

QUAN MAI

PRO OPTIMIZEZLY COMMERCE CLOUD

Understand the framework that builds powerful
eCommerce solutions

Pro Optimizely Commerce Cloud

Quan Mai

This book is for sale at

<http://leanpub.com/prooptimizelycommercecloud>

This version was published on 2023-09-01



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 - 2023 Quan Mai

Also By Quan Mai

Pro Episerver Commerce

Episerver Commerce: A problem - solution approach

Git: in easy steps

For my children - Emmy and Erik.

Contents

- Part 1: Catalog System 1**
 - Chapter 1: How the catalog is structured 2**
 - The entries 6
 - The product-variant configuration 13
 - The two systems. 14
 - ReferenceConverter - the bridge. 17
 - Chapter 2: Catalog content modeling 22**
 - The strongly typed models 22
 - Modeling catalog content types 25
 - The MetaData system 31
 - Dictionary types. 41
 - Chapter 3: Associations, relations and assets 46**
 - Working with associations and relations 46
 - Assets 51
 - Catalog content versions 59
 - Import and export catalogs 68
 - Batch API 73
 - Performance considerations 74

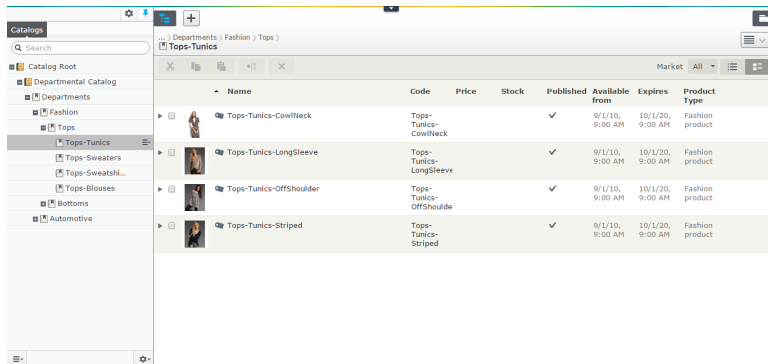
Part 1: Catalog System

The catalog system, in many means, can be called the heart of Optimizely Commerce Cloud. It might be the most used system in entirely Commerce. From cases to cases implementations might want to replace the order system, or to let their CRM to handle customers data, but all of them use the Catalog system.

Chapter 1: How the catalog is structured

##Overview





Catalog Management is the old catalog-editing UI in Commerce Manager, which has been there since the first version of Episerver Commerce. Catalog UI is the rewrite interface which integrates into CMS UI since Episerver Commerce 7.5. Catalog UI is vastly superior to Catalog Management in almost all aspects, and it's recommended to use by Episerver. However we'll show screenshots in both interfaces to see how they're connected.



The screenshot shows the Catalog UI interface. On the left is a tree view with the following structure:

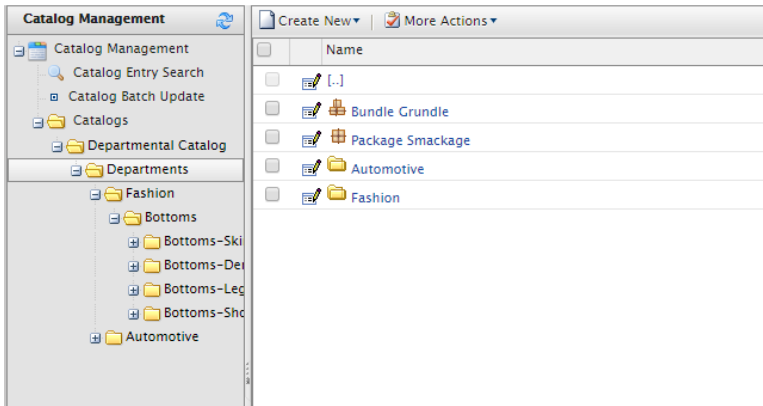
- Catalogs
 - Search
 - Departmental Catalog
 - Departments
 - Fashion
 - Tops
 - Tops-Tunics**
 - Tops-Sweaters
 - Tops-Sweatsh...
 - Tops-Blouses
 - Bottoms
 - Automotive

The main table displays the following data:

Name	Code	Price	Stock	Published	Available from	Expires	Product Type
 Tops-Tunics-CowlNeck	Tops-Tunics-CowlNeck			✓	9/1/10, 9:00 AM	10/1/20, 9:00 AM	Fashion product
 Tops-Tunics-LongSleeve	Tops-Tunics-LongSleeve			✓	9/1/10, 9:00 AM	10/1/20, 9:00 AM	Fashion product
 Tops-Tunics-OffShoulder	Tops-Tunics-OffShoulder			✓	9/1/10, 9:00 AM	10/1/20, 9:00 AM	Fashion product
 Tops-Tunics-Striped	Tops-Tunics-Striped			✓	9/1/10, 9:00 AM	10/1/20, 9:00 AM	Fashion product

Catalog UI - the new catalog management UI from Commerce 7.5

The catalog system - as the heart of Episerver Commerce - has been there for long, long before the merge and long before CatalogContentProvider. In the most basic concept, a catalog can be viewed as a tree - a catalog is the root, the nodes are the branches, and the entries are the leaves. It helps to grasp some ideas about the catalog system, but the catalog is much more complicated than that, which all kinds of relations and associations we'll discuss shortly.



Catalog Management in Commerce Manager



There are several reasons Catalog UI is better: it has vastly improved UX, it allows previews (so you can see how changes look like for end-users, on different devices (desktop, tablet, phones, etc.)), it allows you drag and drop contents, and also provides unification between CMS and catalog content editing, so you can (easily) link a catalog content into a CMS content, and vice versa. It separates language properties in different views, making it easier for you to find and edit a field (If you tried to edit an entry which has 100 metafields in 5 languages in Commerce Manager, you'll know the pain). The only place where it might be inferior than Catalog Management is it might not feel as fast (when you switch between Preview mode and All properties mode, for example). However when it comes to productivity, it's certainly a big step forward. There is no reason to use Catalog Management these days!

The catalog itself, can be seen as the container in the system. You can import and export it at will. Whenever you export a catalog, all other information come along (the nodes, the entries,

the warehouses, the inventories information, the prices, and the metadata classes). And you can import it to another system. You can use many ways to transfer catalog information between system, such as writing the code to update the entries directly from CSV files, use ServiceAPI to update from a PIM like InRiver, ... but the most common way to date, is to export and import the catalog.¹²

A catalog can have multiple languages. Unlike CMS content, when a page might, or might not have a language, a catalog content - node or entry, always have all languages as defined in their true catalog. You can't delete a language branch. If your catalog has `sv` but your entry does not have version on that language yet, a new `sv` version will be created on fly.



Prior to Commerce 9, adding a new language to a catalog will also update all the drafts of all contents, hence it's very expensive operation. It's been much improved in Commerce 9, but it's still quite slow and you won't want to do it on a live site during high load time.

Under catalog, you usually have nodes. You can create an entry to be a direct children of a catalog. However, in that case, it actually means that entry does not belong to any nodes.



In Commerce 11 or later, an entry that does not have any primary node-entry relation will also appear to be direct child of the catalog, even if it appears in other nodes.

¹When you export the catalog, you might assume that you will export all of the asset information attached to your nodes and entries. However, due to the limitations of the content type models (we will talk about it later) , when you export inside Commerce Manager, those information will be lost.

²You will export all of the warehouses and metaclasses in the system, even if the catalog being export does not use those. That's the default behavior. You might argue that it's not very reasonable. I think it's up to your point of view to see if the system should be exporting those or not. If you want a clear "catalog", follow the instruction in [my blog post](#).

Under catalog, you can have nodes[^foo114]. And under the node, you can have another nodes, or entries. The naming can be a little confusing here: It's sometimes called node, or `CatalogNode`, but in the new Catalog UI, it's called `Category` to match with "Category concept" of CMS content.

There are two kinds of relations between nodes: - Each node would have one parent node. Nodes which have no parent node will be direct children of the catalog. This relation is defined by the `ParentNodeId` in `CatalogNode` table.

- A node can be linked to one or multiple nodes. When a node is linked, it will appear in as a normal child node in the parent node. This relation is defined by the `CatalogNodeRelation` table.

A node might contain, of course, many entries. And vice versa, an entry can belong to multiple nodes. This kind of relation is defined is `NodeEntryRelation` table.[^foo115]

And then we have relations and associations between entries.

Node-entry relation

Before Commerce 11, there is no way to know which is the "true" parent node of an entry. Conventionally, the first relation with lowest `SortOrder` is assumed to be the parent node. This is not entirely correct, and it comes with a limitation: you can really drag and drop an entry within a node. If you have a category of "smart phone", you would want to make the most popular phone, like iPhone 14 or Galaxy S22 to the top of the category, because those are more likely to be bought by a customers. Prior Commerce 11, you just ... can't. It is not supported in Catalog UI, and if you want a workaround, it can be completed (for example adding a metafield to the entry type and use that as sort order).

From Commerce 11, the concept of primary `NodeEntryRelation` is introduced. So you can explicitly set a `NodeEntryRelation` to be primary, meaning that node is the true parent of that entry.














There could be only one (or no) primary `NodeEntryRelation` for each entry.

The entries

If the catalog system is the heart of entire Episerver Commerce, the entries can be called the heart of the catalog system. In the end it's the only thing your customers care about, right? And it might not be simple as you might think.

There are 4 types of entries in the system:

	Name	Code	Price	Stock	Published	Available from	Expires	Product Type
	 Bundle Grundle	bundle			✓	2/24/12, 9:25 PM	2/24/22, 9:25 PM	Bundle
	 Package Smackage	package	\$500.00	 10.0%	✓	2/24/12, 9:28 PM	2/24/22, 9:28 PM	Package
	 Tops-Tunics-CowlNeck	Tops-Tunics-CowlNeck_			✓	9/1/10, 9:00 AM	10/1/20, 9:00 AM	Fashion product
	 Tops-Sweaters-Cardigan-Black-Small	Tops-Sweaters-Cardigan-Black-Small_1	\$1,000.00	 10.0%	✓	9/1/10, 3:00 PM	10/2/19, 3:00 AM	Fashion variant/SKI

4 types of entries in the Catalog

- **Product:** Well it's a product. Normally, a Product is a place holder of information, it does not have inventories or prices for itself, so it's not sell-able. (I've seen some customers sell products directly, it's possible but it will be a lot more of works. I would suggest you to stick with the "standard" implementation instead). A product might have one or more variations.



There are Commerce sites which use Products as Variations, such as they have Prices on their own. Or some sites allow a Product to be the parent of other products. Those are possible, however, it'll make the implementation harder, especially when it comes to Catalog UI. I would suggest to stick with the common way.

- Variation/SKU: It's easiest to explain Product-Variation in term of a TV series. Let's pick the LG CX OLED TV for example (I have one of these, it's a pretty awesome TV by the way). We can have it as a product with all specifications (resolution, OS, features and App). There are 4 variations by sizes (48, 55, 65 and 77 inches). Here's a screenshot from LG website:



✓ In Stock

KEY FEATURES +

- Self-lighting OLED: Perfect Black, Intense Color, Infinite Contrast
- Q9 Gen 3 AI Processor 4K with AI Picture Pro/Sound Pro

77"	65"	55"	48"
\$2,799.99	\$1,999.99	\$1,399.99	\$1,499.99

Wall Mounting Service \$129.73

handy Wall mounting by Handy includes, installation of TV mounting bracket, mounting of TV, Load testing the hardware. (Mount not included) [Learn More](#)

ADD

A product with its variations

A variation has its own inventories and prices. In the end, it's the thing your customers actually buy.



It is possible your variations have no products on their own (when each and every variation is unique and don't share anything with other variation, it does not really make sense to have "products" here).

- **Package:** A package consists of two or more variations, which itself act as a variation. Think of a package as a collection of items. You might have Harry Potter books as seven variations, and then you have a collection of them as a package. Package has its own prices and inventories. Therefore, when you buy, you all of items as one. Customers should not have the ability to change the quantity or remove items from a package.
- **Bundle:** A bundle is just a collection of things you can add to the cart at once, but then they acts as individual items. They are something customers buy together, but are not forced to do so. You can change quantity of or remove items from the cart if you'd like to. Bundles have no prices or inventories of its own.



In earlier versions of Commerce, and I mean, really, really early, there was another type of entry, Dynamic Package, which allows customer to adjust some items in the package. The concept is hardly used and was removed in later releases.







And then comes the tangible relations and associations between them.

There are three type of relations between entries:

- `ProductVariation` between a product and its variations.
- `BundleEntry` between a bundle and its entries.
- `PackageEntry` between a package and its entries.

ContentBelongs ToVariantsAssetsRelated EntriesSettings

Variants of this product ✕

Name	Path
 Tops-Tunics-CowNeck-Black-Small	Catalog.Root\Departmental_Catalog\Departments\Fashion\Tops\Tops-Tunics\Tops-Tunics-CowNeck-Black-Small
 Tops-Tunics-CowNeck-Black-Medium	Catalog.Root\Departmental_Catalog\Departments\Fashion\Tops\Tops-Tunics\Tops-Tunics-CowNeck-Black-Medium
 Tops-Tunics-CowNeck-Black-Large	Catalog.Root\Departmental_Catalog\Departments\Fashion\Tops\Tops-Tunics\Tops-Tunics-CowNeck-Black-Large
 Tops-Tunics-CowNeck-Black-ExtraLarge	Catalog.Root\Departmental_Catalog\Departments\Fashion\Tops\Tops-Tunics\Tops-Tunics-CowNeck-Black-ExtraLarge
 Tops-Tunics-CowNeck-Black-SuperSize	Catalog.Root\Departmental_Catalog\Departments\Fashion\Tops\Tops-Tunics\Tops-Tunics-CowNeck-Black-SuperSize
 Edit variants	

If you look at the tab Variations/SKUs when editing a Product in Commerce Manager, you’ll see “Quantity” column. This is not entirely correct as the Quantity is never used in a ProductVariation relation. Catalog UI reflects this better. It only matters in BundleEntry and PackageEntry relations.

