PRO EPISERVER COMMERCE

Creating powerful eCommerce solutions

QUAN MAI

Pro Episerver Commerce

Creating powerful eCommerce solutions.

Quan Mai

This book is for sale at http://leanpub.com/proepiservercommerce

This version was published on 2016-12-07

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.

© 2014 - 2016 Quan Mai

Tweet This Book!

Please help Quan Mai by spreading the word about this book on Twitter!

The suggested hashtag for this book is #episerver.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#episerver

Dedication to my wife, Huong Hoang, for her extraordinary help, understanding and support during the time I write this book.

Contents

Who is this book for **2**
Prerequisites 2
How to get started 2

Chapter 0: A brief history **3**

Chapter 0.5: A framework overview **8**
A framework built on pluggable dependencies. 8
Nuget packages and upgrading your site. 10
Terminologies and conventions 11

Part 1: Catalog System 12

Chapter 1: How the catalog is structured **13**
The entries 15
The two systems. 19
ReferenceConverter - the bridge. 22

Chapter 2: Catalog content modeling **25**
The MetaData system 25
The strongly typed models 28
Modeling catalog content types 31
Dictionary types. 38

Chapter 3: Associations, relations and assets **44**
Working with associations and relations 44
Assets 49
How versions work 56
Optimizing your catalog 61

Part 1.1 Important concepts. 63

Chapter 3.1 Markets **64**

Chapter 3.2 Application-level settings **69**

Part 2: Inventories and Prices 71

Chapter 4: Prices. **72**

Chapter 5: Inventories **87**
Warehouse 87
Inventory 88

Part 3: The order system - selling is what (really) matters 103

Chapter 6: The Order system. **104**
Overview 104
How stuffs are structured. 104

Chapter 7: New Order system - the new promise **133**
Why a new system 133
Everything is an interface. 133
IOrderRepository - Gooobye CartHelper 135
IOrderFactory - everything in memory 137
IOrderAddress 139
Workflows or non-workflows, that's the question. 140
The new cart system. 141

Chapter 8: Taxes **144**
Taxes configuration 144
Taxes calculation 146

Part 4: Events - Listen and react 147

Chapter 9: Content events. **148**

Chapter 10: Inventories, Prices and Orders events **155**
Inventories and prices events 155
Orders events 158

Part 5: Search 160

Chapter 11: Search **161**
Searching for products. 161

Searching for orders . . . 190

Part 6: Promotions - Everybody loves discounts. . . . 194

Chapter 12: The old promotion system. . . . 195
How promotion works? . . . 204

Chapter 13: The new promotion system . . . 206

Part 7: Business Foundation . . . 248

Chapter 14: Business Foundation overview . . . 249
Extending Business Objects . . . 252
Customer Contact . . . 255
Customer Address . . . 256

Part 8: Extending the system. . . . 257

Plug your implementations - Episerver Framework way. . . . 258

The unusual ways. . . . 262
Customize order processing . . . 262
Work with "workflows". . . . 263

The legacy continues . . . 270

Access rights and Permissions. . . . 275

Payment providers . . . 278
Configure payment methods. . . . 279
IPaymentGateway . . . 280
Manage payment methods programmatically. . . . 283
How payments work . . . 283

ProfileModule . . . 285

CustomerContact events . . . 287

The Migration Steps . . . 290
Migration settings: . . . 292

Part 9: Advanced stuffs **294**

Caching **295**

Routing **297**
The SEO Uri approach 297

Appendix **304**
How to ensure a successful upgrade. 304
Now what 305

The cover

The cover of this book was made with the work by David Iliff, as found here[1]

"Photo by DAVID ILIFF. License: CC-BY-SA 3.0"

Raven is hardly a beautiful bird, but it's damn beautiful picture and David was generous to give it away for free. My thanks to David. The image was cropped to fit the cover and the book title was added.

Why Raven?

At Episerver, the projects have code names as birds. Episerver Commerce was called Cuckoo for R3, while Commerce Sample site was called Seagull internally. Commerce 7.5 was developed with internal code name Condor (an internal joke inside the team was the release was big and ugly as the bird, but that's not necessary true). After 7.5, the "Vnext Commerce" has been named as Raven, but Episerver eventually switched to small releases model via Nuget packages, so no new code names (which are supposed for big releases). Commerce has kept the code name ever since, althought it's less and less used in discussion.

Disclaimers

While I am an employee of Episerver, and my bosses support me to write this book, Episerver neither requests nor pay me writing this. The view of this book is mine only, it does not represent the view of my employer about Episerver Commerce or Episerver products in general.

[1] https://en.wikipedia.org/wiki/Common_raven#/media/File:Corvus_corax_(Common_Raven),_Yosemite_NP,_CA,_US_-_Diliff.jpg

Who is this book for

If you are an exprienced .NET developers looking to start working with Episerver Commerce, or you are already working with Episerver Commerce but want to deepen your knowledge, this might be the book for you. I will try to explain the concepts, how things work in Episerver Commerce and how you can apply those information into your daily works.

I will not try to make this a cookbook (although I'll cover some detail implementations when I think they are needed). If you want to be up to speed, I would recommend taking one of Episerver Commerce courses (http://www.episerver.com/Training/) - there are some of very good trainers there. This book aims to be an view from inside Commerce, how things work, how can it be more efficient,etc. If you like in-depth books with internal information, this is for you.

This book targets Episerver Commerce 9. However, to explain concepts and decisions, I might talk about older versions in some places.

Prerequisites

You'll need Visual Studio 2013 or later (2015 prefered) and SQL Server 2012 or later to get the sample solution up and running. Some experience with C#/SQL Server are needed. Knowledge of Git and Github is a must (Come on, it's 2016, I should not have to say this). Previous experience with Episerver CMS or Commerce is a big plus, but not required.

How to get started

Episerver supports you to create a website from scratch with a Visual Studio extension[2]. This actually create a CMS site, and you can simply install Episerver.Commerce nuget package to it to make it a Episerver Commerce site. More detailed steps can be found here[3].

However, I would recommmend you to start with Episerver Quicksilver site[4]. Clone the repository, build it and run the set up command, enjoy a coffee and wait for the setup to complete in a couple of minutes, and you're good to go. Simple as that.

