction` type:

- 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.

- Change persistence type of attribute to dynamic.
- Provide a custom bean if for attributeHandler.

```
<itemtype code="CMSCatalogRestriction"
  jalo="de.hybris.platform.cms2.jalo.restrictions.CatalogRestriction"
  extends="AbstractRestriction"
  autocreate="true"
  generate="true">
  <attributes>
    <attribute qualifier="description" type="java.lang.String" redeclare="tr
      <persistence type="dynamic" attributeHandler="catalogRestriction
      <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"
            <persistence type="dynamic" attributeHandler="inverseRestriction"
            <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, CMSInverseRestrictionModel>
{
    @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, CMSCatalogRestrictionModel>
{
    @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.cmscatalogrestriction.description",
                "Page only applies on catalogs: ");
            result.append(localizedString == null ? "Page only applies on catalogs: " : localizedString);
            for (final CatalogModel cat : catalogs)
            {
                result.append(" ").append(cat.getName()).append(" ").append(cat.getId());
            }
        }
        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.

1. 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());
    }
    // ...
}
```

2. 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.CMSCatalogRestriction}
     * 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 inte
<defaultvalue>em().getEnumerationValue("enumSampleType", "sample2")
</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](#)