And then between entries, you can set the associations. It is to show relations between products, for example, they can be used to indicates the products that could be displayed as CrossSell or UpSell items. The framework does not use these information, it just allows you to store them and use them as you see fit.



It is up to merchandisers to set the associations. In many cases, the connections are just clear to add associations, for example, for an iPhone, it’s obvious that you should add associations for accessories such as charger, or AirPods (Pro). It will be trickier for things that customers usually buy together, as this is dynamic and can change over time. Those can be done by an external system. Optimizely has a “recommendation” service which applies machine learning to determine which products are best to recommend to customers viewing a certain product. More information can be found at [link](https://world.optimizely.com/documentation/developer-guides/personalization/)³

Relations between entries are managed by IAssociationReposi-

³<https://world.optimizely.com/documentation/developer-guides/personalization/>

tory, it is a fairly simple interface with just three methods:

```
1  public interface IAssociationRepository
2  {
3      /// <summary>
4      /// Gets the associations for the catalog content specified by the content link.
5      /// </summary>
6      /// <param name="contentLink">The content link.</param>
7      m>
8      /// <returns></returns>
9      IEnumerable<Association> GetAssociations(ContentReference contentLink);
10
11      /// <summary>
12      /// Removes the associations.
13      /// </summary>
14      /// <param name="associations">The associations to remove.</param>
15      void RemoveAssociations(IEnumerable<Association> associations);
16
17      /// <summary>
18      /// Updates matching associations and adds new associations for an entry.
19      /// </summary>
20      /// <param name="associations">The associations.</param>
21      m>
22      void UpdateAssociations(IEnumerable<Association> associations);
23  }
```


Variations

As the most important entry types in the system, variation and package receive a bit of special treatment. While they share some tables with other entry types, they have one specific table, Variation, which contains some of very important information about your SKU. You can view or edit those values in the Pricing tab in both Commerce Manager and Catalog UI:

Warning. Updating this item through Commerce Manager will overwrite any previous and

Overview	Markets	Pricing	Warehouse Inventory	SEO	Associations
Display Price	<input type="text" value="1000.00"/>				
Min. Quantity:	<input type="text" value="1.000000000"/>				
Max. Quantity:	<input type="text" value="50.000000000"/>				
Merchant:	<div>select merchant ▾</div>				
Weight:	<input type="text" value="5"/>				
Length:	<input type="text" value="0"/>				
Height:	<input type="text" value="0"/>				
Width:	<input type="text" value="0"/>				
Package:	<div>select package ▾</div>				
Tax Category:	<div>select tax category ▾</div>				
Track Inventory:	<div><input checked="" type="radio"/> Yes <input type="radio"/> No</div>				

Pricing tab in Commerce Manager

- Display Price (or List Price) is an interesting field - it's long obsolete since R3 with the birth of pricing system, however, if you have a catalog exported from old version, it might still be there. If you delete the value (aka set the value to be null), the field will be hidden. This field is not mapped to any property in strongly typed content, nor displayed in Catalog UI.
- Min Quantity and Max Quantity are the lower and upper limit a customer can add this SKU to cart. Those values will

be used in several workflows during checkout, so they are particularly useful if you want to configure something like “Only buy by batch of at least 4”, “Only 10 per cart”, etc. Note that almost every value of quantity in Episerver Commerce is stored in decimal. While this might not really be useful in normal shop (it does not really make sense to have 1.5 TV or 2.3 Blu-ray discs, right?), it’s a must for B2B scenarios or liquor stores.

- **Merchant** is a field supposed to identify the merchant provides this SKU, but I hardly see any real world uses of it. This is not reflected in strongly typed content.
- **Weight**: It should describe itself, but only the value without unit. The unit is the base weight setting of the Catalog. (which can be either kilograms or pounds). Optimizely Commerce does not really use the unit in any calculations, the setting is more of a convention. You can assume that the unit of the weight, for example, as gram instead. But in most of the cases, sticking with the setting would help you avoid later confusions.
- **Height, Width and Length**: Those are values of the dimensions. Like weight, they are not tied to an unit. You can set the unit in the base length setting of the Catalog (which is by default, is null, but you can choose between inches and centimeters).
- **Package**: Choose the shipping package the SKU might come with. You can define the packages in Administration/Order System/Shipping/Shipping Packages section of Commerce Manager. Previously, which come with dimension (Height, Width and Length) settings. Prior to 8.7, this is the only way to define the shipping dimensions, but you now can use much more convenient way with the Height, Width and Length attributes. However, this is still useful when you have non-box SKUs, such as flower vases.
- **Tax category**: The tax category this SKU belong to. You usually don’t update this manually, but you can still go to

Administration/Order System/Tax Configuration to add a tax category and try.

- **Track Inventory:** This is one important flag. If it's set to false it does mean you don't want to track the inventory of this SKU, so any check for inventory such as available quantity etc. will be skipped during checkout. That can be useful in cases such as your SKU is a software and is distributed via download.

The product-variant configuration

The most common configuration of catalog is product-variation. One product has multiple variants in size, color, material: you have a T-shirt, where you store information like brand, material, etc. You then have variants of that T-Shirt, with different size and color.

It is, of course, not the only configuration.

Another fairly common case is to have standalone SKUs: a variant is an entry of its own and does not belong to a product. This can be used when the item you are selling is unique, and have no variants.



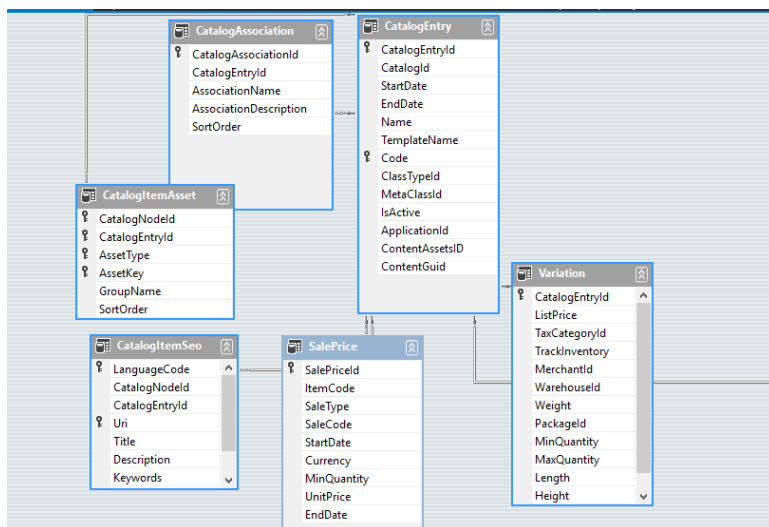
A content is only route-able (i.e. visible to the visitors) if it has a “template”. If you are using MVC for example, to route to your `MyVariantContent`, you would need to define a controller that inherits from `ContentController<T>`, which `T` is `MyVariantContent`, or its superclass.

Another, less common scenario is the product-product-variant. The first level product defines general information, like T-shirt. The second level product defines size, while the variants defines other information like color. This is used in a few websites - and to be honest I'm not sure why it's needed. It requires customizations on

framework level and the benefits aren't that clear. In my opinion, yes, you can, but that doesn't mean you should.

The two systems.

As mentioned in the brief history, Episerver Commerce was based on Mediachase eCF. Mediachase eCF built its catalog API:s around one interface - `ICatalogSystem` (If you worked with Commerce R3 and earlier, you might use the instance of it instead - `CatalogContext.Current`). `ICatalogSystem` methods use mostly DTO objects as inputs and outputs. The DTO:s are actually typed DataSets, and in most case, maps to several tables in database.



CatalogEntryDto

`ICatalogSystem` had its days. By today standards, it might not be the best API design, but it worked quite well until Commerce R3, which is the compatibility release for Episerver CMS 7. The biggest new feature in CMS 7 was the content concept (Before that, we had pages, and pages), so it became a big demand to have a content

API:s to work with catalog/node/entry, so Episerver Commerce can be considered as a first-class product in Episerver framework. It was made real in Commerce 7.5, and has been improved and refined ever since.

The content API:s to work with catalog content use `ICatalogSystem` internally. To be fair, `ICatalogSystem` is a performant, proven, reliable API:s, so why not use it directly?

First, using the new API:s means you use the same API:s as the Episerver CMS. Working on a unified API:s allow you to increase the code- recognition. Once you're familiar with the API:s, it'll be much quicker for you to know what the intention the code are doing to do (code-read), and much unlikely for you to use wrong method (code-write). In the end, it's your productivity which improves.

Second, the new API:s use a much more efficient caching mechanism, or it's the old system which does not cache thing very effectively. For the content way, if you load an object, you can cache it by its `ContentReference` and that's it. Every call to that `ContentReference` will hit the cache easily. Caching the DTO:s is a much harder problem. A `CatalogEntryDto`, for example, can have multiple rows, and depends on the `ResponseGroup` you specified, the data in some table might not be present. You get the cache, but hey, do you know if the next time can you re-use it when your parameters change?

Third, with the content way, you can read or update both of the "static" properties and the metafields in the same API:s. They are, after all, just properties. With the old way of `ICatalogSystem`, you can only update static properties and will need the help from `MetaObject` when you want to read or update the metafields.

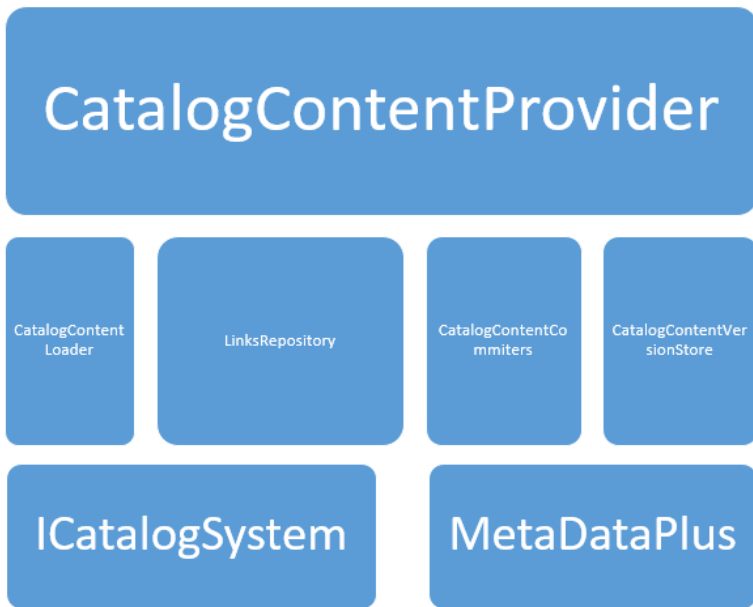
Forth, the POCO (Plain Old C# Object) approach in the content way is arguably move intuitive than the DTO:s (`DataSet:s`) approach of `ICatalogSystem`. Let's see how to change a name of an `EntryContentBase` vs a `CatalogEntryDto`:

```
1 entryContent.Name = "This is new name";
```

vs

```
1 entryDto.CatalogEntry[0].Name = "This is new name";
```

Tell me, which one will catch your eyes?



A simplified architecture of CatalogContentProvider

And finally, ICatalogSystem simply provides nothing to work with versions. New content APIs are simply superior because they are designed to work with versions.

In short, if you start your Optimizely Commerce Cloud project today, you should be using the latest and greatest version, and you should be using the IContentRepository. There are very little reasons to use ICatalogSystem, content is the way forward.



In the past, one place where you might have to consider `ICatalogSystem` instead of `IContentRepository` is when you want to process entries in batches. With `IContentRepository` you will always edit content one by one, but with `ICatalogSystem` you can save multiple `CatalogEntry` at once.

ReferenceConverter - the bridge.

One of the biggest features in Episerver CMS is its content provider system, which allow you to plug your own content provider to the system. So anything can be considered content, and can be manipulated with one unified API:s. However, to do that, you have to solve the first issue - the identity.

There are two problem Commerce had to resolve when they try to plug the catalog structure into the content provider system. Firstly, the data are stored in three separated tables (and you might already know, `Catalog`, `CatalogNode` and `CatalogEntry`), with the auto incremented identity. So a content with ID = 1 can be a `Catalog`, a `Node` or an `Entry`. Secondly, there can be multiple catalogs, while you need a “root” content to attach your content provider system.

The second one can be easily resolved by introducing a virtual root. It’s just another content, but virtual, you can never edit or delete it. It’s there, created on the fly whenever you need it. But the first one is a bit tricky. That was when `ReferenceConverter` came to life.

The content provider system requires every content to be identified by a specific `ContentReference`, which is supposed to be unique in the system. A `ContentReference` consists of three parts:

A sample catalog content reference: 123_1_CatalogContent

- The first one is the content id, which is mandatory. The ID is supposed to be unique in the system.
- The work id, which identifies the version of the content.
- The content provider name. This will allow the content system to know which provider should be handling the content. For catalog content, it's always `CatalogContentProvider` who does the leg works.