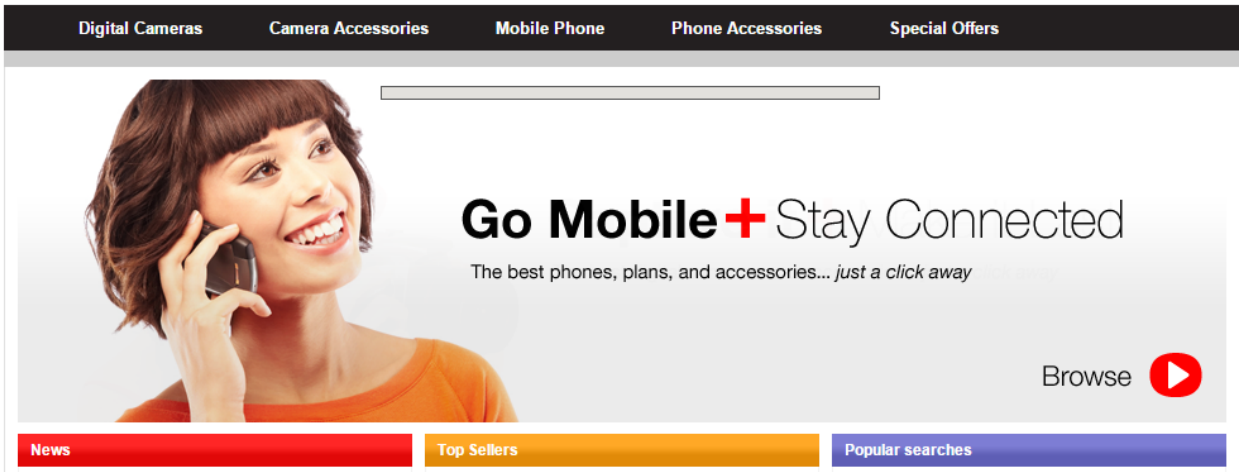
[2] https://visualstudiogallery.msdn.microsoft.com/4ad95160-e72f-4355-b53e-0994d2958d3e

[3] http://world.episerver.com/documentation/commerce/get-started-with-commerce/2--creating-your-project/

[4] https://github.com/episerver/Quicksilver

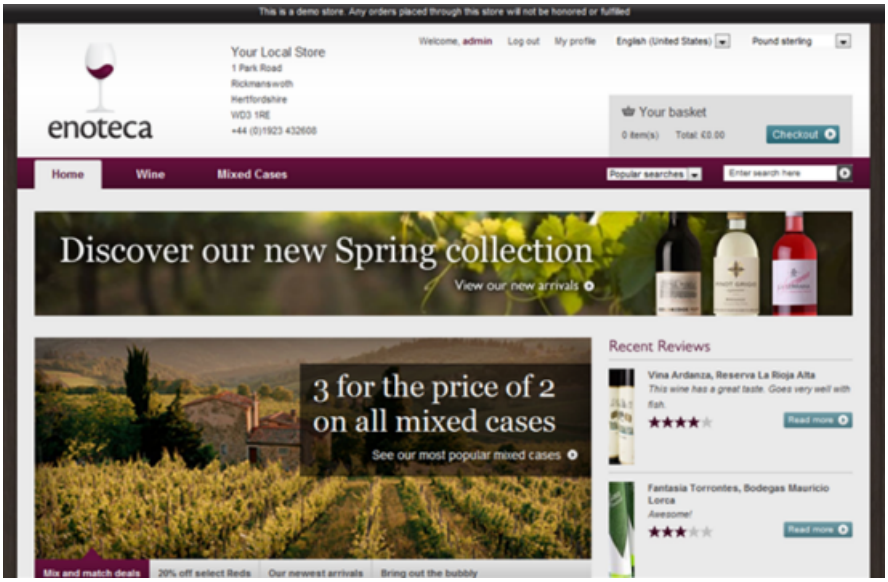
Chapter 0: A brief history

Episerver was formerly known as EPiServer AB, but follow the merge with Ektron in early 2015, it was rebanded to Episerver (without the capitalized P and S) recently. Throughout this book, we will always call the company as Episerver for consistency, even for the dates before the rebanding. However, it's still EPiServer in namespaces and package names.)



Click+Talk, demo site in Commerce R1 which built based on Mediachase template

Episerver Commerce, of course, is one product in entire Episerver platform, which consists of Episerver CMS (Core and UI) and Episerver Find. The history of Episerver Commerce dates back to 2010, when it was still a white brand product. Episerver already had a full-fledged, powerful CMS platform at that time, but there was no e-Commerce solution to provide to customers, so they decided to buy. Mediachase eCommerce Framework was bought and then integrated to Episerver CMS in white brand manner. Essentially, there were still two products: the front-end (Episerver CMS) and the back-end (Commerce Manager). The integration part was mainly CatalogPageProvider, a "PageProvider" which allows the front-end pages access to the catalog system. All changes to functionalities as well bug fixes had to go through Mediachase development. Severals releases were released, mostly to keep it compatible with CMS releases and fix bugs, from R1 to R1 SP2 in April 2012. The product was distributed as a plugin of Episerver Deployment center, and also came with Click+Talk, a sample WebForm site, based on the template site from Mediachase. Episerver then developed their own sample, Enoteca, still a WebForm site but with AJAX functionalities, for demonstration.



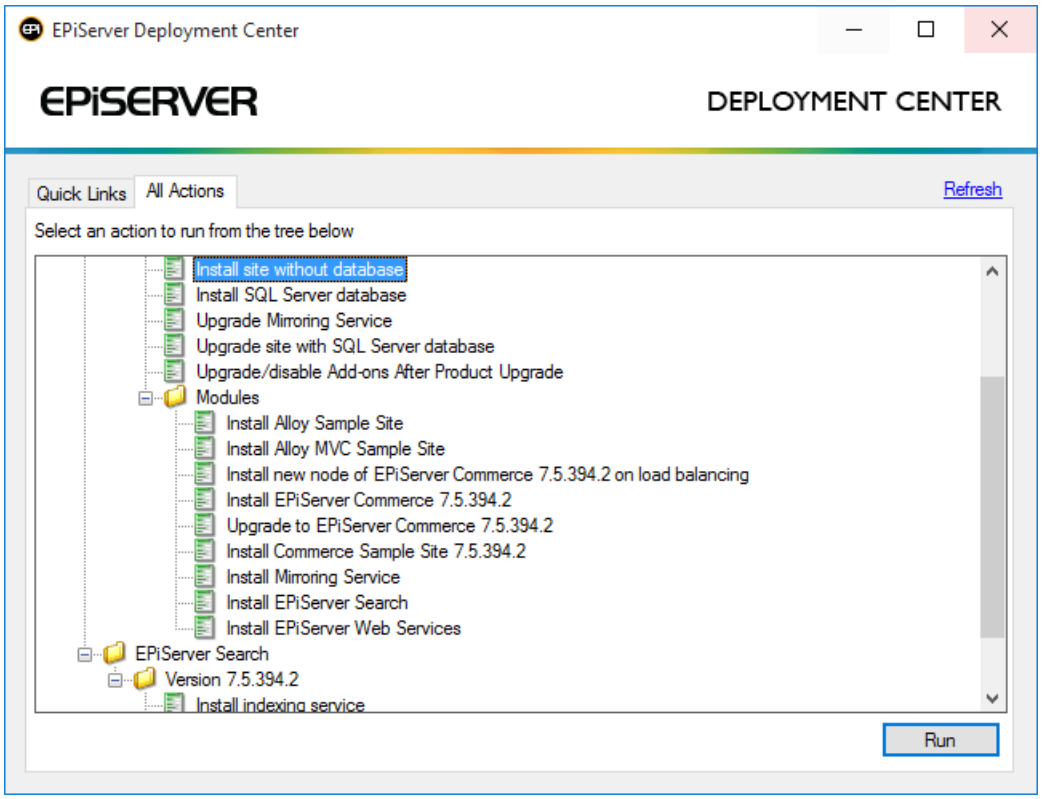
Enoteca, it's also my first project at Episerver

