Type System

Summary

This page describes the recommended practices when defining the project data model in the Hybris Commerce Type System. For a primer on the Type System, see Type System Documentation

This page relates to the Core Type System however it is also possible to create extensions that use the Core+ Type System for persistence. The DataHub and Order Management System both use Core+. Both Core and Core+ use similar concepts for data modeling however this page only relates to Core. For the official documentation for Core+ see The Core+ Type System.

Prerequisites

- None

Deliverables

- Data Model Definition (`*-items.xml` files)

Suggested Tools

- None

General Recommended Practices

Rule	Description

A deployment table must be defined for all Items extending *GenericItem*	Failing to declare a deployment type means that all instances of this stored in the genericitems in this table with other types with no deploy table. This table can become very large in terms of rows, columns as indexes. This can make the database work inefficiently causing perfo issues . Fixing this issue in a production environment requires a data script.
	
	Example:
	
	`<itemtype code="mytype" extends="GenericItem" autocreate="true" generate="true">`
	`<attributes> <!-- deployment should be defined here before attributes -->`
	`...`
	`</attributes>`
	`</itemtype>`

A deployment table must not be defined for any Items extending any item other than *GenericItem*	**A deployment table must not be defined for any Items extending other than *GenericItem***
	
	Defining a deployment table for an type that already has a deployme inherited from a super type can cause serious performance issuesbe queries for the super type require a *UNION* clause with the new table
	
	`SELECT {PK} FROM {Media} WHERE {code} = ?code UNION`
	`SELECT {PK} FROM {MyMedia} WHERE {code} = ?code`
	
	Fixing this issue in a production environment requires a data migratic
	
	Example:
	
	`<itemtype code="MyMedia" extends="Media"`
	`autocreate="true" generate="true">`
	`<deployment table="mymedia" typecode="22222"/> <!--`
	`deployment should NOT be defined -->`
	`<attributes>`
	`...`
	`</attributes>`
	`</itemtype>`

Deployment type codes must be > 10000	Type codes less that 10000 are reserved for core Hybris Commerce There is a strong possibility that there is a conflict with a system type you use a value less than 10000. Bear in mind that even if there are with the current version of Hybris Commerce and extensions being u upgrade to a newer version of Hybris Commerce could cause a type confict.
	
	There are few exceptions even for code numbers above 10000, see a Deployment for hybris Platform Types for more details.
	
	`<itemtype code="MyCustomType" extends="GenericItem"`
	`autocreate="true" generate="true">`
	`<deployment table="mycustomtype" typecode="1000"/> <`
	`deployment typecode should be > 1000 -->`
	`<attributes>`
	`...`
	`</attributes>`
	`</itemtype>`

Jalo class must not be defined if an existing type is being extended	It is only possible to specify a Jalo class if we are sub-typing
	`<itemtype code="Product"  autocreate="false"`
	`generate="false"`
	`jaloclass="com.hybris.product.jalo.MyProduct"> <!--`
	`violation here, the type Product already has a jaloc`
	`-->`
	
	`<attributes>`
	`...`
	`</attributes>`
	`</itemtype>`

A deployment table must be defined for all many-to-many relations	This is very similar to the need having explicit deployment tables for i extending the GenericItem. Without an explicit deployment table, the will be persisted in the *links* table. The performance degradation happ result of too many rows, large indexes and table locking during insert / deletes. Typically, if the *links* table is one of the five largest tables ir system, this is the place to fix it.

Attributes should not have a persistence type of 'cmp'	This is a deprecated persistence mode that should not be used anym
	`<itemtype code="MyProduct" extends="Product"`
	`autocreate="true" generate="true">`
	`<attributes>`
	` <attribute qualifier="myAttribute"`
	`type="java.lang.String">`
	`         <persistence type="cmp"/> <!-- should only`
	`property or dynamic -->`
	` </attribute>`
	` </attributes>`
	`</itemtype>`

Mandatory fields (where optional='false') must either have initial set to 'true' or a default value defined	This ensures data consistency on the Type System level. Also note, t set a default value in the items.xml file, the default value will be persis newly created items only, existing data will not be updated.
Immutable fields (where write='false') must have initial set to 'true'	Setting write="false" means that a value can only be set when we Iter created. Setting initial="true" effectively achieves the same. For cons should set initial="true" when write="false"
Boolean attributes must be defined as mandatory	It is possible to store 3 values for a Boolean attribute (true, false or nu null is almost always not the desired outcome and it will usually be m true or false value. It's possible that the mapping is not always in plac is potential for logic errors or NullPointerExceptions