It was a big change in Commerce 9 in the way work id is handled. Prior Commerce 9, the work id is only unique within the catalog content. So it's possible to have both `1_3_CatalogContent` and `2_3_CatalogContent`. This was not consistent with how CMS handles the work id. In Commerce 9, the work id is unique across the system. So you can only have `1_3_CatalogContent` and `2_4_CatalogContent` and `2_5_CatalogContent` and `1_6_CatalogContent` and so on and so forth. This will introduce a limitation, as you can only have a maximum of 4294967295 catalog content versions in your entire system, but it's plenty even you have the biggest catalogs in your system. The change in the way work id is handled, was proved to improve the overall system performance.



CatalogContentProvider is the unsung hero who does all the hard work of processing catalog contents, but you should never work with it directly. The only APIs you should work with are IContentLoader and IContentRepository. Those will make sure to call the responsible content provider to handle the content it's processing (and doing other stuffs such as caching, raising events or so). I will even make a bold statement here: if you're using CatalogContentProvider directly, you are (probably) doing it wrong.



It might be a little confusing with the term of CatalogContent. It can mean either catalog content in general, or a content of catalog.

ReferenceConverter is a small class resides in Medichase.Commerce namespace. Its APIs are actually quite simple:

```

1  public class ReferenceConverter
2  {
3      /// <summary>
4      /// Gets a <see cref="ContentReference"/> instance
5  e with the specified commerce object ID and type
6      /// encoded in the content ID.
7      /// </summary>
8  public virtual ContentReference GetContentLink(int object\
9  Id, CatalogContentType contentType, int versionId)
10
11      /// <summary>
12      /// Gets the actual id of commerce object.
13      /// </summary>
14  public virtual int GetObjectId(ContentReference contentLi\
15  nk)
16

```

```
17         /// <summary>
18         /// Gets the type of the commerce object. Parse t\
19 he content ID to take two most significant bits to
20         /// determine CatalogContentType.
21         /// </summary>
22 public virtual CatalogContentType GetContentType(ContentR\
23 eference contentLink)
24
25     ...
26 }
```

These are two most interesting parts of the class. As I said above, the content id is supposed to be unique, while the catalog id, catalog node id and catalog entry id are not. The solution? Bitwise comes to rescue.^[^foo33]

Episerver Commerce use the first two bits of the id to store the content type. So basically:

- 00: CatalogEntry
- 01: CatalogNode
- 10: Catalog
- 11: Do you want to guess? Well, it's the Catalog root. You can always get the root's content link by `GetRootLink()` method of `ReferenceConverter`. It's always the boring same content reference every time.

So these methods work in an extremely efficient way to convert the id to `ContentReference` back and forth: From an id, add the two first bit (MSB - most significant bit), based on the content type, then add the fixed content provider name (CatalogContent) to get the `ContentReference`. From a `ContentReference`, clear the two first bits to get back the id. Easy, huh?



When the `CatalogContentProvider` was designed (it was called `CommerceContentProvider` at that time, prior to Commerce R3), there was an idea of having a forth part in the `ContentReference` to indicate the content type, so it'll become `12_3_CatalogContent_Entry`, or `12_3_CatalogContent_Node`. The idea was shot down by the architects, so we have less “human-readable” `ContentReference` as it is today. However, the `ContentReference` was not designed to be human-readable in mind (although it's quite easy to do so), it's the systems which need to understand it.



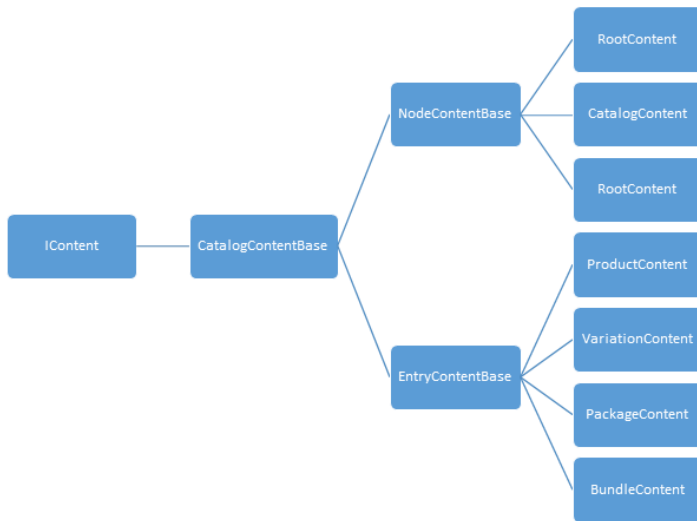
: Another interesting part of `ReferenceConverter` is the API:s to convert back and forth between the `ContentReference` and the code (of Entries and Node only, Catalog does not have that concept). While the `ContentReference` is what it needs for the content provider system to work, it's the code which the external systems, such as ERP, want to know to work with a specific node or entry. There is no bitwise or in-memory operations to help with that. Until very recently, the public virtual `ContentReference` `GetContentLink(string code, CatalogContentType type)` method of `ReferenceConverter` is quite slow and memory consumption. The most recent versions of Episerver Commerce improve it, while it's still no where near the bitwise performance, it's much better now. That's one of the reasons you should always stay with the most recent releases of Episerver.

Chapter 2: Catalog content modeling

The strongly typed models

Now you can have the basic ideas of how the catalog is structured - it's time to start working with some real code. As I mentioned above, you'll be working with `IContentRepository` (in some cases, with its base interface - `IContentLoader`), around `IContent`. Any content must implement `IContent` to be handled by the content providers - so the first thing to get the Catalog integrated into the content system (aka to get `CatalogContentProvider` to work) is modelling the catalog, node, and entry with `IContent`

Let's see how the catalog contents are modeled



Hierarchy of catalog content types, (forgive my drawing skills)

All catalog content types inherit from `CatalogContentBase`.

`RootContent` is a “virtual” content - it has no `Catalog` counterpart. You can only have one `RootContent` in the system, its `ContentLink` is fixed and it is always read-only. You can try to save it but nothing will be persisted. You will hardly work with `RootContent` directly, but with its `ContentReference`, via `ReferenceConverter.GetRootLink()`.

Let's us see a simple code of creating a `Catalog`, then editing its language, then deleting it. (We are just showing how the API:s work)

```

1      //Create a CatalogContent
2      var catalogContent = contentRepository.GetDefault<CatalogContent>(referenceConverter.GetRootLink());
3
4      catalogContent.Name = "Pro Episerver Commerce";
5      catalogContent.DefaultCurrency = Currency.USD;
6      catalogContent.DefaultLanguage = "en";
7      catalogContent.WeightBase = "kgs";
8      var catalogContentLink = contentRepository.Save(catalogContent, DataAccess.SaveAction.Publish, EPiServer.Security.AccessLevel.NoAccess);
9
10
11
12     //Edit it to add Swedish
13     catalogContent = contentRepository.Get<CatalogContent>(catalogContentLink).CreateWritableClone<CatalogContent>();
14
15     catalogContent.CatalogLanguages.Add("sv");
16     contentRepository.Save(catalogContent, DataAccess.SaveAction.Publish, EPiServer.Security.AccessLevel.NoAccess);
17
18
19
20
21     //Delete it
22     contentRepository.Delete(catalogContentLink, false, EPiServer.Security.AccessLevel.NoAccess);
23

```

One thing to remember about `CatalogContent` is it's different from `NodeContent` or `EntryContentBase` - it cannot be extended. While `CatalogNode` or `CatalogEntry` can be enriched by `MetaClasses` (as they implement `IMetaClass` interface), `CatalogContent` has no such abilities, you'll always work with `CatalogContent` directly, never with its subclasses.

This will simply does not work:


```
1 [CatalogContentType(GUID = "1FD3E34D-6476-40E2-8CB5-8EFB0\
2 DB79E3E")]
3 public class MyCatalogContent : CatalogContent
4 {
5     public virtual string MyExtendedProperty { get; set; }
6 }
```

CatalogContentScannerExtension will throw an EPiServerException exception during the initialization if it detects any content types inherit from CatalogContent:

The content type 'EPiServer.Commerce.Sample.MyCatalogContent' extends 'EPiServer.Commerce.Catalog.ContentTypes.CatalogContent', which is not supported.

Working with Nodes and Entries is more or less the same. But you'll have to model your metaclasses first.

Modeling catalog content types

It's not required to model your metaclasses. You can use the defined classes such as NodeContent or ProductContent, and throw everything into Property properties, instead of content.VeryLongPropertyName, you can use content.Property["VeryLongPropretyName"]. (Remember? Any instance of IContent has Property property as a "property bag" to store its properties). It's why Catalog UI (which relies on content types) will still work when you import a Catalog with metaclasses you didn't model after. But that will defeat one of the most important benefit of content types: strongly typed types mean Intellisense and compile-time checking. Noticed the typo I intentionally made above? Visual Studio will catch that for you.

```

1  /// <summary>
2  /// FashionProduct class, map to Fashion_Product_Class me\
3  tadata class
4  /// </summary>
5  [CatalogContentType(GUID = "18EA436F-3B3B-464E-A526-564E9\
6  AC454C7", MetaClassName = "Fashion_Product", GroupName = \
7  "Fashion",
8      DisplayName = "Fashion product", Description = "fashi\
9  on product with Add to Cart button.")]
10 [ImageUrl("~/Templates/thumbnail-fashion.png")]
11 public class FashionProduct : ProductContent
12 {
13     [Display(Name = "Description", Order = -15)]
14     public virtual XhtmlString Info_Description { get; se\
15 t; }
16
17     [Display(Name = "Features", Order = -11)]
18     public virtual XhtmlString Info_Features { get; set; }
19
20     [Display(Name = "Facet Brand", Order = -1)]
21     [Searchable]
22     public virtual string Brand { get; set; }
23 }

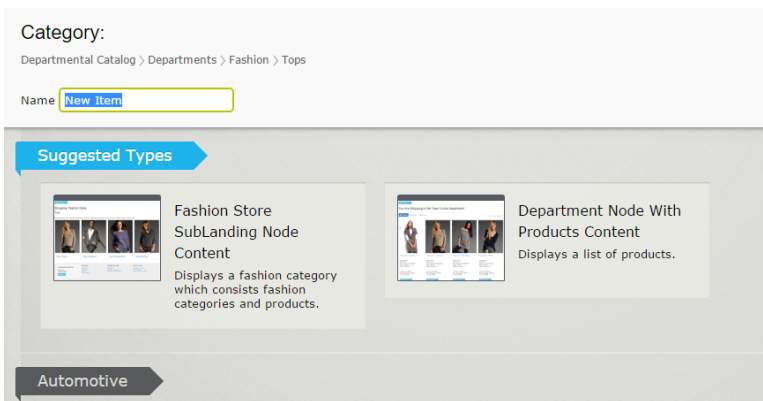
```

Catalog Content modeling uses attributes intensively. As you might see, there are plenty to explain:

- **CatalogContentType**: this is the most important attribute. It inherits from CMS **ContentType** attribute, which makes this content type an option to select when you create new content in Catalog UI. GUID is something required by CMS to identify the content type, even its name and its namespace is changed. It has **MetaClassName** property indicate that your class is mapped to the corresponding metaclass. In case the metaclass does not exist **GroupName** groups your content

type with other content types, so when you create a new content in CMS, they will in same group and be easier to select.

- Other properties is to make the display of your content type in Catalog UI nicer and more recognizable.
- ImageUrl1 thumbnail allow you to set a thumbnail to your content type.



How the attributes work in Catalog UI

When you define your strongly typed content, there are some rules to keep in mind:

- The property must match the name and the type of the metafield. Casing is not important. So if you have a string metafield named `facet_color`, your property must be string `Facet_Color`.



There was a plan to make your property name detached from the metafield name, it might be convenient at first though, but can be a problem of its own. That was never materialized.

- The property must be declared as virtual (An exception will be thrown if one or more properties are not declared virtual)
- As a metafield can be used across multiple metaclass, the property which maps to it also has to be identical across content types, including all of the attributes decorated. That means you cannot have a property named `Facet_Color` as string in one class and another named `Facet_Color` as double in another class. Or `Material` which has `CultureSpecific` attribute in one content type, and without that property in another. If one of the properties has mismatched attributes across classes, an exception will be thrown during site initialization.
- In most of the case, the underlying `MetaDataType` is automatically determined, so you don't have to do anything else. This is, however, not true with dictionary types. Both of them require you to specify a `BackingType`. In case of `MetaDataType.EnumSingleValue` and `MetaDataType.DictionarySingleValue`, if you fail to do so, it'll be created as a normal string, while `MetaDataType.EnumMultiValue` and `MetaDataType.DictionaryMultiValue` will throw exception during site startup.

Here's how you create them:

```

1  [BackingType(typeof(PropertyDictionaryMultiple))]
2  public virtual IEnumerable<string> MultipleValue { get; set; }
3
4
5  [BackingType(typeof(PropertyDictionarySingle))]
6  public virtual string SingleValue { get; set; }

```

These are attributes you can use to decorate the property, which also map to attribute of a metafield:

- **Required:** the property cannot be null
- **Searchable:** the property will be indexed by the search provider
- **Tokenize:** the value of the property will be tokenized while indexing. So if its value is “Hello world”, when `Tokenize` is set to true, then you can find this content with either “Hello” or “world”. Otherwise it will only be returned when you search for “Hello world”
- **IncludeValuesInSearchResults:** Same as “Include value in search results” setting for metafield.
- **IncludeInDefaultSearch:** same as “Include in default search” setting for metafield.
- **Encrypted:** Indicates that the property will be saved to database (as a metafield) encrypted. This attribute, however, does not take effect, if your database is set to be Azure-compatible.
- **DecimalSettings:** Allow you to set the precision and scale of a decimal type. Note that the highest precision of decimal type stored in SQL Server is (38,9), so anything bigger than that will not be allowed.