In the shift of strategy, Episerver saw the need of a full-fledged Commerce system. In several options available, they decided to buy Mediachase as a whole to get both their source code, employees and customers. The acquisition was made in April 2012, and with the source code of Mediachase eCF, the development of Episerver Commerce became full speed. The first version after the acquisition was not a big release, Commerce R3a, released in November 2012 as a compatibility release for Episerver CMS 7. The most notably new feature is the new pricing system. Previously, prices were part of SKUs, but now they can managed separately. There were other new concepts, but the time was not enough, so they focused on stabilization on the system instead (in other words, fixing bugs). Commerce R3a came with Enoteca as default sample site.

The true new major version of Episerver Commerce was only released one year later, in the bundle of Episerver 7.5. The biggest feature of new version was the CatalogContentProvider (along with the death of CatalogPageProvider, as the PageProvider was removed in Episerver CMS 7.5). From now on, the catalog system can be handled in the same way with CMS content via IContentRepository. New Catalog UI, a streamlined editing UI for catalog content was also introduced, and Episerver Commerce can be officially considered as a first class product in Episerver platform. The features which were targeted for R3a were then finished and released, too, most notably Market concept. Commerce 7.5 also replaced Enoteca with a new Sample site. While Enoteca was intended to be a fully featured site, it was more suitable for demonstration than to be a starter template. Developers, especially new Commerce developers, found it hard to customize the site to match their needs. New Sample site was intended to be a simple Starter package where developers can quickly get to speed and use it as a base to develop their site.

After 7.5, Episerver changed their release model. Instead of a major release every year, they switched to continous releases cycle. New versions, are released in every couple of weeks (usually 2 weeks) via Nuget packages. The shorter release cycle means new features and bug fixes find their way to partner developers quicker, and Episerver can get more responsive feedback, allow them to adapt

more quickly to customers' needs. Along with new Nuget packages, Episerver Deployment center, which has been there since forever, was ditched.

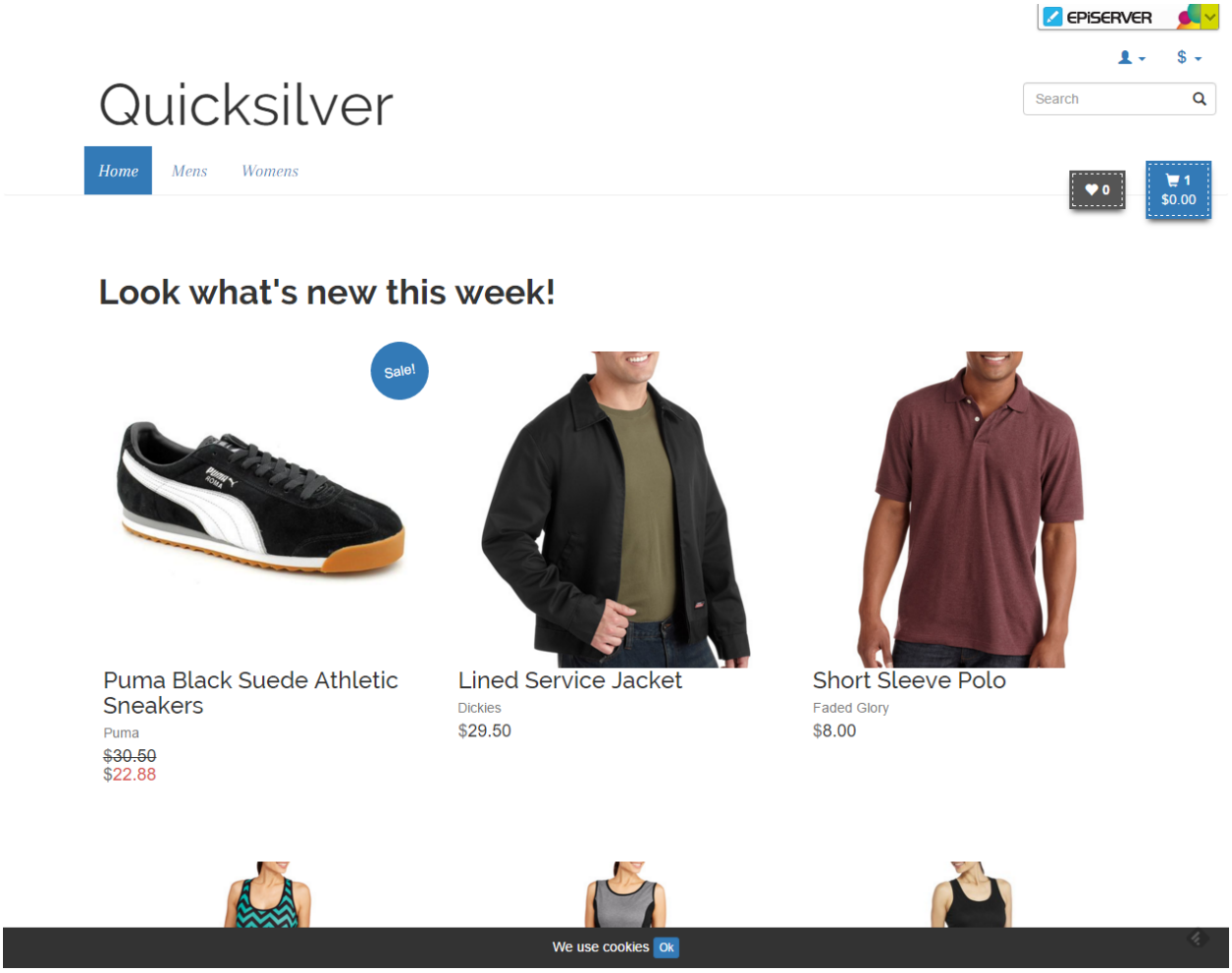


Deployment Center is dead, long live Deployment Center

Episerver, in general, goes with semantic version. Basically, a major version comes with breaking changes in API:s, a minor version comes with new features while a revision change means bug fixes. While there are no fixed schedules for releases, Episerver Commerce has been on track with one major release every year. In October 2014, Episerver Commerce 8.0 was released, with the most significant changes are to make it Cloud compatible. Episerver CMS has been Cloud compatible since 7.5 and Commerce needed to be able to deploy to cloud, Microsoft Azure and Amazon Elastic Compute Cloud, as well. The dependencies on Common Framework, mostly to support community features such as product rating and comments, were removed. Fulltext Search, which was not supported by SQL Azure Database, was removed. Some others notable changes between 7.5 and 8.0 were enhancements to Catalog UI: Creating and editing catalogs, editing dictionary metafields in Catalog UI is now possible, better permanent link handling and an interesting internal system, Migration Steps (will be discussed later).