Attributes should not have a persistence type of 'jalo', use 'dynamic' instead	JALO is a deprecated API and should not be used. Previously JALO were used when we need some business logic to be executed before a result. This behaviour can now be achieved with Dynamic Attribute Service Layer without needing to touch JALO
	`<itemtype code="MyProduct" extends="Product" autocreate="true" generate="true">`
	`<attributes>`
	` <attribute qualifier="myAttribute" type="java.lang.String">`
	`         <persistence type="jalo"/> <!-- should only property or dynamic -->`
	` </attribute>`
	` </attributes>`
	`</itemtype>`

Type Names (including EnumTypes and Relations) must start with an uppercase letter	For consistency all type names should start with an uppercase letter. run into issues with the code generators (e.g. platformwebservices) i not follow this convention.
	`<itemtype code="myProduct" extends="Product" autocreate="true" generate="true"> <!-- violation he "myProduct" should be "MyProduct"`
	`<attributes>`
	` ..`
	` </attributes>`
	`</itemtype>`

Type Names must not start with the string Generated	Starting a type name with "Generated" can have some side effects b abstract class that starts with "Generated" is created by Hybris Comr already
	`<itemtype code="GeneratedProduct" extends="Product" autocreate="true" generate="true"> <!-- violation he "GeneratedProduct" will cause some side effects -->`
	`<attributes>`
	` ..`
	` </attributes>`
	`</itemtype>`

Item attribute names must start with a lowercase letter	For consistency all item type attribute names should start with an low letter. You may run into issues with the code generators (e.g. platformwebservices) if you do not follow this convention.
Relationship qualifier names must start with a lowercase letter	For consistency all relation type attribute names should start with a lc letter. You may run into issues with the code generators (e.g. platformwebservices) if you do not follow this convention.

Any side of a relation that has cardinality='many' should not have ordered='true' unless absolutely necessary	Order="true" has a significant impact on performance when reading a from the database. The queries required to provide a defined order tc that are returned are much more complex and expensive that an uno query. Using ordered="true" should only be used when absolutely ne
Any side of a relation that has cardinality='many' should have collectiontype='set' unless absolutely necessary	The set guarantees that every relation is persisted only once. For exa Product is related to a Categoryit is usually desirable to store this rela one row in the database.
CatalogVersion attribute should be marked unique for Catalog aware types	The CatalogVersion attribute forms part of the unique key for Catalog Synchronization. To ensure that the same uniqueness rules are appli other processes (e.g. ImpEx) we need to set the CatalogVersion attri unique="true" `<itemtype code="MyCatalogType" extends="GenericItem"` `autocreate="true" generate="true">` `<deployment table="MyCatalogType" typecode="XXXX"/>` `<custom-properties>` `<property name="catalogItemType">` `<value>java.lang.Boolean.TRUE</v` `</property>` `<property name="catalogVersionAttributeQualifier">` `<value>"catalogVersion"</value>` `</property>` `<property name="uniqueKeyAttributeQualifier">` `<value>"uid"</value>` `</property>` `</custom-properties>` `<attributes>` `<attribute qualifier="uid" generate="true" autocreate="true" type="java.lang.String">` `<persistence type="property" />` `<modifiers optional="false" unique="true" />` `</attribute>` `<attribute qualifier="catalogVersion` `type="CatalogVersion">` `<modifiers optional="false" /> <` `violation here because missing unique="true" -->` `<persistence type="property" />` `</attribute>` `</attributes>` `</itemtype>`

```
<itemtype code="MyCatalogType" extends="GenericItem"
            autocreate="true" generate="true">
            <deployment table="MyCatalogType"
typecode="XXXX"/>
            <custom-properties>
                <property name="catalogItemType">
                    <value>java.lang.Boolean.TRUE</v
                </property>
                <property
name="catalogVersionAttributeQualifier">
                    <value>"catalogVersion"</value>
                </property>
                <property
name="uniqueKeyAttributeQualifier">
                    <value>"uid"</value>
                </property>
            </custom-properties>
            <attributes>
                <attribute qualifier="uid"
generate="true" autocreate="true"
type="java.lang.String">
                    <persistence type="property" />
                    <modifiers optional="false"
unique="true" />
                </attribute>
                <attribute qualifier="catalogVersion
type="CatalogVersion">
                    <modifiers optional="false" /> <
violation here because missing unique="true" -->
                    <persistence type="property" />
                </attribute>
            </attributes>
        </itemtype>
```