When your site starts, `CatalogContentScannerExtension` will do all the leg works to scan the content types and synchronize them to the metadata system - content types to metaclasses and and properties to metafields. In case of any conflicts between those pairs, catalog content types win. For example, you have a metafield with decimal precision of (18,0), but the mapping property has `DecimalSettings` to be (33,9), `CatalogContentScannerExtension` will update the metafield to be (33,9), not the way around.

IDimensionalStockPlacement

Most of properties of the base content types provided by Optimizely Commerce (`CatalogContent`, `NodeContent`, `ProductContent`, etc.) are flat, which means you can access directly, for example:

```
1 nodeContent.Code = "This-is-a-code";
```

But the contents implement `IDimensionalStockPlacement` (by default `VariationContent` and `PackageContent`), are exceptions. They are accessible via a block - `ShippingDimensions`.

```
1 variationContent.ShippingDimensions.Height = 10m;
```



Unlike other fields which have existed since before Commerce 7.5, shipping dimensions were added in a later version. At that time, it was already possible for some customers to add the properties for dimensions themselves, and having properties named `Height`, `Width` or `Length` in your `VariationContent` is not very uncommon. If Episerver added the property to `VariationContent`, it might have broken some customer sites which have the conflict - so they chose the safe way and added a block instead. That's why the shipping dimensions are displayed a bit differently in Catalog UI:

That's why the shipping dimensions are displayed a bit differently in Catalog UI:

Content	Belongs To	Pricing	Inventory	Assets
Min. quantity	<input type="text" value="1"/>			
Max. quantity	<input type="text" value="50"/>			
Weight	<input type="text" value="5"/>			
Shipping Package	<input type="text" value="box"/> ▼			
<h3>Shipping Dimensions</h3>				
Length	<input type="text" value="0"/>			
Height	<input type="text" value="0"/>			
Width	<input type="text" value="0"/>			

Shipping dimensions as a block

If you have worked with CMS, you might already know that you don't have to initialize a block before using it - it'll never be null for a content. So you don't have to write code such as:

```
1 variationContent.ShippingDimensions = new ShippingDimensi\
2 ons();
```

Just use the block directly, as we showed above.

The MetaData system

It's hard to fully understand the Catalog system without knowing about the MetaData system, with its MetaClass(es) and

MetaField(s). The Catalog is just the barebone, it's the MetaData system which bring it to life.



It's important to remember that there are two classes named `MetaClass` in Optimizely Commerce. The one resides in `Mediachase.MetaDataPlus.Configurator` is the one we are talking about. Yes, it's not the best to have two classes named exactly the same in your framework, even if they are in two separated assemblies. I must admit I am still confused by them, and it takes me a few seconds to navigate to the correct class. However, they have been there since forever and any changes would become a big breaking change. We'll have to live with it for now.

The idea of metaclasses and metafields is simple. To make it effective to editing your catalog, you'll need pre-defined templates. So a shirt needs a style (tailor fit, slim fit,...), a material (silk, cotton, polyester,...), a color (red, white, blue), a texture,..., while a TV needs a size (40", 50", 55",...), a resolution (HD ready, FullHD or 4k), Smart features (Netflix, Youtube, Browser), 3D or not, etc. As a framework, Episerver cannot predefine all of those things out of the box - it's up to you to define your "templates" for your needs. So each template is a metaclass, and each attribute of the template, is a metafield.

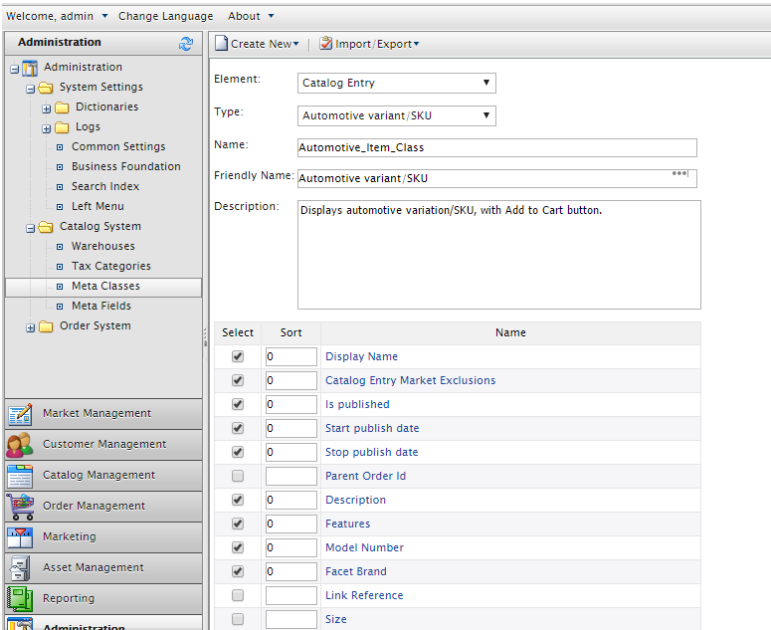
And each instance of the template, is a `MetaObject`. `MetaObject` is, in the simplified view, just a `HashSet` of values.



The metadata system is used for both catalog and order. However, it is used much more intensively for catalog. The default catalog metaclasses are really simple and can't almost be used in anything. The default order metaclasses, in other hands, are much more comprehensive. In most implementations you'll see, the catalog metaclasses are mostly user-defined, while you can start using the order with just one or two additional metafields.

At API:s level, there's nothing really interesting about `MetaClass(es)` or `MetaField(s)` in particular. Today, you will be hardly working with them - you should be working with the strongly typed models instead, which will come up in next part. However, they are still the internals of the catalog system, so it'll make sense for us to explore how they works.

Let's take a closer looks at `MetaClass`. At its core, it's a simple class with some attributes (`Id`, `Name`, `TableName`, `Namespace`, etc) and a list of `MetaFields` attach to it. If you take a look at the `MetaClass` management in Commerce Manager (Commerce Manager/Administration/Catalog System/`MetaClass`), it's as close as bare metal you can get (without looking at database, of course)



A MetaClass

The key feature of MetaData system is the extensibility. You can attach or remove a metafield from a metaclass on-the-fly. You can share metafields between metaclasses.



There is no hard limit of how many metafields can be in a metaclass, but my opinion is you should not exceed 50.

It's metafield which is more interesting

Name:

Friendly Name:

Description:

Type:

☒ Supports Multiple Languages

☒ Use in comparing

☐ Allow Null Values

Search Properties:

☐ Allow Search

☐ Enable Sorting Search Results

☐ Include Values in Search Results

☐ Tokenize

☐ Include in the Default Search

Create a MetaField

What do these settings mean?

- Support multiple languages: That mean the metafield will be per-language, each language version of the MetaObject will have an unique value.
- Use in comparing: This is a legacy setting which no longer has any meaning. (It should have been removed)
- Allow Null Values: This metafield is not required.
- Encrypted: The metafield will be encrypted by the master key in database and be decrypted in the fly. This option, however, does not show up if your database is set to be

Azure-compatible. (As Sql Azure Database does not support encryption at the time of this writing).

- Precision: This only set-able if your metafield is a decimal. This was important in the past because it will affect how the decimal is stored in the database, but with Commerce 9, the decimal will always be stored in the (38,9) form.

Search attributes: These attributes will affect how will this metafield be indexed in a Lucene-based search provider:

- Allow Search: This metafield will be indexed by the search providers.
- Enable Sorting Search results: The value of this metafield can be used for sort by the search providers.
- Include Value in search results:
- Tokenize: The value of this metafield will be tokenized. So if its value is “Hello World”, you can either search with “Hello” or “World”
- Include in Default Search: This metafield will be include in the default Lucene query.

If you can only choose between either `CatalogEntry` or `CatalogNode` for `MetaClass`, then it's whole lot more thing to choose for metafield. The full list of `MetaType` which you can choose from: `{lang=C#} public enum MetaDataType { Integer = 26, Boolean = 27, Date = 28, Email = 29, URL = 30, ShortString = 31, LongString = 32, LongHtmlString = 33,`

```
1      DictionarySingleValue = 34,
2      DictionaryMultiValue = 35,
3      EnumSingleValue      = 36,
4      EnumMultiValue       = 37,
5      StringDictionary     = 38,
6      File                  =      39,
7      ImageFile             =      40,
8
9      MetaObject            =      41
10 }
```

The only caveat of choosing a type is that you can't change the type of a metafield after you saved it. So if you make a wrong move, it might be easiest to just delete the metafield and create again.

Then how MetaObjects are stored?

Here's the map between PropertyData types and MetaField types:

MetaDataType	PropertyData type
MetaDataType.Decimal	PropertyDecimal
MetaDataType.BigInt	
MetaDataType.Money	
MetaDataType.SmallMoney	
MetaDataType.DateTime	PropertyDate
MetaDataType.SmallDateTime	
MetaDataType.Date	
MetaDataType.Float	PropertyFloatNumber
MetaDataType.Real	
MetaDataType.Bit	PropertyNumber
MetaDataType.Char	
MetaDataType.Int	
MetaDataType.Integer	
MetaDataType.SmallInt	
MetaDataType.TinyInt	PropertyNumber
MetaDataType.NChar	PropertyLongString
MetaDataType.VarChar	
MetaDataType.NText	

MetaDataType	PropertyData type
MetaDataType.Text	
MetaDataType.NVarChar	
MetaDataType.LongString	
MetaDataType.Boolean	PropertyBoolean
MetaDataType.Email	PropertyEmailAddress
MetaDataType.URL	PropertyUrl
MetaDataType.ShortString	PropertyString
MetaDataType.UniqueIdentifier	
MetaDataType.LongHtmlString	PropertyXhtmlString
MetaDataType.EnumMultiValue	PropertyDictionaryMultiple
MetaDataType.DictionaryMultiValue	
MetaDataType.EnumSingleValue	PropertyDictionarySingle
MetaDataType.DictionarySingleValue	
MetaDataType.Timestamp	Unsupported
MetaDataType.Variant	
MetaDataType.Numeric	
MetaDataType.Sysname	
MetaDataType.Binary	
MetaDataType.VarBinary	
MetaDataType.StringDictionary	
MetaDataType.Image	
MetaDataType.File	
MetaDataType.ImageFile	
MetaDataType.MetaObject	

How properties are stored

It might not be particularly useful to look into how properties are stored in database, but it can be interesting to know the detail (You might someday look into your databases to investigate, when a problem arises.)



Optimizely does not officially disclose database schema and stored procedures, and those might be changed them without notices so think twice before relying on the internal database schema.

In Commerce 9, the way properties are stored is fundamentally changed.

If you have worked with CMS content properties before, you might know about `tblContentProperties`. Commerce 9 adapts the same idea.

I	MetaFieldName	LanguageName	Boolean	Number	FloatNumber	Money	Decimal	Date	Binary	String	LongString
1	DisplayName	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	Tops-Turics-LongSleeve
2	_ExcludedCatalogEntryMarkets	en	NULL	1	NULL	NULL	NULL	NULL	NULL	NULL	NULL
3	Epi_IsPublished	en	1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	Epi_StartPublish	en	NULL	NULL	NULL	NULL	NULL	2010-09-01 07:00:00.000	NULL	NULL	NULL
5	Epi_StopPublish	en	NULL	NULL	NULL	NULL	NULL	2020-10-01 07:00:00.000	NULL	NULL	NULL
6	Info_Description	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	<p>Lorem ipsum dolor et amet. co
7	Info_Features	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	 Feature 1 Featu
8	Info_ModelNumber	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	1034321
9	Facet_Brand	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	Brand 1
10	DisplayName	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	Departments
11	Epi_IsPublished	en	1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
12	Epi_StartPublish	en	NULL	NULL	NULL	NULL	NULL	2010-09-01 07:00:00.000	NULL	NULL	NULL
13	Epi_StopPublish	en	NULL	NULL	NULL	NULL	NULL	2020-10-01 07:00:00.000	NULL	NULL	NULL
14	Info_Description	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	<p>Integer posuere erat a ante ve
15	DisplayName	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	Tops-Turics-OffShoulder

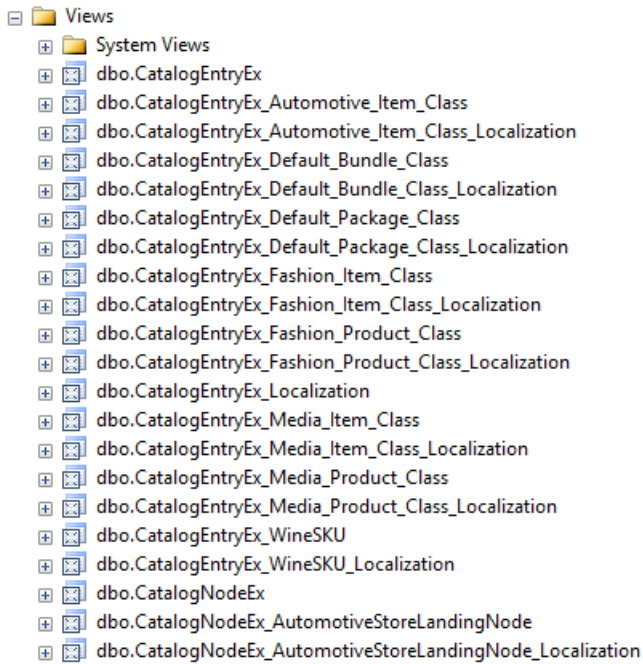
CatalogContentProperty table, it's new in Commerce 9

Commerce 9 has a new table - `CatalogContentProperty` to store property. Besides the identity columns, there are 10 columns for the datatypes `MetaDataPlus` supports: `Boolean`, `Number` (`Integer`), `FloatNumber`, `Money`, `Decimal` [`^foo151`], `Date`, `Binary`, `String` (for strings shorter than 256 characters length), `LongString` (any strings long than that) and `Guid`. A metafield, for each content and for each language (in case it's culture specific) will be stored in one row matches with it value. So a string property value will be in `String` column, while an `Integer` property value will be in `Number` column, and so on and so forth, while the other columns remain null for that row.