If 7.5 was the first version that Commerce can be considered to be a first-class product in Episerver framework, 8.0 strengthened that statement. More and more edges were refined. However, there were still many not-so-great parts of the system. Performance in certain areas was still questionable. Order system was still not great to work with, and promotion system was still a pain point. And partners asked for a demo MVC site, instead of Commerce sample, as almost every new site is built

on MVC, not WebForms anymore.



QuickSilver - Episerver new MVC demo site, and was released as an open source project on GitHub. It has been very well-received

Episerver continued their efforts to make further improvements to Commerce. After numerous small releases in 8.x versions (Which 8.16.1 was the last version to support CMS 8), Commerce 9 was released in October, 2015.

The biggest changes in Commerce 9 were the re-architect of Catalog system storage. Back to the days of Commerce R3, Catalog content did not support version the way CMS content did, and it was added to Commerce 7.5 with Dynamic Data Store - DDS as a backing store. DDS has its advantages of being simple for uses, and readily available out-of-the-box, but it is also not so great in terms of performance. Unsurprisingly, it became the bottle neck of the catalog system when the catalog is big enough. Having the data in two places also mean that you need to sync data, adding another overhead. Inn Commerce 9, the entire catalog system backend was re-architected, which versions are now handled entirely in eCF database. The performance improvements were huge, there were a fold of 2-10x performance gain depends on the area. Moreover, the limitations in language versioning

are now lifted. The catalog versioning system is finally what it should be, performance and fully featured.

Commerce 9 is not the last product. The improvements are continously made. The focus after Commerce 9, is to complete the new promotion system, which has been in BETA mode for quite some times. As mentioned above, the semantic version Episerver follows does not allow to introduce new breaking changes without a major version. However, to make the best-of-bread API:s for the new promotion system, which has been rewritten entirely from scratch, it takes try-and-error steps. It's hard to make it right the first time. So the new API:s are marked with a BETA tag (it's just a comment to the class), saying it might not stable enough to use in production. Then it can be changed when it needs to. (Note that, the API:s itself might be unstable, it does not affect the stablization of the system at all. If you don't use the new API:s, it will not affect you one bit). The Commerce team is working hard to get their API:s stabilized and the BETA tags removed, some times later this year - 2016

Guess what will come in next version of Episerver Commerce?

Chapter 0.5: A framework overview

Episerver Commerce is a framework built on other framworks of Episerver: It has dependencies in Episerver Framework, Episerver CMS (both Core and UI). It has its roots in Mediachase eCF, but more and more stuffs have been moving to, or built on Episerver frameworks. I find it's hard to explain how Episerver Commerce works without talking about the underlying framework and API:s, so let's work through the most important concepts in Episerver frameworks. If you have been working with Episerver CMS and have knowledge in this area, you can skip this chapter.

A framework built on pluggable dependencies.

The ability of replacing the default implementations comes from two important features in the system: dependencies injection and initialization modules [5].

Up until very recent, Episerver uses structuremap[6] to handle the dependencies inside the system (Episerver.Framework has been updated to allow you to use your DI framework of choice, and `structuremap` is only the default option, since version 9.5). They built a wrapper (`ServiceLocator`) and so around `structuremap`, so you can easily register and resolve the dependencies.

Initalization module is another concept from Episerver Framework. Basically, it's a special interface which allows Framework to do stuffs during startup. If you looks into web.config/system.webServer/modules, you will see this:

```
      <add name="InitializationModule" type="EPiServer.Framework.Initialization.\
InitializationModule, EPiServer.Framework" preCondition="managedHandler" />
```

This little module is registered to run at startup, and it will do all "initialization" works from there - one of the stuffs is to get all classes implement IInitializationModule and run them in a correct order. [7]

One of the most important features of Episerver frameworks in general, is the ability to override the default implementation - it allows great flexibility when it you need some extra processing to fit your business needs. Let's say you need to pull out prices from an external source, for example, ERP, instead of using the system provided out-of-the-box. What you can do is to create a class implements `IPriceService`:

[5] Initialization modules concept is not as critical for the plug-able architect as the dependencies injection. You can use some other methods when the initialization modules are not available, such as static method of static class, but it will make the plugging easier and more manageable.

[6] https://github.com/structuremap/structuremap

[7] You can easily see the caveat here: the initialization modules only work with a web application. If you are working with non-web applications, such as console apps or Windows services, static method in a static class is your only option.

```
public class MyCustomPriceService : IPriceService
{
    //Your implementation goes here.
}
```

Now you need to register your class to be the implementation if `IPriceService`. There are two ways to do that, one by attribute:

```
[ServiceConfiguration(typeof(IPriceService))]
public class MyCustomPriceService : IPriceService
```

And another one is to register in a `IConfigurableModule`:

```
[InitializableModule]
[ModuleDependency(typeof(EPiServer.Web.InitializationModule))]
public class CustomInitialization : IConfigurableModule
{
    public void Initialize(InitializationEngine context)
    {
    }

    public void Uninitialize(InitializationEngine context)
    {
    }

    public void ConfigureContainer(ServiceConfigurationContext context)
    {
        services.AddTransient<IPriceService>(locator => new MyCustomPriceService\
());
    }
}
```

After that, you will be able to get a new instance of your custom class by calling `ServiceLocator.Currrent.GetInstance<IPriceService>`. This is how the pluggable dependencies work: When you ask `ServiceLocator` to resolve a class for you, it will try to create an instance of the class by calling the default constructor, with the parameters resolved from the dependencies. Let's assume we have `MyCheckoutService` which takes `IPriceService` as one of the constructor's parameters, and `MyCustomPriceService` takes `IErpService` as a constructor's parameters, then when you tell `ServiceLocator` to create an instance of `MyCheckoutService`, it will first try to resolve an instance of `IPriceService`, which in turn, it needs to create an instance of `IErpService`. After having an instance of `IErpService`, it will now able to create an instance of `MyCustomerPriceService` and in

the end it's able to create an instance of `MyCheckoutService`. You don't have to bother to create all the instances yourself, you just need to define the dependencies chain.