Unique attributes should match the catalog unique attribute key (uniqueKeyAttributeQualifier)	The unique attributes defined for the type and the attributes defined i "uniqueKeyAttributeQualifier" should have the same attributes. "uniqueKeyAttributeQualifier" defines the unique attributes for Catalo Synchronization whereas processes using the Service Layer (includi Catalog Sychronization persistence) use the unique attributes on the

```
<itemtype code="MyCatalogType" extends="GenericItem"
            autocreate="true" generate="true">
            <deployment table="MyCatalogType"
typecode="XXXX"/>
            <custom-properties>
                <property name="catalogItemType">
                    <value>java.lang.Boolean.TRUE</v
                </property>
                <property
name="catalogVersionAttributeQualifier">
                    <value>"catalogVersion"</value>
                </property>
                <property
name="uniqueKeyAttributeQualifier">
                    <value>"uid"</value>
                </property>
            </custom-properties>
            <attributes>
                <attribute qualifier="uid"
generate="true" autocreate="true"
type="java.lang.String">
                    <persistence type="property" />
                    <modifiers optional="false"
unique="false" /> <!-- violation here because missin
unique="true" -->
                </attribute>
                <attribute qualifier="catalogVersion
type="CatalogVersion">
                    <modifiers optional="false"
unique="true" />
                    <persistence type="property" />
                </attribute>
            </attributes>
        </itemtype>
```

Database indexes should be defined for the unique attributes of type	There should be a covering index defined that includes all the unique for type. There are Service Layer interceptors that validate uniquenes type, which generate a query for all the unique attributes. Failure to a index can cause serious performance issues. Furthermore, the valida unique attributes in the Service Layer cannot guarantee that there wo duplicate records in the database, only a unique index/constraint can
	
	`<itemtype code="MyType" extends="GenericItem"`
	`            autocreate="true" generate="true">`
	`            <deployment table="MyType" typecode="XXX`
	`            <attributes>`
	`                <attribute qualifier="uid"`
	`generate="true" autocreate="true"`
	`type="java.lang.String">`
	`                    <persistence type="property" />`
	`                    <modifiers optional="false"`
	`unique="true" />`
	`                </attribute>`
	`            </attributes>`
	`   <indexes>`
	`   <!-- violation here because missing index for uid`
	`   </indexes>`
	`        </itemtype>`

Catalog aware types must not be part of a one-to-many relation when the many side is not a Catalog aware type	Creating a one-to-many relation where the one side of the relation is aware type and the many side is not a Catalog aware type causes ar when synchronizing. The reference to the Catalog aware type will be transferred from the Staged to the Online version. The Staged versio longer have a reference.
	
	`<relation code="Product2Author" generate="true"`
	`localized="false" autocreate="true">`
	`            <sourceElement type="Product"`
	`cardinality="one" qualifier="product"/> <!-- violati`
	`here, MyItem is not a catalog aware type and it can`
	`reference a single Product (not a Staged and Online`
	`version -->`
	`            <targetElement type="MyItem"`
	`cardinality="many" qualifier="myItems" ordered="true`
	`</relation>`

All types should have unique attributes defined	Every type should have a unique identifier(s) defined or should inheri unique attributes from a super type. This allows the data to be easily imported/exported into different systems and prevents duplicate entri database.
	`<itemtype code="MyType" extends="GenericItem" autocreate="true" generate="true">`
	`<deployment table="MyType" typecode="XXX`
	`<attributes> <!-- missing a unique attri -->`
	`<attribute qualifier="name" generate="true" autocreate="true" type="java.lang.String">`
	`<persistence type="property" />`
	`<modifiers optional="false" unique="false" />`
	`</attribute>`
	`</attributes>`
	`</itemtype>`

Unoptimized attributes should not be used	Using dontOptimize="true" stores the data for this type in a secondar defined by the "propertytable" attribute of the deployment or the prop no propertytable is defined. Generally it is not recommended to store values in this way because it is more expensive to store and retrieve because joins are required.
	`<itemtype code="MyType" extends="GenericItem" autocreate="true" generate="true">`
	`<deployment table="MyType" typecode="XXX propertytable="MyTypeProps"/>`
	`<attributes>`
	`<attribute qualifier="name" generate="true" autocreate="true" type="java.lang.String">`
	`<persistence type="property" />`
	`<modifiers optional="false" unique="false" dontOptimize="true"/> <!-- defining dontOptimize stores this value in the property table this type (MyTypeProps) of the props table if none i defined -->`
	`</attribute>`
	`</attributes>`
	`</itemtype>`

Extending Existing Type vs Sub-Typing

When extending an existing type to add new attributes we can either add the attributes to the existing type

e.g.