It's hard to say if the old or new approach is definitely better than the other. The new one might be more flexible, but it means more rows and is not as “natural” as the old one. However, the new one has simply access approach - you only read or update or delete from one table, while the old one has to rely on the automatically generated stored procedures (which are updated every time you add or remove metafields to metaclasses). In long run, the new approach might be simpler to maintain. Let's hope for that.

When you upgrade your site to Commerce 9, all old metaclass tables

are dropped. Of course, it does not really make sense to keep the tables you no longer update. However, you might still have custom queries to work on that tables (such as to report something on your catalog). To replace those tables, new views are created. They are created by querying the data from CatalogContentProperty table, and are updated every time you update your metaclasses (just like the way old metaclass tables did).



The new views to replace metaclass tables

Those views are, as you can expect, slower than the original tables, some tests shows that they can be 30% slower than the original tables. However, you'll not use them frequently enough for the difference to really make senses.

Dictionary types.

Previously we discussed on how properties work with catalog content. However - if you have dictionary types in your MetaClasses, they will work differently. In this section we will examine these special data types - this applies to Order system metaclasses as well.

As we all know - there are three types of dictionary in Episerver Commerce:

- Single value dictionary: editor can select a value from defined ones.

Name:

Friendly Name:

Description:

Type:

☐ Supports Multiple Languages

☐ Multiline

☐ Editable

☐ Use in comparing

☒ Allow Null Values

Search Properties:

☐ Allow Search

☐ Enable Sorting Search Results

☐ Include Values in Search Results

☐ Tokenize

☐ Include in the Default Search

In Commerce Manager, you can create new metafield with type of Dictionary, but without “Multiline” option

Single value dictionary type is supported in the strongly typed content types - you’ll need to define a property of type `string`, with backing type of `typeof(PropertyDictionarySingle)`

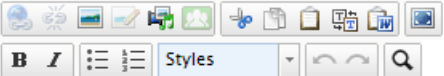
```
1 [BackingType(typeof(PropertyDictionarySingle))]  
2 public virtual string Color { get; set; }
```

- Multi value Dictionary: editor can select multiple values from defined ones. The only different from Single value dictionary is it has the “Multiline” option enabled.

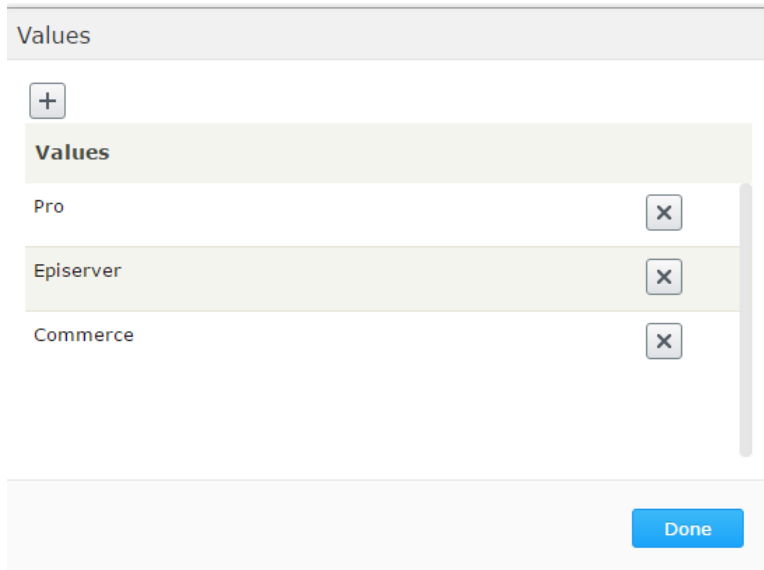
You can define a property for Multi value dictionary in content type by `IEnumerable<string>` and `typeof(PropertyDictionaryMultiple)` backing type

```
1 [BackingType(typeof(PropertyDictionaryMultiple))]  
2 public virtual IEnumerable<string> Colors { get; set; \  
3 }
```

Both single and multi value dictionary types are fully supported in Catalog UI, including editing:

Content	Belongs To	Pricing	Inventory	Assets
<p>Description</p> <div><p>Item Specific Description Etiam porta sem n faucibus mollis interdum. Lorem ipsum dolo</p><p>Path: p</p></div>				
<p>MultiValueDictionary</p> <ul style="list-style-type: none"><input checked="" type="checkbox"/> Pro<input checked="" type="checkbox"/> Episerver<input checked="" type="checkbox"/> Commerce				

and administration:



The screenshot shows a 'Values' settings panel. At the top is a header 'Values' with a '+' button to its left. Below the header is a scrollable list of values. The list contains three items: 'Pro', 'Episerver', and 'Commerce'. Each item is on a light yellow background and has a small 'X' button to its right. At the bottom right of the panel is a blue 'Done' button.

Manage values for a dictionary field in Settings view



There is a bug in Commerce 11.5 and before, when you don't have a strongly typed content type for a metaclass containing a dictionary type metafield, the property will be rendered incorrectly in Catalog UI. This is fixed in Commerce 11.5.1

- String dictionary: this is the true “dictionary type”: you can define pairs of key and value (the previous types are actually “list”).

String dictionary is not supported by the strongly typed content types, nor Catalog UI. To manage this metafield, we'll have to use Commerce Manager, or use MetaDataPlus API:s

StringDictionary:

Key:
 Value:

Key	Value
Book	Pro Episerver Commerce
blog	http://vimvq1987.com

StringDictionary

Pairs of key and value can be managed directly in CatalogEntry edit view in Commerce Manager

In later chapters, we will see how to add the support for it in Catalog UI.

How dictionary works

We've learned in previous chapters how properties are stored and loaded. Technically, dictionaries are “just another properties”. However they are stored differently. If you look at CatalogContentProperty table (or ecfVersionProperty, for draft versions), you'll see the dictionary properties are stored as numbers. What do those numbers mean?

- For single value dictionary type, that number is the MetaDictionaryId of the selected value in MetaDictionary table.
- For multi value dictionary type, things are a bit more complicated. That number is the MetaKey value in MetaKey table. This MetaKey, is, however, connected to MetaMultiValueDictionary, which itself points back to MetaDictionary. An “usual” design for 1-n relation in database, right?
- For string dictionary type, it's more or less the same as multi value dictionary. However, the MetaKey will point to pairs of key and value in MetaStringDictionaryValue table.

Chapter 3: Associations, relations and assets

If you work with CMS before - prepare to be surprised. How catalog entities are related to each other is much, much more complicated than how CMS contents are.

Associations are the connections between products, so you can cross sell, or up sell other products. For example, if you are selling a table, you would want your customers to buy a chair.

Relations are even more complicated. You have relations between nodes (how node are linked together), between nodes and entries (how entries belong to nodes), between entries (how variants belong to products, how variants belong to packages and bundles).

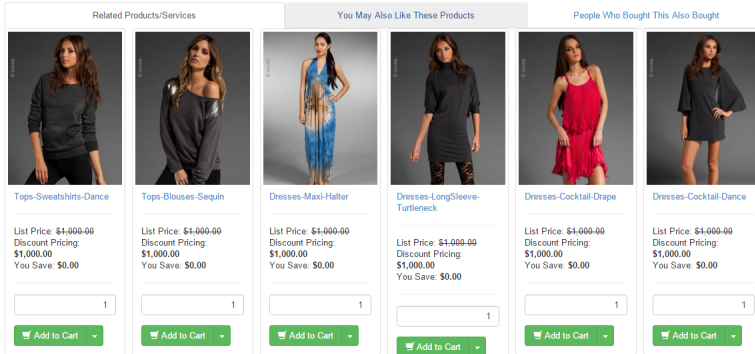
Assets - or what we will be talking about - is how to associate CMS assets (which are “contents”) to products and categories. Products can be boring without all images, videos, and product manuals - and that’s why we need them.

Working with associations and relations

From Optimizely Commerce 11, `IRelationRepository` and `IAssociationRepository` are the main API:s you would use to work with relations and associations, respectively.

`IAssociationRepository` is a quite simple interface with only three methods which are all easy to understand (which comes from the nature of Associations, as we learned in previous section). Just to

remind you, an association defines two entries to have a connection - they can be sold together, customer might be suggested the alternatives.



A sample of how associations can be used to up sell your products

```

1 public interface IAssociationRepository
2 {
3     /// <summary>
4     /// Gets the associations for the catalog content spe\
5 cified by the content link.
6     /// </summary>
7     IEnumerable<Association> GetAssociations(ContentRefer\
8 ence contentLink);
9
10    /// <summary>
11    /// Removes the associations.
12    /// </summary>
13    void RemoveAssociations(IEnumerable<Association> asso\
14 ciations);
15
16    /// <summary>
17    /// Updates matching associations and adds new associ\
18 ations for an entry.
19    /// </summary>

```

```

20     void UpdateAssociations(IEnumerable<Association> asso\
21     ciations);
22 }

```

Associations and Relations groups.

When you define an association or relation, you can set “group” of it. It’s just a custom string so you can filter the associations later on. For example, you can have a association group named CrossSell to define the products which can be suggested to customers when they have a product in cart. By default, there is only one group for association (“default”) and one group for relation (well, “default”, too) in the system. Those can’t be set by the Catalog UI, only by code. You can define more association groups by adding this to your initialization module:

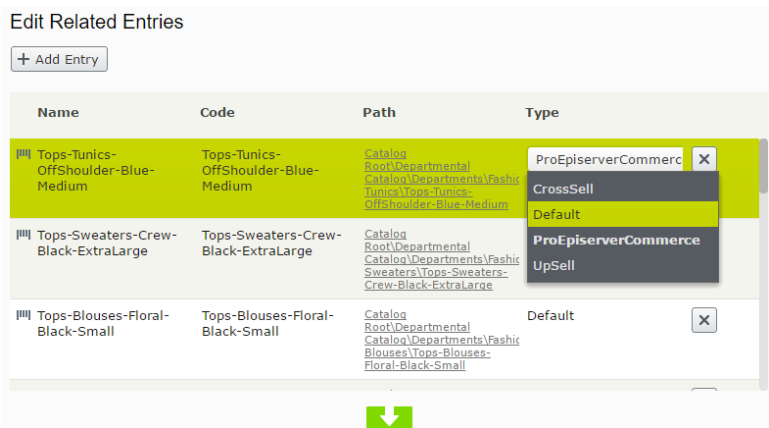
```

1     var associationDefinitionRepository =
2         context.Locate.Advanced.GetInstance<GroupDefiniti\
3         onRepository<AssociationGroupDefinition>>();
4         associationDefinitionRepository.Add(new AssociationGr\
5         oupDefinition { Name = "CrossSell" });
6         associationDefinitionRepository.Add(new AssociationGr\
7         oupDefinition { Name = "Upsell" });
8         associationDefinitionRepository.Add(new AssociationGr\
9         oupDefinition { Name = "ProOptimizelyCommerce" });

```

These association groups will be added if they haven’t existed in your system, otherwise nothing is change. Now when you edit an association in Catalog UI, you can choose one of those:

You can do the same with relation by using `RelationGroupDefinition`, however, this is not used elsewhere in the Catalog UI. All the entry relations are “default” by the way.



You can now choose between the defined association groups

Relations in Commerce

There are essentially three types of relations between entities in Commerce: - Relations between nodes - Relations between nodes and entries - Relations between entries

Those are presented by these three types, respectively, `NodeRelation`, `NodeEntryRelation`, and `EntryRelation`. `EntryRelation` itself is an abstract class, and has 3 implementations:

- `ProductVariation` is for relation between a product and its variants.
- `BundleEntry` is for relation between a bundle and its content.
- `PackageEntry` is for relation between a package and its content.

All classes above inherit from `Relation`, and are all managed by `IRelationRepository`



Episerver Commerce obsoletes APIs that no longer used, contain typos (yes, that happens!), or with bad performance all the time, and they will normally stay at least for 12 months. However, it's always a good idea to try to fix the obsolete warnings whenever you upgrade and they appear. Generally, I recommend to turn on the option to "treat warnings as errors" in your core projects. Unless you are absolutely sure that a warning is harmless, it's better to fix it sooner than later.

Another big change in how relations are handled in Commerce 11 is the introduction of `IsPrimary` property for `NodeEntryRelation`.

As we learned earlier, an entry can belong to multiple nodes, but only one of them is "true" parent - primary node, and others are linked. However, it is not easy to tell which is primary, and which is linked. In 10.x and earlier, the way to determine the primary node is based on the the lowest `SortOrder` value. That was based on a wrong assumption, and it comes with the cost of unable to sort the entries within a node. To do that, the `SortOrder` needs to change and it will screw up the relations.