The former is more convenient, and is used commonly. However, it has a major drawback: You can't control with class to be registered last - one of the most important characteristics of the dependency injection is "The last (registered) one wins". In the case of overwriting the framework default implementation, you will need to make sure registration of your class comes *after* the default one, so it'll be used. You can control that order by specifying the module in `ModuleDependency`. In the example above, `CustomInitialization` will always run after `EPiServer.Web.InitializationModule`. If we want to make sure `MyCustomPriceService` is registered over the default implementation, you can change `ModuleDependency` to `CommerceInitialization` - where the default one is registered.

We will go more in depth about this pluggable dependencies in part 3 - Advanced stuffs.

Nuget packages and upgrading your site.

After 7.5, Episerver switched to continuous releases process, which allows new features and bug fixes to come out every 2 weeks or so via Nuget packages, instead of 18-months major release as before. Episerver nuget packages are available here[8]

There are plenty Nuget packages for Commerce, but most of them were created for the modularization of the system. And thanks to Nuget dependencies model, when you upgrade a package, it will figure out which packages need to upgrade as well, so there are some packages you really need to care about:

- EPiServer.Commerce: This is the main package to upgrade your front-end site. Normally you will update this package to update your site.
- EPiServer.Commerce.Core: This is the package contain core functionalities: Catalog system, Promotion system, etc. You usually do not have to upgrade it directly, unless you have a site without Editing UI (in a load-balancing setup for example), or a (Windows) service.
- EPiServer.Commerce.UI: This package contains the UI functionalities, including the gadgets, Catalog UI, Marketing UI and so on.
- EPiServer.CommerceManager: This package contains updates for Commerce Manager, you should only update it in your Commerce Manager site.
- EPiServer.Find.Commerce: This package adds Find-support functionalities to Commerce, such as indexing the catalog content when the prices/inventories change, or filter by prices, etc. This is optional, and should not be installed if you don't use Episerver Find.
- EPiServer.ServiceApi.Commerce: This package add REST-based API:s to your site to manipulate the catalog (import, export, update nodes and entries). This can be installed to your site, or to a separated site, to make it a REST-server.

[8] http://nuget.episerver.com/

Episerver follows sematic versioning rules. Which means when you look at a package version, you can know what it contains: - A revision change means this package contains only bug fixes. New API:s might be added, but no changes to current API:s. Strictly speaking, one or more API:s can be obsoleted, but you can still use them without changing your code. - A minor version change means this package contains new features, and possibly bug fixes. Same as above, no changes to current API:s. - A major version change mean it includes pending changes. Obsoleted API:s can now be removed. In most of cases, you will have to update your code to make them built.

In the past, a major version from Episerver means a big release - as it contains both breaking changes and new big features (One exception is Episerver 7.5, which is a major release for Commerce, but only the minor version changed), and critical bugs will be distributed via patches. However, everything changed when Episerver moved to continuos releases after 7.5 - for now, a major version only means it contains breaking changes, without any new features. A new feature, even bug, can be released in a minor version. This is indeed a very good thing as instead of waiting months, or even a year to get your annyoing bugs fixes, you can get them in weeks. And you can use new features right after they're done, not until a major version which only be released once every year.

Terminologies and conventions

It should be easier if we make sure we are talking about the same thing. In this book, we will use the terminologies as following:

- CMS-site or front-end: the CMS-based site which comes with new stuffs, contents and integration.
- View-mode: user-facing interface, e.g. what end-users see, without any editing features.
- Commerce Manager or backend-site: the "old" site with legacy stuffs.
- Catalog UI: the new UI for catalog content management.
- Marketing UI: the new UI for marketing management.

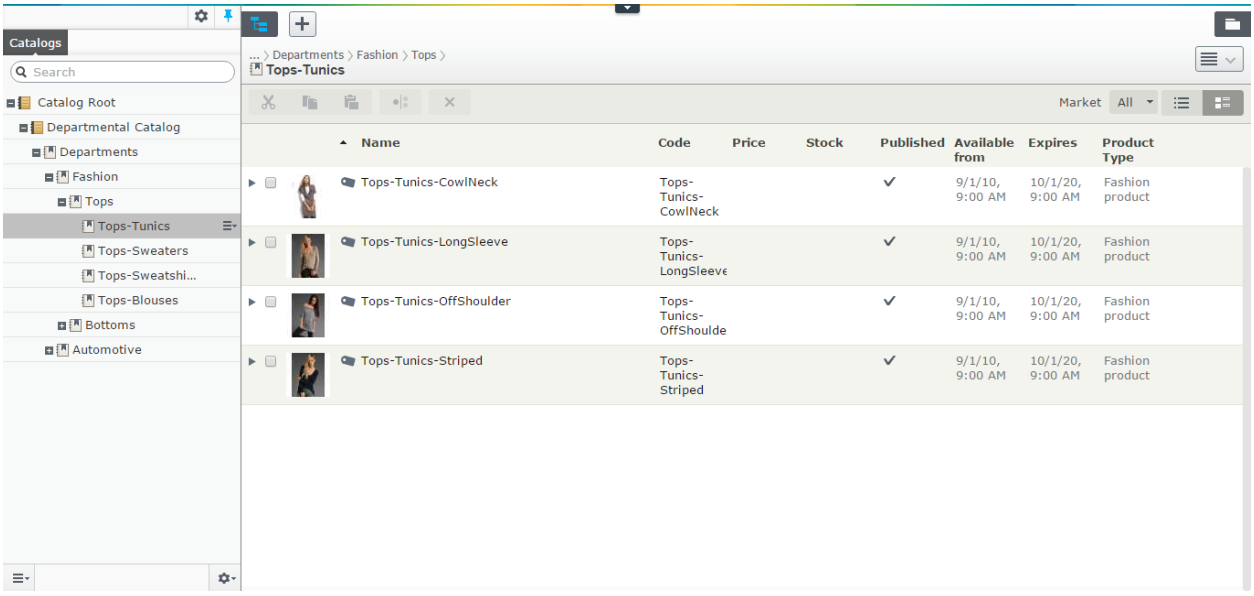
Part 1: Catalog System

The catalog system, in many means, can be called the heart of Episerver Commerce. 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