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

Or we can sub-type e.g.

```
<itemtype code="MyProduct" extends="Product" autocreate="true"
generate="true">
<attributes>
     <attribute qualifier="newAttribute" type="java.lang.String">
     <modifiers read="true" write="true" search="true" optional="true"/>
      <persistence type="property"/>
</attribute>
</itemtype>
```

Whether to sub-type or extend the existing type should be based on the following decisions:

Change	**Existing or Sub-Type**	**Reason**
Adding an attribute for all instances	Existing	If all instances of the existing type could have a value for this data then the attribute belongs on existing type. Creating an unnecessary sub-type can create issues because a business user could create an instance of the super type that doesn't have all the required/expected attributes e.g. a user could create instances of items that are not compatible with the business logic For example, if we sub-type Category and create new type with a mandatory attribute called "enabledForFrontend". Business users can still create a standard Category item that doesn't have the required attribute that the front end is expecting. In addition the Hybris Commerce APIs expect the existing type so the project code will be cluttered by "if instanceof then (cast)" constructs if a subtype is used.
Adding an attribute for specific instances	Sub-Type	If only some instances have data for a particular attribute then we should sub-type.
Re-declaring an existing attribute e.g. change mandatory to optional	Sub-Type	Re-declaring requires sub-typing

Redeclare only when necessary because it can cause potential issues if business users are still able to use the existing types that don't have the attributes required by the system. If sub-typing is required then consider redeclaring attributes so that the new sub-types are returned instead of the existing types.

Although it is typical for implementations to extend a Hybris Commerce type by subclassing, it is preferable to override the type and embed the new fields into it.

For example, if a customer wants to add new fields to the *Category* model, the standard approach used in implementations is to subclass it. If we know there will not be any other direct subclasses or the new fields will be shared by other subclasses, it is preferable to simply add the new fields directly to the *Category* model. This results in a simplification of the data model and queries against the data model.

Finally, rather than extending or subclassing a type, it may be preferable use a composition approach in which we create new types with relations to the core Hybris Commerce types.

Product Features vs Product Attributes

When creating a complex data model for Products we have 2 options for defining new attributes. Creating an new Type System attribute in `*-items.xml` or creating a new Product Feature in the Classification Guide. The Classification Guide has the benefit of allowing new features to be defined at run time rather than requiring a deployment however it comes at a significant cost in terms of performance and scalability. Loading the Product Features for a Product can be costly and is something we should only be doing on specific pages where the impact on performance is minimal (e.g. Product Details Page). Also Product Features are not easily accessible by the Hybris Commerce API so any business logic using Product Features will be a more difficult to implement and maintain.

Change	Feature vs Attribute	Reason
Attribute belong to most items e.g. height, width, color	Attribute	If the attribute belongs to most items then for efficiency we should store it as a attribute of the Product
Attributes is shared by a small range of Products	Feature	For example in an Electronics Catalog we haves TVs as a Product range. These products we will have a number of common attributes that we can define as Product Features. Creating sub-types would mean an explosion in the number of Product types in the system that would be difficult for the business to manage
Attribute definitions that change frequently	Feature	If we need to regularly change the attribute definitions as new Product lines are release or Products change then we should use Product Features to avoid needing to deploy a new release for these changes
Attribute is an Item reference e.g. reference to Media	Attribute	Product Features are restricted to simple types e.g. String, Date, Number

See also Defining Product Attributes

Take special care when using Product Features from the Classification Guide for large catalogs due to the impact not only on the front end but also the back end processes (imports, catalog publication, Solr export) and backoffice applications.

Where is This Recommended Practice Used?

Phases:

- Agile PDF - 4. Engineering Phase

Activities:

- Agile PDF - Engineering Implementation Sprints

Page Information

Disciplines: Architecture and Development

Page Views : 2782

Contributions

Contributor	Contribution
Various	-