In Commerce 11, the problem has been corrected. `SortOrder` is now just ... sort order. Node-entry relation has now a new property named `IsPrimary` - and for all the node relations of an entry, only one can has that as true - and that will be the primary node - and all other ones are linked. When you upgrade to Commerce 11, the relation with lowest `SortOrder` will be selected to be primary (because it was supposed that way). After that, `SortOrder` will be truly what it is meant to be: You can drag and drop the entries in a node. Let's say you have a category of iPhone - and there are plenty of model being sold. At this time of writing, iPhone X is the hot shot, and you would naturally want it to be on top of your category. How would you do it - drag and drop it to the top, baby



Sorry, no iPhones, I could have renamed the entries, but I want to be honest



The APIs to explicitly set the `IsPrimary` property for a node-entry relation is only available in Commerce 11.2, with the new type - `NodeEntryRelation` - it inherits from `NodeRelation` with `IsPrimary` property - Now you see me saying it again - upgrade to latest version when possible, because it means new features (and other good things). One entry can only have one primary node, so if you are setting a new primary node, it will take over the current one.

Assets

Products always need some assets. Images, videos, documentations such as specifications and manuals,...As a developer/an editor, you'll need a way to work with assets.

In the old days, you manage assets via `ICatalogSystem` - by accessing `CatalogItemAsset` datatable directly. Since Commerce 7.5, everything can, and should be done via the `IAssetContainer` interface (which is implemented by `EntryContentBase` and `NodeContent`), and is accessible as `CommerceMediaCollection`.

As mentioned above, we'll work with `CommerceMediaCollection`, which itself is a collection of `CommerceMedia`, as following:

```
1 public class CommerceMedia : ICloneable
2 {
3     public ContentReference AssetLink { get; set; }
4
5     public string AssetType { get; set; }
6
7     public string GroupName { get; set; }
8
9     public int SortOrder { get; set; }
10
11     //Other methods and properties omitted for brevity.
12 }
```

Those properties should explain themselves. `AssetLink` is the `ContentReference` to the asset (which is supposed to be another media content), `AssetType` is the type of class you defined to map with the asset (we'll talk about it later). `GroupName` is the group you want the asset to be, for easier management (such as "videos", or "images", or "manual"). `SortOrder` is the order they appear in the Asset tab in Catalog UI, with one convention: the first in the list will be the default one.







One small mistake that I have seen even experienced developers made - if you want to make change to the `CommerceMediaCollection`, you make a writable clone the content (by using `CreateWritableClone<T>()`), not to clone the `CommerceMediaCollection` itself. This applies to all other properties of the content.

Content
Belongs To
Pricing
Inventory
Assets
Related Entries
Settings

Media for this entry

[+ Add Media](#)

Name	Group
 Tops-Tunic-CowlNeck-Black.jpg	small
 Tops-Tunic-CowlNeck-Blue.jpg	default
 Tops-Tunic-CowlNeck-Chocolate.jpg	default



You can drop [media](#) here

Assets, ordered by SortOrder

You can add any asset which the type implements `IContentMedia`, but it would be more convenient if you define your class like this:

```

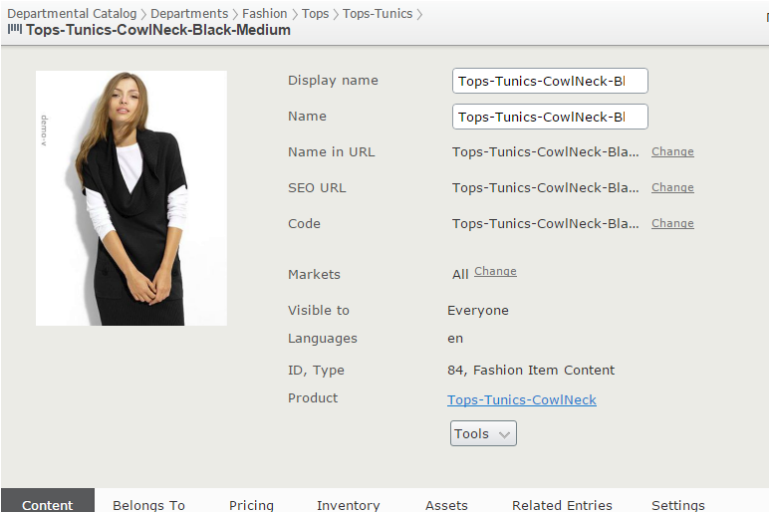
1  using System;
2  using EPiServer.Commerce.SpecializedProperties;
3  using EPiServer.DataAnnotations;
4  using EPiServer.Framework.DataAnnotations;
5
6  namespace EPiServer.Commerce.Sample.Models.Files
7  {
8      [ContentType(GUID = "872AA39E-5B79-43BF-B7D5-F34D4155\
9  53BD")]
10     [MediaDescriptor(ExtensionString = "jpg,jpeg,jpe,ico,\
11 gif,bmp,png")]
12     public class ImageFile : CommerceImage
13     {
14         /// <summary>
15         /// Gets or sets the description.
16         /// </summary>
17         public virtual String Description { get; set; }
18     }
19 }
```

This is pure “content” stuff. The most important thing in this example class is the `MediaDescriptor` attribute, which allows you to specify which extensions you want to handle by this class. So in this example, any jpg, jpeg and so on files uploaded to Media gadget will be handled by `ImageFile`, and if you drag and drop that file into the Asset list of a catalog content, its `AssetType` will be `EPiServer.Commerce.Sample.Models.Files.ImageFile`.

There are two interesting things about `CommerceImage`. It has two properties like this:

```
1 [ImageDescriptor(Width = 256, Height = 256)]
2 public virtual Blob LargeThumbnail { get; set; }
3
4 //Inherited
5 [ImageDescriptor(Width = 48, Height = 48)]
6 public override Blob Thumbnail { get; set; }
```

The first one belongs to the `CommerceImage` itself, it defines a `Blob` named `LargeThumbnail` with the size of 256x256. Well, as you might guess, it's the big header image whenever you edit an entry or node in Catalog UI:

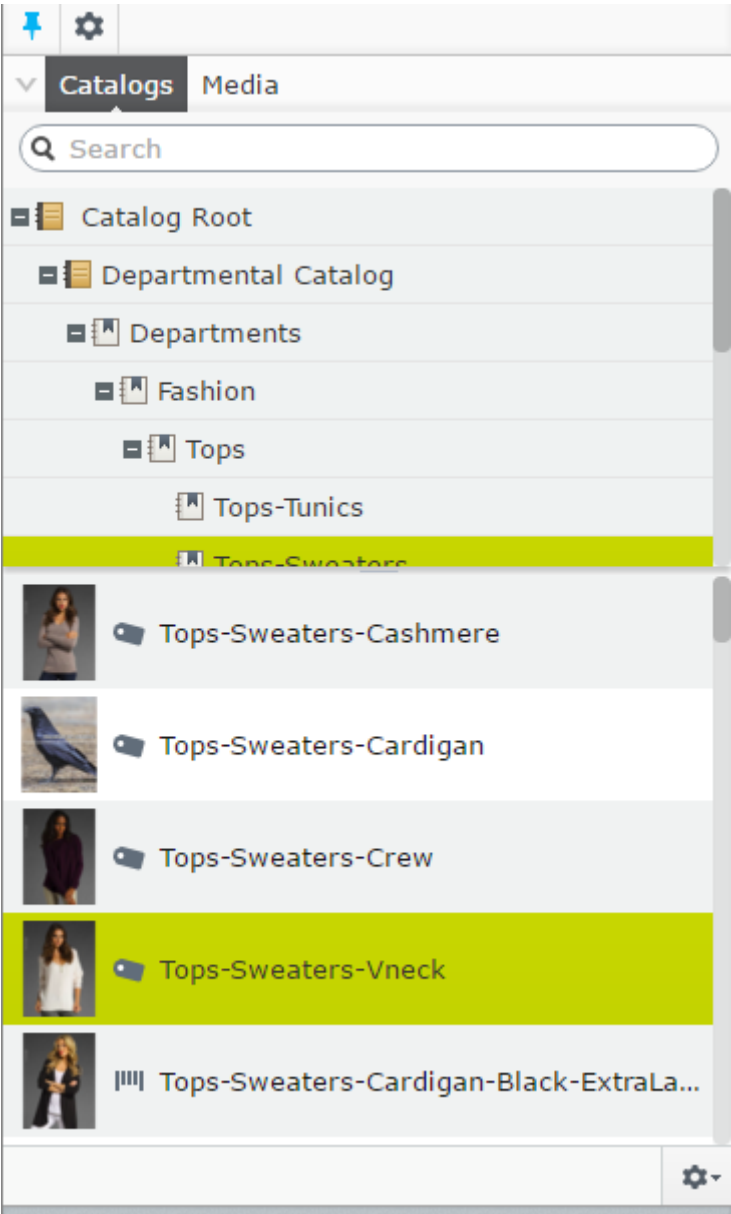


Thumbnail of a catalog content



If you forget to inherit your image content types from `CommerceImage`, the content header will be broken in Catalog UI. Because no `LargeThumbnail` is found, the full-size image will be used, and that breaks the layout.

The second one is inherited from `MediaData`, it defines a `Blob` named `Thumbnail` with the size of 48x48. It's the thumbnail in the entries list of Catalogs gadget:



Catalog entry list of Catalogs gadget

The first asset of type `CommerceImage` in your asset list (which has smallest `SortOrder` value) will be taken to generate those thumbnails. They are background-transparent, and if you don't like the size, just override it in your class:

```
1  [ContentType(GUID = "872AA39E-5B79-43BF-B7D5-F34D415553BD\  
2  ")]  
3  [MediaDescriptor(ExtensionString = "jpg,jpeg,jpe,ico,gif,\  
4  bmp,png")]  
5  public class ImageFile : CommerceImage  
6  {  
7      /// <summary>  
8      /// Gets or sets the description.  
9      /// </summary>  
10     public virtual String Description { get; set; }  
11  
12     [Editable(false)]  
13     [ImageDescriptor(Width = 128, Height = 128)]  
14     public override Blob LargeThumbnail { get; set; }  
15  
16     [Editable(false)]  
17     [ImageDescriptor(Width = 64, Height = 64)]  
18     public override Blob Thumbnail { get; set; }  
19 }
```

Now the thumbnail header should be only 128x128. The thumbnail list of Catalogs gadget, however, is styled by an CSS class, so it's not resized automatically on the UI. If you want to, you'll have to override this class yourself:

```

1 .Sleek .epi-thumbnailContentList.dgrid .dgrid-row .epi-th\
2 umbnail {
3     width: 48px;
4     height: 48px;
5 }

```



It might be interesting to look into how those Blobs work. For each Blob property of a MediaData content, CMS will create a PNG resize for it. If you look at the blobs folder where the assets are stored, you can see the created thumbnails are placed in same folder with the original picture, but with the property names as suffix. For example if your file is `1741f024affb4057952419058cccf599.jpg` then the thumbnails will be `1741f024affb4057952419058cccf599_Thumbnail.png` and `1741f024affb4057952419058cccf599_LargeThumbnail.png`. If you have other Blob properties, those will be created as well. To save the resource, the blobs are only created for the first request, and then they are stored. After that, they can be accessed by the URL. For example, if the link to your asset is `/globalassets/catalogs/tops/sweaters/tops-sweaters-crew.jpg/`, then you can access your thumbnail at `globalassets/catalogs/tops/sweaters/tops-sweaters-crew.jpg/LargeThumbnail` or `globalassets/catalogs/tops/sweaters/tops-sweaters-crew.jpg/Thumbnail`. There is a scheduled job in CMS to clean all the thumbnail blobs. Mind you, it is resource intensive, especially for the CPU and should be run with care if your site has a lot of assets.

There are two limitations regarding the AssetType:

- The full name of the type must not exceed 190 characters in length. It's the maximum length of the column for

`AssetType` in `CatalogItemAsset` table to fit into an index. `CatalogContentScannerExtension` will validate this during the site start up, and throw an exception if it finds any media class which does violate that rule.

- Asset type must be resolve-able when you import or export the catalog. If you import or export your catalogs in the context of Commerce Manager, the asset mappings with entries and nodes will be skipped.

Catalog content versions

One of the most important features in `CatalogContentProvider` is it bring versioning to catalog content. It was somewhat limited with Commerce 7.5 (the languages handling was quite sloppy), but it has been much more mature since Commerce 9. The versioning system in Commerce 9 is now more or less on par with versioning in CMS, and it's a good thing.

If you're new to Episerver CMS/Commerce, then it might be useful to know how version and save action work in content system. Of course, you can skip this section if you already know about it. The version status is defined in `EPiServer.Core.VersionStatus`. When you save a content, you have to pass a `EPiServer.DataAccess.SaveAction` to `IContentRepository.Save` method.