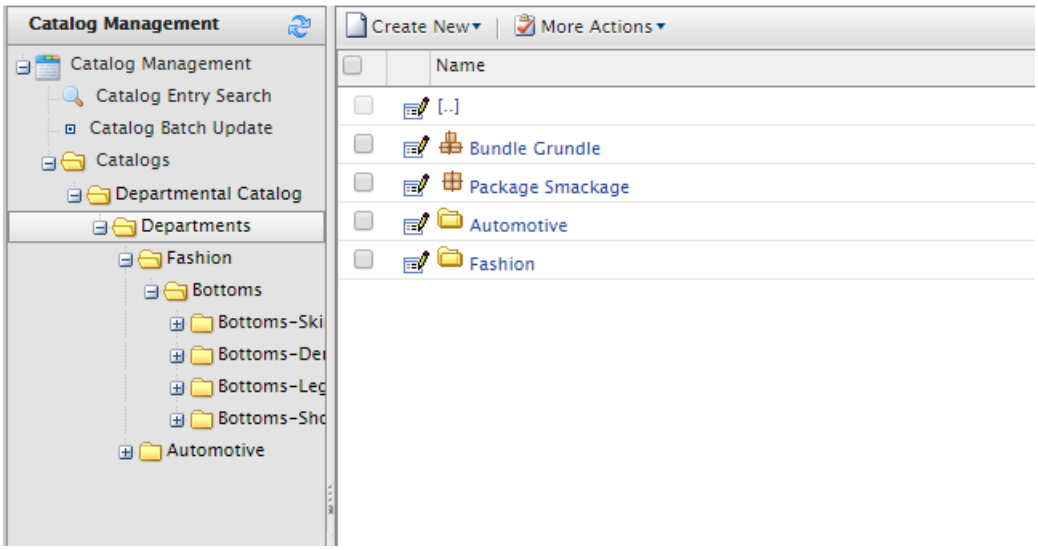
##Overview

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 almost as full featured as Catalog Management, except for some small features, and can be used to edit the Catalog entirely. However we'll show screenshots in both interfaces to see how they're connected.



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 it's not enough. Mind you, the catalog is much more complicated than that, which all kinds of relations and associations we'll discuss shortly.



Catalog Management in Commerce Manager

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.[9][10]

The data of Catalogs are stored in `Catalog` and `CatalogLanguage` tables. If there is anything interesting to look at, it's CatalogLanguage. The languages available here will define which languages you content will have in. 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 can have nodes[11]. 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.

[9]When 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, which is mentioned later, when you export inside Commerce Manager, those information will be lost. We will have an exercise of writing a small tool to export the catalog with the asset information later.

[10]You 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.

[11]You can, of course, 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.

- 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.[12]

And then we have relations and associations between entries.

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 no 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	✓	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	✓	9/1/10, 3:00 PM	10/2/19, 3:00 AM	Fashion variant/SKI

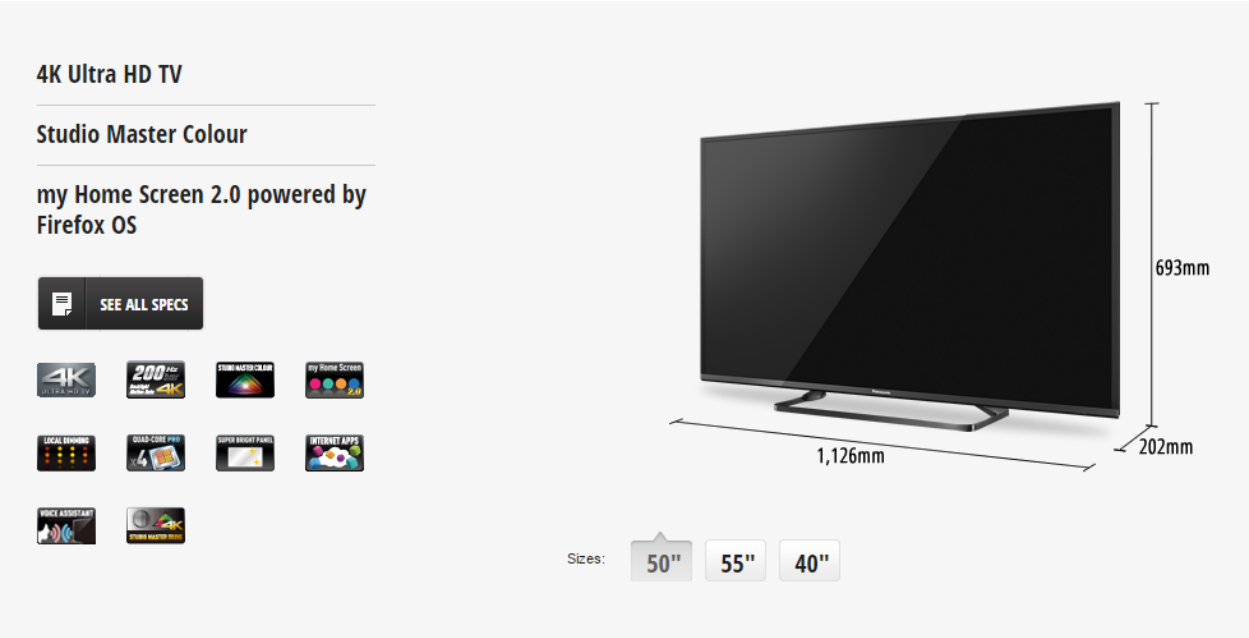
4 types of entries in the Catalog

- 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. I would suggest to stick with the common way.

[12] Unlike the Node relation, there is no way to know which is the "true" parent node of an entry. Currently, the first relation with lowest `SortOrder` is assumed to be the parent node. This is not entirely correct. Future version of Episerver Commerce might correct this and introduce a `ParentNodeId` for each entry

- Variation/SKU: It's easiest to explain Product-Variation in term of a TV series. Let's pick the Panasonic CX680 4K 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 3 variations of its by size. Here's a screenshot from Panasonic website:



A product with its variations

A variation might have its own inventories and prices. In the end, it's the thing your customers actually buy. However, it's 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. [13]
- 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.

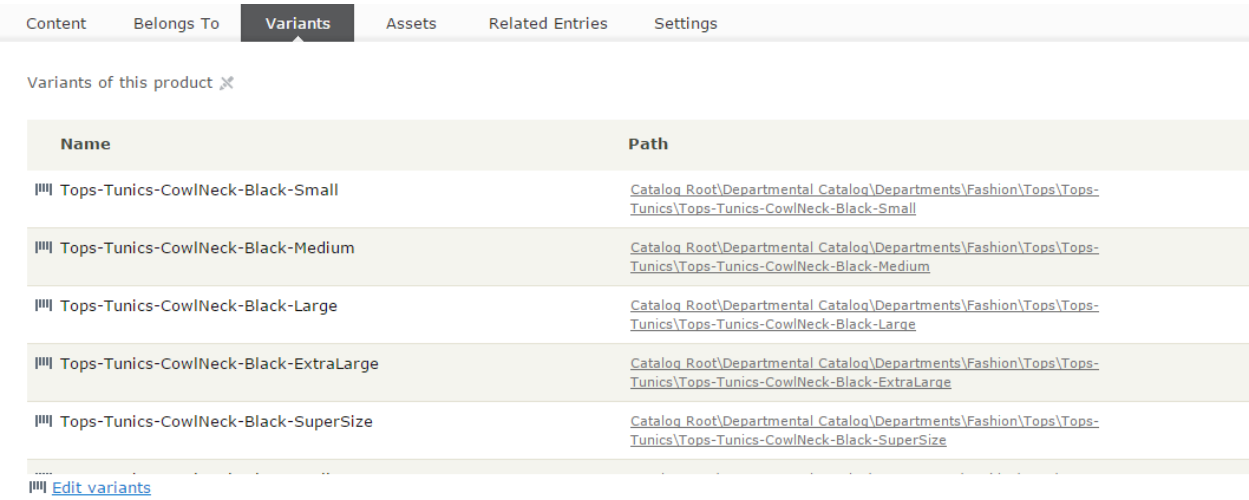
And then comes the tangible relations and associations between them.

