# 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.

**Note**

**Both ServiceLayer And Jalo Layer Covered in This Document**

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

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

## Location

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

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

## Structure

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

### Basic Structure

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

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

   <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-items.xml]
[yxmlschemavalidator]
[yxmlschemavalidator] line:24,column:15 : cvc-complex-type.2.4.d: Invalid content was found starting with element 'itemtypes'. No child element is ex
[yxmlschemavalidator]
[yxmlschemavalidator] ERROR(S): [file:///C:/hybris/trunk/bin/myextension/resources/myextension-items.xml]
```

### 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>
```

**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.training.jalo.MyProduct">
   <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

> **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.

> **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="CartEntryCollection">
               <modifiers read="true" write="true" search="true" removable="true" optional="true" partof="false"/>
            </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.

**Note**

You can't use types unrelated to their hierarchy.

## ServiceLayer

**Note**

**ServiceLayer-only section**

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

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

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

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

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

### Null Value Decorators in Models

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

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

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

```
public Boolean getCalculated()
```

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

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

## Jalo Layer

> **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>
```

> **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>
```

> **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>
```

```
</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.

### Overriding Column Types

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

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

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

### System Update Possibilities

Enumeration, Map, Collection

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

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

Relation

| Creating | removing | changing | change partOF attribute | source/target attribute name change |
|---|---|---|---|---|
| Works | works but cleanup necessary | manual remapping needed, for (1- | works | works, cleanup necessary |

| | | | | |
|---|---|---|---|---|
| | | n) cleanup doesn't remove created instances, for (n-m) it does | | |

Type

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

Attribute

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

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

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

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

Such configuration translates into creating the following index:

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

## Defining Index with Included Columns in Configuration Files

You can include or override columns using the following property:

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

Separate multiple attributes using a comma (“,”).

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

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

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

## Primitive Types

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

Assumed that you have a definition like this:

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

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

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

An update system is sufficient to make the change effective.

## Support for the @Deprecated Annotation Attributes

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

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

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

## Related Information

Type System Documentation

Build Framework

Initializing and Updating the SAP Commerce

Jalo Layer

Using Encryption for Attribute Values

Specifying a Deployment for hybris Platform Types

Jalo-only Attributes

Working with Enumerations

http://java.sun.com/products/jdo/JDOCMPFAQ.html