The documentation for those enum:s are pretty good, and the combinations of `SaveActions` can be quite complicated, but we can consider a basic case so you'll get the idea:

```
1  var parentLink = ContentReference.Parse("1073741845__Cata\
2  logContent");
3  var contentRepo = ServiceLocation.ServiceLocator.Current.\
4  GetInstance<IContentRepository>();
5
6  //Unsaved content, should have status of NotCreated.
7  var variationContent = contentRepo.GetDefault<VariationCo\
8  ntent>(parentLink);
9  variationContent.Name = "New variation";
10
11 //Save the content, now it is CheckoutOut
12 var variationLink = contentRepo.Save(variationContent, Sa\
13 veAction.Save, AccessLevel.NoAccess);
14
15 //A saved content is read only. To edit it, we must creat\
16 e a "writable" clone
17 variationContent = contentRepo.Get<VariationContent>(vari\
18 ationLink)
19     .CreateWritableClone<VariationContent>();
20 variationContent.Code = "New-variation";
21
22 //Check in, in the UI, it's Ready to Publish, which mean \
23 the edit was complete.
24 //The content status is now CheckedIn.
25 variationLink = contentRepo.Save(variationContent, SaveAc\
26 tion.CheckIn, AccessLevel.NoAccess);
27 variationContent = contentRepo.Get<VariationContent>(vari\
28 ationLink);
29
30 //Oops, made a typo. reject it.
31 variationContent = contentRepo.Get<VariationContent>(vari\
32 ationLink)
33     .CreateWritableClone<VariationContent>();
34 //Now it's Rejected.
35 variationLink = contentRepo.Save(variationContent, SaveAc\
```

```
36 tion.Reject, AccessLevel.NoAccess);
37
38 //Correct the mistake.
39 variationContent = contentRepo.Get<VariationContent>(vari\
40 ationLink).CreateWritableClone<VariationContent>();
41 variationContent.Code = "New-variation";
42
43 //Publish it directly. Use ForceCurrentVersion flag so no\
44 new version will be created
45 //it will overwrite the "rejected" version.
46 variationLink = contentRepo.Save(variationContent,
47     SaveAction.Publish | SaveAction.ForceCurrentVersion, \
48     AccessLevel.NoAccess);
```

One thing to remember about the code above is that we used the versioned-ContentReference:s (ContentReference with WorkId bigger than 0). By default, if you pass a ContentReference without version to `IContentRepository.Get<T>`, you will get back the published version (of the master language), or the CommonDraft version if there is no published version available. With WorkId, a specific version is returned (given that version exists).

One fundamental change in Commerce 9 versioning is the uniqueness of WorkId, as we mentioned earlier. Prior to Commerce 9, WorkId is only unique for a specific content, but now it's unique across system. It does mean from the WorkID, you can know anything, from the content itself to the version you're pointing to. This also means WorkId triumphs everything else. So for some reasons, you get your ContentReference wrong, such as the ID points to a content, but the WorkId points to a version belongs to another content, then the WorkId wins, and the content returned is the content WorkId points to (In Commerce 8, the content ID wins). That's why you should always make sure you get the correct version of ContentReference. If you want to load a version with a specific status without knowing its WorkId, make sure can use `IContentVersionRepository`. For example, to get the latest "Previ-

ously Published” version in English:

```

1  var contentLink = ContentReference.Parse("83__CatalogCont\
2  ent");
3  var versions = contentVersionRepository.List(contentLink);
4  var previouslyPublished = versions.OrderByDescending(c =>\
5    c.Saved)
6    .FirstOrDefault(v => v.Status == VersionStatus.Pr\
7    eviouslyPublished && v.LanguageBranch == "en");

```

The unique `WorkId` across system is, again, consistent with CMS. The other changes, comes at a much lower level - database.

Versioning was reason catalog system in Commerce 7.5 was significant slower than Commerce R3. The non-version parts (`ICatalogSystem` and `MetaDataPlus`) are still fast, but the implementation of versioning in Dynamic Data Store⁴ was the bottleneck. First, DDS was was not designed to handle a “store” with multiple millions of rows. Secondly, the queries are generated automatically and they are less than optimal to access data.

The idea for storage was pretty simple and perhaps was chosen because it looked quite straightforward. Each version (which was called `CatalogContentDraft`) is stored in one row, and except the “static” data which is supposed to be on all versions (such as content link, code, etc.), all properties (aka `IContent.Property`) are serialized and stored in a big column of `NVARCHAR(MAX)`. Compared to the way it’s stored in metadata classes, this was, of course, slower and contribute to the slowness of system as well.

And having data in two places means you have to sync it every time you save, which adds even more overheads to the system.

To improve the performance, those issues must be addressed: DDS must be ditched, serialization should be minimized and

⁴You can read more about Dynamic Data Store [here](#). If you want to take a look at how catalog versions were stored (Given you have Commerce 8 databases), check `tblBigTable` in CMS database. The catalog content drafts are in `CatalogContentDraftStore` store.

synchronization should be reduced. All were done in Commerce 9, by rewriting the versioning storage entirely. The versions of catalog contents are now stored in more or less the same way with “published” versions (from the previous section, you might already know about `CatalogContentProperty`), and inline with what CMS does. Version information are stored in `ecfVersion` table, while version properties are stored in `ecfVersionProperty` table. `ecfVersionProperty` has same “schema” as `CatalogContentProperty`.

Language versions.

If you have been working with CMS content - then language is an area that Catalog has significant difference compared with CMS content.

It’s fair to say that Episerver CMS is more “advanced” when it comes to language settings. CMS allows you to set a content exists in a certain language or not. However, Catalog content does not have such flexibility. There are some characteristics to keep in mind:

- All catalog content language versions are defined by the language setting in the catalog. A content (node or entry) will exist in all languages enabled in its parent catalog. However, it does not mean that when you enable a language, versions in that language will be created automatically. The missing language versions will be created on-the-fly on demand the next time you request it.



All node and entries in this catalog will be available in English and Swedish

Let's take an example, your catalog was English only, then because you start selling in Sweden market, you want to add Swedish version - that can be done by enable Swedish language in CMS, and then add it to your catalog. At this point your nodes and entries still only have versions in English. When you request the Swedish language version (explicitly or implicitly), it is created on the fly, saved to database and returned to you.

CMS content in general, has the concept of master language. For catalog content, the master language of products is defined by the default language of their catalog. When a property is decorated with `CultureSpecific` attribute (aka its corresponding metafield has `MultiLanguageValue = false`), it is supposed to save in master language, and is shared between other languages. Those properties are not editable when you edit non-master language version:

The screenshot shows a CMS property editor interface. The left column contains properties: 'Display name' (with a text input field), 'Name' (with a value 'Tops-Tunics-CowlNeck-Bi.' and a 'Change' link), 'Name in URL' (with a 'Change' link), 'SEO URL' (with a 'Change' link), and 'Code' (with a value 'Tops-Tunics-CowlNeck-Black..'). The right column contains: 'Markets' (with a value 'All' and a 'Change' link), 'Visible to' (with a value 'Everyone'), 'Languages' (with a value 'en,sv'), 'ID, Type' (with a value '83, Fashion Item Content'), and 'Product' (with a value 'Tops-Tunics-CowlNeck' and a 'Tools' dropdown menu). The 'Name', 'Name in URL', 'SEO URL', and 'Code' fields are highlighted in grey, indicating they are non-CultureSpecific and not editable in the current non-master language version.

Non CultureSpecific properties are grey out

By default, these properties are non-CultureSpecific: - Code - Name - Markets - Assets - Prices, Inventories are also not editable in non-master languages

That goes the same when you update the content with API:s. This code will not throw any error, but it does not really save anything to database:


```

1  var contentLink = ContentReference.Parse("83__CatalogCont\
2  ent");
3  var content = contentRepository.Get<FashionItemContent>(c\
4  ontentLink, CultureInfo.GetCultureInfo("sv"))
5      .CreateWritableClone<FashionItemContent>();
6  //Assuming Facet_Size is non CultureSpecific.
7  content.Facet_Size = content.Facet_Size + " edited";
8  contentRepository.Save(content, DataAccess.SaveAction.Pub\
9  lish, EPiServer.Security.AccessLevel.Publish);

```



One important behavior is when you try to load versions of catalog content (via `IVersionContentRepository.List`), Episerver Commerce ensures that every language has at least one version. You might argue that this is an unwanted side effect - and I can agree with you on that. It can also happen implicitly, as few APIs might call to that method without your intention. It has some quite significant performance impact. Unfortunately due to the way content version API is structured, it's very hard to change/fix.

When a content is loaded, how are its properties treated? If the language is different from master language: `CultureSpecific` properties are loaded from that language, but non-`CultureSpecific` properties are loaded from master language. That's why Catalog content always requires master language version to be published before publishing any other versions. This also comes with a caveat: You should not change the default language of a catalog, otherwise all non-`CultureSpecific` property will be lost (until you want to get your hands dirty and update directly in database, which is generally advised against). This is, however, an extreme case and I don't expect you to do such action.

The same rule of "WorkId triumphs everything else" also applies to language. That means if you want to load a content with a

WorkId points to a language which is different to the language you want to load, then the language pointed by WorkId will be loaded. IContentRepository and IContentVersionRepository both provide methods for you to get a specific version of a content.



One important rule of content languages to remember: never change ExistingLanguages until you absolutely know what you are trying to do. Optimizely Commerce use ExistingLanguages internally to save versions in different languages, and if you change that it can lead to some unpredicted behaviors. It should have been a read only property, but as it's an implementation of ILocalizable, it is required to be write-able.

Version settings

There are two important settings for version in Commerce:

The first one is DisableVersionSync. This can be set by a key in appSettings section. If this setting is true, when you update catalog content from ICatalogSystem directly (which means, in most of the cases, catalog import export), the update will also delete all other versions, only latest, published version is kept. This setting comes in handy when you are using a PIM system to manage your catalog content, and it has some performance benefit.

The second one is UIMaxVersions. However, this setting affects both catalog content and CMS content, so think carefully before using it. This setting allows you to set the maximum number of versions per content you want to keep, if the number is reached, the oldest version would be trimmed when you save a content. You can either set this by code:

```
1 EpiServer.Configuration.Settings.Instance.UIMaxVersions =\  
2 1;
```

Or by adding `uiMaxVersions` (not `UIMaxVersions`, the attributes in `siteSettings` are case-sensitive) attribute to `siteSettings` in `episerver.config`.

If you don't set the value, by default both CMS and Commerce will keep 20 most recent versions of each catalog content. It's a best practice to set this to a specific value of your choice to keep your version history manageable (and in long term, maintain performance, too many versions might be bad for performance).

Another option to clear old versions is to use the extended flag for `SaveAction - ClearVersions`. Instead of just using `Save.Publish` when you publish content, make sure to add the extended flag:

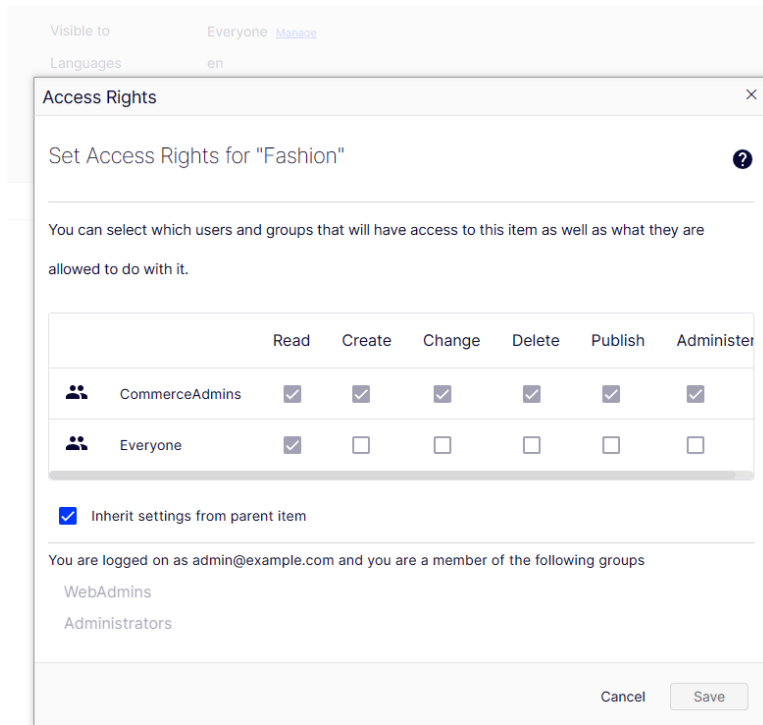
```
1 var extendedAction = SaveAction.Publish.SetExtendedAction\  
2 Flag(ExtendedSaveAction.ClearVersions);
```

Compared to the two previous options, this option is more flexible. You can apply it on certain catalog content which match specific criteria.

Access right settings

Catalog content has basic access right settings. Compared to CMS which can set access rights to each individual content, Commerce only allows you to set access rights on Catalog and categories. Entries will inherit access rights from their “true” parents. As we learned before, that means the category which they have the `NodeEntryRelation` with `IsPrimary` set to true. If an entry has no parent node, it will inherit access right settings from the catalog.

You can either set the access rights using Catalog UI



How to set access rights for catalog content

or using the APIs, specifically `IContentSecurityRepository`:

```

1 var descriptor = _contentSecurityRepository.Get(catalogLink\
2 nk);
3 //make changes to the settings.
4 _contentSecurityRepository.Save(catalogLink, descriptor, \
5 SecuritySaveType.Replace);

```

Import and export catalogs

It's fairly common that sites use an external system to manage catalog information. One of the examples is to use Product Information

Management - PIM - system, then export the changes and import to Optimizely B2C Commerce site. You might have heard about PIM, like inRiver, and their connectors to export the catalog to a format that Episerver Commerce understands.

CatalogImportExport

This is the most common way to import/export catalog.

When using `CatalogImportExport` - exporting is always full export - so everything in the catalog will be exported. However, import can be partly, you might have catalog which only contains part of the catalog. That make senses because you don't always update all catalog information, and importing only the changes will help you save both time and resource.