There are three type of relations between entries:

[13] In earlier versions of Commerce, such as R3, 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.

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

Those relations are stored in `CatalogEntryRelation` table, and are managed by the the Variants tab (for Products) or Package Entries (for Packages) or Bundle Entries (for Bundle)

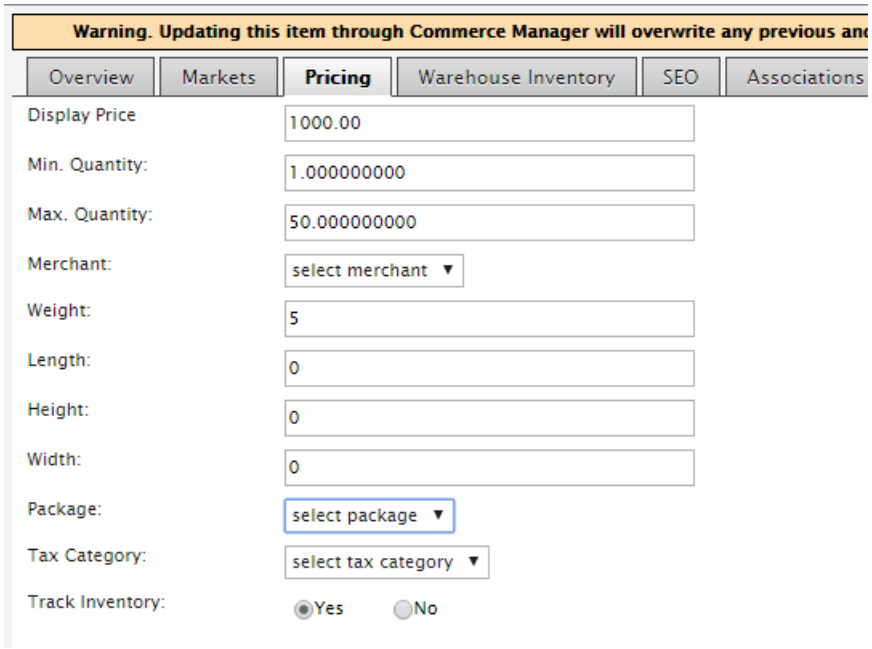


If you look at the tab Variations/SKUs when editing a Product in Commerce Manager, you'll see "Quantity" column. This is not entirely correct as the Quantity is never used in a `ProductVariation` relation. Catalog UI reflects this better. It only matters in `BundleEntry` and `PackageEntry` relations.

And then between entries, you can set the associations. It can be the "CrossSell" or "UpSell" items that you might want to show in "You might also want" section when a customer browses a product detail page. The associations are stored in `CatalogEntryAssociation` and `CatalogAssociation` table.

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:



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 Bluray 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 note: this is only the value without unit. It's supposed that the unit is the base weight setting of the Catalog. (which, by default, can be either kilograms or pounds). Episerver 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. Prior to 8.7, it comes with dimension (Height, Width and Length) settings and is the only way

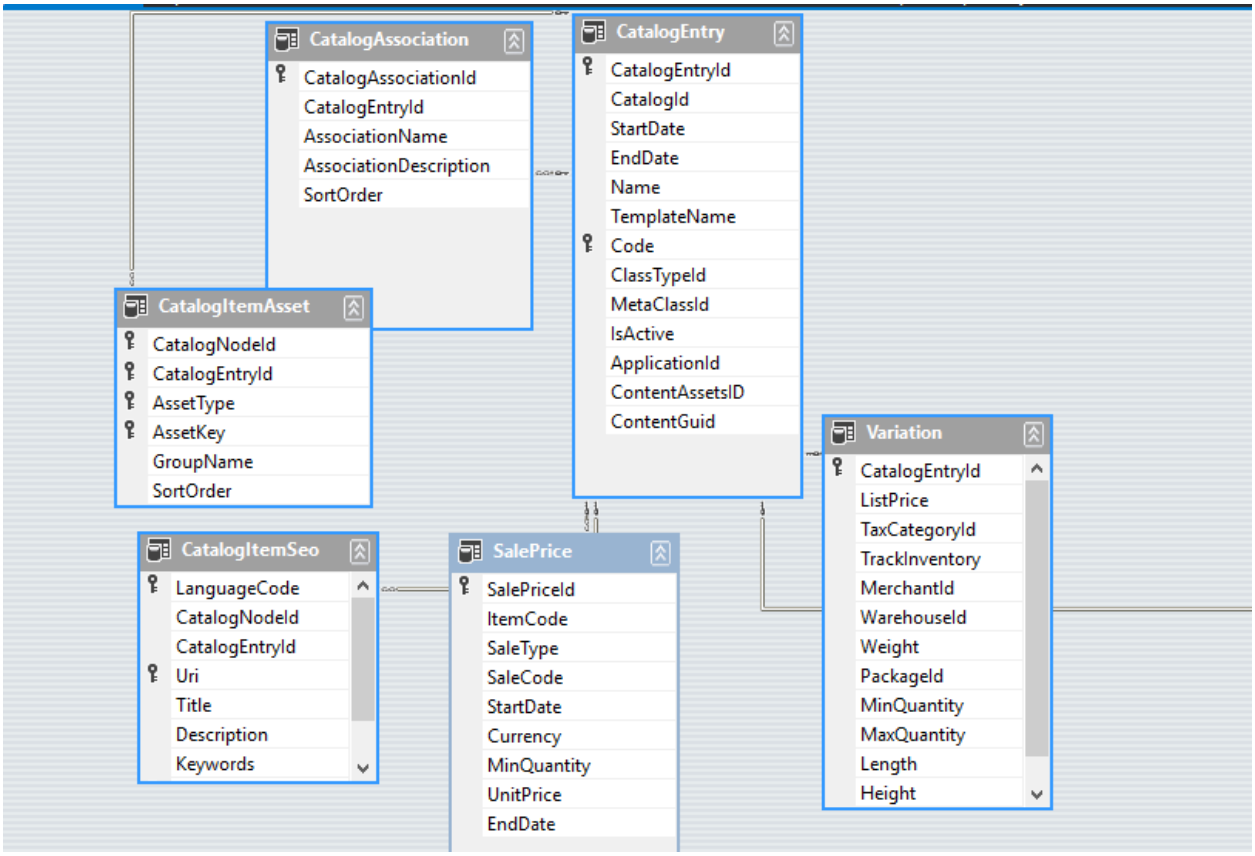
to define the shipping dimensions, but you now can use much more convenient way with the `Height`, `Width` and `Length` attributes. However, this is still useful when you have non-box SKUs, such as flower vases.