The heart of import/export catalog is `CatalogImportExport`, but you will mostly work with its wrapper, `ImportJob`. The reason we need `ImportJob` was because importing can be a long task and `ImportJob` has an async implementation with feedback - i.e. the caller can know what is the current progress - how many entries were imported etc. Unlike many other classes where you create a new instance by the inversion of control framework, like `structuremap`, you create a new instance of `ImportJob` by the constructors, using one of these constructors:

```
1 var importJob = new ImportJob(sourceZipFile, overwriteDup\
2 licates);
```

or

```
1 var importJob = new ImportJob(sourceZipFile, sourceXmlIn\
2 ip, overwriteDuplicates);
```

or

```
1 var importJob = new ImportJob(sourceZipFile, sourceXmlInZ\  
2 ip, overwriteDuplicates, isModelsAvailable);
```



In retrospect, those parameters should be in the
Execute method below.

Here you are passing the path to the catalog zip file, the name of the catalog file inside that zip (default is “catalog.xml”), and if you want to overwrite catalog items if existing ones found, and if the strongly typed content models are available (which will affect if the asset links will be imported or not). The next step is just to run it:

```
1 importJob.Execute(addMessageAction, cancellationTokens);
```

`addMessageAction` is an `Action<IBackgroundTaskMessage>` so you can have a callback to your method to decide what to do with the messages the importer sends to you (either display them to the UI, log them, or just ignore), and `cancellationTokens` is a `CancellationToken` so you can cancel the job (only has effect if the import job is extracting the zip file).

To export catalog, it's even simpler:

```
1 var importExport = new CatalogImportExport();  
2 using(FileStream fs = new FileStream(filePath, FileMode.C\  
3 reate, FileAccess.ReadWrite))  
4 {  
5     importExport.Export(CatalogName, fs, folderPath);  
6 }
```

This will export your selected catalog (via `CatalogName`) to the `filePath` in `folderPath`. Normally, you would want `filePath` to have file name of `Catalog.xml`. You can then zip this file to make it easier to store, or to transfer, something like this

```

1 ZipFile.CreateFromDirectory(folderPath, String.Concat(fol\
2 derPath, ".zip"));
3 var stream = new FileStream(String.Concat(folderPath, ".z\
4 ip"), FileMode.Open);
5 return File(stream, "application/octet-stream", catalogOu\
6 tputName + ".zip");

```

CSV Import

CatalogImportExport is not the only way to import the changes from an external source - you can also use CSV for such task. However while a Catalog.xml is very self-contained and has everything you need for a catalog, CSV files are very simple. So you would need an extra part - a mapping file to let Commerce knows which column in CSV maps to which field in catalog.

For example here's a mapping rule:

```

1 <RuleItem><SourceColumnName>Code</SourceColumnName><SourceColumn\
2 nType>System.String</SourceColumnType>
3 <DestColumnName>Code</DestColumnName><DestColumnType>NVarChar\
4 Char</DestColumnType>
5 <FillType>CopyValue</FillType><CustomValue />
6 <DestColumnSystem>True</DestColumnSystem></RuleItem>

```

You can define and change these rules by Commerce Manager:



Update mapping rules

Creating mapping rules might not be the most interesting part of your job, but fortunately you only have to do it once - as long as your CSV format does not change.

```

1  var dataContext = CatalogContext.MetadataContext.Clone();
2  dataContext.UseCurrentThreadCulture = false;
3  dataContext.Language = "en";
4  var mappingMetaClass = new EntryMappingMetaClass(dataCont\
5  ext, metaClassName, 1);
6  Rule mappingRule = Rule.XmlDeserialize(dataContext, mappi\
7  ngFilePath);
8  char textQualifier = '\0';
9  if (mappingRule.Attribute["TextQualifier"].ToString() != \
10  "")
11  {
12      textQualifier = char.Parse(mappingRule.Attribute["Tex\
13  tQualifier"]);
14  }
15  IIncomingDataParser parser = null;
16  System.Data.DataSet rawData = null;
17  parser = new CsvIncomingDataParser(sourceFolder, true, ch\
18  ar.Parse(mappingRule.Attribute["Delimiter"].ToString()), \
19  textQualifier, true, Encoding.Default);
20  rawData = parser.Parse(Path.GetFileName(csvFilePath), nul\
21  l);
22  System.Data.DataTable dtSource = rawData.Tables[0];
23  string userName = Thread.CurrentPrincipal.Identity != nul\
24  l ? Thread.CurrentPrincipal.Identity.Name : String.Empty;
25  return mappingMetaClass.FillData(FillDataMode.All, dtSour\
26  ce, mappingRule, userName, DateTime.UtcNow);

```

CSV can only be used for import, there is no built in way to export data to a CSV. But that would be easy to do so, as CSV is an universal, well known format.

Batch API

As discussed in previous section, `CatalogImportExport` is the most common way to import catalog changes into Optimizely Commerce Cloud, mostly due to its simplicity and performance. If you have a PIM system with connector to export the changes as a catalog (and you don't care about versions), it's probably the most performant way you can get. But if you don't, and if you want to have more control of the "importing" process, writing code to publish catalog content using content API is the most common way. But then you quickly realize it's not the best way to do it, the process is usually slow. Updating a few hundred products a day is usually OK, but if you need to make changes to few thousands ones every house, the system can't seem to keep up with it. In most cases you will get about 1 product/second which is too slow.

Then the batch saving API comes to rescue. Introduced in Commerce 13.10 as an extension of `IContentRepository`, the batch APIs fill the gap of performance. It is a very simple one, you take a list of `CatalogContentBase`, and a flag that indicates if you want to sync the changes to the draft versions or not, and that it

```
1 public static void Publish(this IContentRepository conten\  
2 tRepository, IEnumerable<IContent> contents, PublishActio\  
3 n publishAction)
```

The performance gain varies case by case, but it's reasonable to expect a 5-10x gain.



If the list of contents contains a catalog, or if it is a new content, `Publish` will fallback to the normal saving individual path. Otherwise it will save multiple contents at one go. If you have a very long list however, it will use batches internally.

How could it be fast? To understand that, we need to understand why `IContentRepository.Save` is slow. If you profile your code with that API (which you really should, it's important to understand your code's characteristics), you will see that a lot of time is actually spending in validation. Combining with multiple database roundtrips, there is a performance wall that `IContentRepository.Save` can't break even if you optimize everything else.

`Publish` avoids those issues by doing ... none. It bypasses the publishing pipeline almost entirely.

While it's not the complete replacement of `Save`, it's the best thing you can try if you do not care about versions, and performance is your priority.

Performance considerations

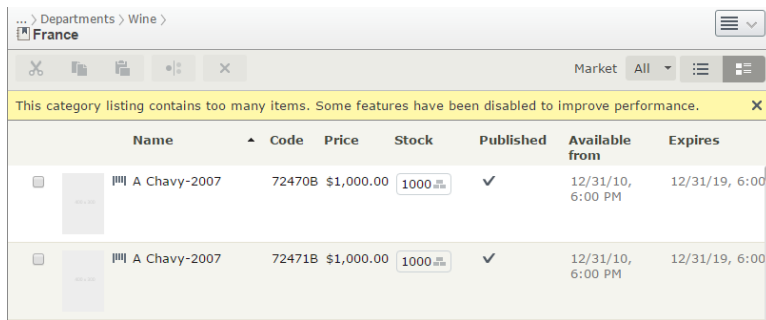
A frequently asked question I have received is that “Can Optimizely Commerce Cloud support our catalog”. While the number of entries (products/SKUs) in your catalog is definitely a major factor, it is so much more than that. Catalog size is more than just the number of entries, but also the number of languages, the number of versions, number of metafields, how you structure your catalog, and how do you CRUD them. It's pretty complicated topic to begin with.

Optimize your catalog structure

As the most frequently hit entities your system, Catalogs have a big impact of performance for your website - so put some thoughts into organizing it properly will definitely help.

There are a couple of things to consider:

- Do not put too many entries in one node. This has been greatly improved in Commerce 9, but still, Catalog UI will still struggle to manage that many items. It was simply not designed to handle a lot of entries in a same category. Performance, both backend and frontend will suffer.



The screenshot shows the Commerce 9 Catalog UI for the 'France' department. A yellow warning banner at the top states: 'This category listing contains too many items. Some features have been disabled to improve performance.' Below the banner is a table with columns: Name, Code, Price, Stock, Published, Available from, and Expires. The table contains two rows of wine products, both named 'A Chavy-2007'.

Name	Code	Price	Stock	Published	Available from	Expires
A Chavy-2007	72470B	\$1,000.00	1000	✓	12/31/10, 6:00 PM	12/31/19, 6:00 PM
A Chavy-2007	72471B	\$1,000.00	1000	✓	12/31/10, 6:00 PM	12/31/19, 6:00 PM

This warning will appear if your node has more than 2000 entries

By default, Catalog UI will display a node in a product - variations structure, variations will appear as children of products. However if the number of entries in a node reaches 2000, then it will only display flat structure. The threshold can be changed by `SimplifiedCatalogListingThreshold` value in `AppSettings`.

There is no definitive value for the optimal number of entries per node, it totally depends on your catalog, but speaking from experience, you might want to try to limit it to less than 2000. Anything bigger than that and you should be thinking about adding sub-categories.



In the past, it's not just CatalogUI. the hierarchical routing system will need to find all of children of a node to find the next matching `RouteSegment`. Having too many children, of course, can't be a good thing. However that is fixed in later version of Commerce, so the performance for routing will remain almost linear regardless of the number of children. That's why you should upgrade to latest version whenever you can, because the framework will be more tolerant with structuring "mistakes".



I have seen categories with more than 600.000 entries (yes, six hundred thousands). Needless to say, Catalog UI performance suffers and not only that, it bring the entire site down every time an editor open the UI. There have been multiple improvements made to make the situation better, but simply put, nothing beats a well, proper categorized catalog.

- Think about which properties to put in a product, and what to put in its variations. Sometimes it's crystal clear to do so, sometimes it's not, so take some considerations for this, with priority given to product. Having multiple variations to share same metafields will reduce the number of rows we have to load from `CatalogContentProperty` table. Less rows mean better performance. That might not sound a lot, but imagine you have 100.000 variations, in 10 languages. Each metafield less means 1.000.000 less in database. It can be some thing.
- Same goes to culture-specific and non culture-specific metafields. Culture-specific fields will be shared across all languages, so check if you need a field to be culture-specific. (Of course, if your site is going to have many languages. If it has only 2-3 languages then the difference is very small, thought.)

Optimize your catalog operations

In previous section we talked about how to structure your catalog for the best performance. But that's only the half of the picture - what even more important is how you work with your catalog.

As we learned before, by default, the catalog content is cached - which is a good thing, because the next time you need it, Episerver Framework will happily load it from memory, save you both precious time and disk I/O. However, if your catalog content is only loaded for one use, cache is not helping here. And remember your server's memory is limited - and one instance of catalog content is not that small, depends on your modeling, it can be easily 10MB or 20MB. No cache can live forever. It will, sooner or later, be removed for new catalog content. The more that happens, the worse it is for performance - even worse if Garbage Collector needs to kick in often. If you are continuously loading and discarding catalog contents from cache, then you are probably doing something wrong.

As a rule of thumb, only load the content when you absolutely need it. I've seen more than often when the developers feel generous to load more than they need - and in the end, the site struggles when it has to serve multiple concurrent users.

Here's something to keep in mind. If you can - do a review of your projects and fix the following problems. The performance gain might surprise you:

- You don't have to load an entire content just to get its code. That's what `ReferenceConverter` is for - a new method `GetCode(ContentReference)` was added recently and was even improved in Commerce 10+. It's much more faster and memory-efficient than a content. This is also true when you want to get, for example, `ContentReference` of all variations of a product. You don't have to get all the `VariationContent` - `IRelationRepository`) can provide the lightweight solution

for you. The general rule is - if you are not using the entire content, but just a small part of it, then look around. The framework might already provide something for you, with (much) better performance.

- Keep in mind that `UrlResolver.GetUrl` also load contents, and not just one - it loads content recursively until it finds the configured root (which we will discuss more in part 3 - Advanced Stuffs). Sometimes, call to `UrlResolver.GetUrl` is inevitable, but in many cases, you can cache it ...somewhere. If you are using `Episerver Find` (or even better, `Find.Commerce`), then the content url can be included in the index and reused for later calls.



In later version of CMS, `UrlResolved.GetUrl` has cache for itself, so it's significantly less expensive. However, the fastest code is the code that does not need to run, so if you have a way to avoid calling it, it's even better.

- If you are using `IContentLoader.GetChildren` for the product listing page, you are probably doing it wrong. Thing is, when a customer visits a product listing page, they unlikely click on every product - just one or two out of ten. Loading all of the children, even with paging, is not a good idea. A common best practice is to avoid loading the content until you needs to - and avoid loading multiple content in product listing page. That's what the search provider (or `Find/Find.Commerce`) is for. We will talk more about this is part 5 - Searching.
- Last but definitely not least, using batch APIs is arguably the best way to improve your catalog operation performance. If you are loading catalog content one by one inside a `foreach` loop, chances are you are doing it wrong. Ask yourself if the catalog content

could be loaded by `IContentRepository.GetItems` instead. You might be surprised with the performance improvement! Also, as discussed in previous section, using `IContentRepository.Publish` can dramatically improve performance.



This does not apply to catalog content only, but also for prices and inventories. We will discuss more in later chapters.