- 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. We will discuss more about this in Chapter 8
- 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.

At this point, you might want to say: "So me some code to play with". How's about No? The API:s to manage those relations and associations will be introduced later when we discuss the content API:s. You can, of course, use `ICatalogSystem` to manipulate those information, but I would advise against that. For now, let' try to play with Catalog UI and/or Catalog Management system in Commerce Manager. Try to create the relations and associations between entries and see how that's reflected at database level. We'll get into API:s soon enough.

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, and 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, proved, reliable API:s, so why not use it directly?

Firstly, 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.

Secondly, 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?

Thirdly, 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.

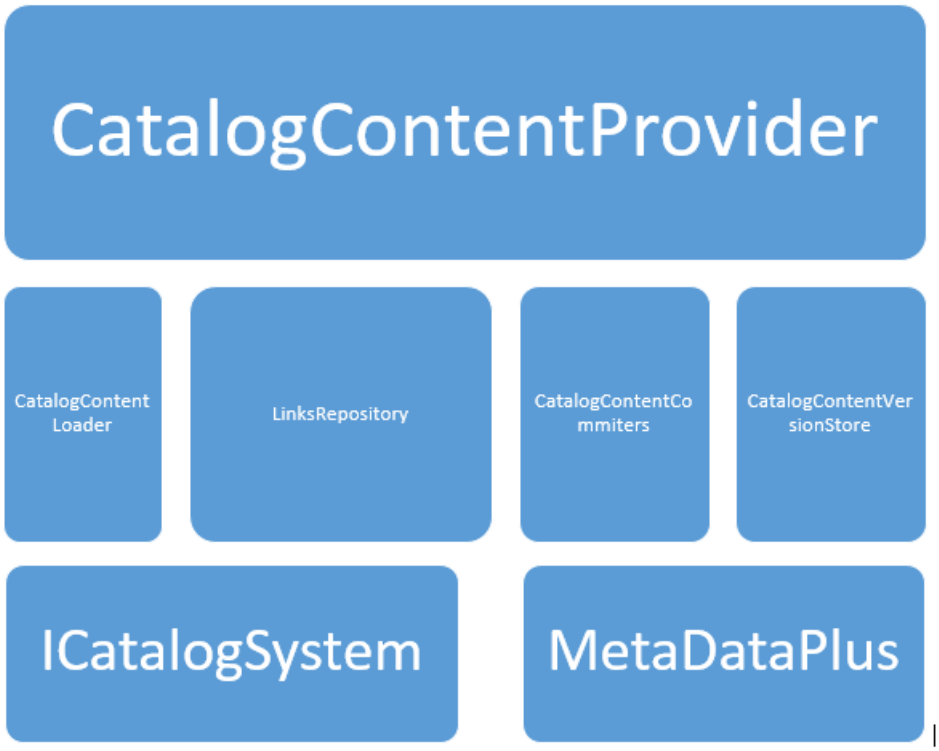
Finally, the POCO (Plain Old C# Object) approach in the content way is arguably move intuitive than the DTO:s (DataSet:s) approach of `ICatalogSystem`. Let's see how to change a name of an `EntryContentBase` vs a `CatalogEntryDto`:

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

vs

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

Tell me, which one will catch your eyes?



A simplified architecture of CatalogContentProvider

And one other approach - when you work with versions, `ICatalogSystem` simply provide nothing to work with versions. New content API:s are simply superior because they are designed to work with versioning.

In short, if you start your Episerver Commerce project today, you should be using the latest and greatest version (Episerver releases new packages of Commerce every one or two weeks, to it's always a best practice to keep to up-to-date as much as possible), and you should be using the `IContentRepository`. There are very little reasons to use `ICatalogSystem` (well, there always is a reason for something), but you should always think about the content way first. It's the way forward.

One place where you should 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 `CatalogEntrys` 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, meaning 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 within the system.
- The work id, which identifies the version of the content. [14]

[14] It was a big change in Commerce 9 in the way work id is hanled. 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 inline which 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.

- The content provider name. This allows the content system to know which provider should be handling the content - in this case - the content repository knows it should let `CatalogContentProvider` handle this content. For catalog content, it's always `CatalogContentProvider` who does the leg works. [15][16]

`ReferenceConverter` was a small class resides in `Medichase.Commerce` namespace, it's not very "small" now, but in this section we only discuss its three "base" methods:

```
public class ReferenceConverter
    {
        /// <summary>
        /// Gets a <see cref="ContentReference"/> instance with the specified co\
mmerce object ID and type
        /// encoded in the content ID.
        /// </summary>
        public virtual ContentReference GetContentLink(int objectId, CatalogCont\
entType contentType, int versionId)

        /// <summary>
        /// Gets the actual id of commerce object.
        /// </summary>
        public virtual int GetObjectId(ContentReference contentLink)

        /// <summary>
        /// Gets the type of the commerce object. Parse the content ID to take t\
wo most significant bits to
        /// determine CatalogContentType.
        /// </summary>
        public virtual CatalogContentType GetContentType(ContentReference conten\
tLink)

        ...
    }
```

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:

[15] It's `CatalogContentProvider` who does the hard work of processing catalog contents, but you should never work with it directly. The only API:s 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 working with CatalogContentProvider, you are (probaby) doing it wrong. However, I think it deserves an honorable mention for its unsung works. You're the silent hero, `CatalogContentProvider`.

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

- 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? [17]

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 (and its sibling which takes multiple codes at once) 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 another reason you should always stay with the most recent releases of Episerver.

[17] 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 conten 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 (althought it's quite easy to do so), it's the systems which need to understand it.

Chapter 2: Catalog content modeling

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 barebones, it's the MetaData system which bring it to life.

> Note that there are two classes named `MetaClass` in Episerver Commerce. The one resides in `Mediachase.MetaDataPlus.Configurator` is the one we are talking about. [18]

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 nothing more than a `HashSet` of values.

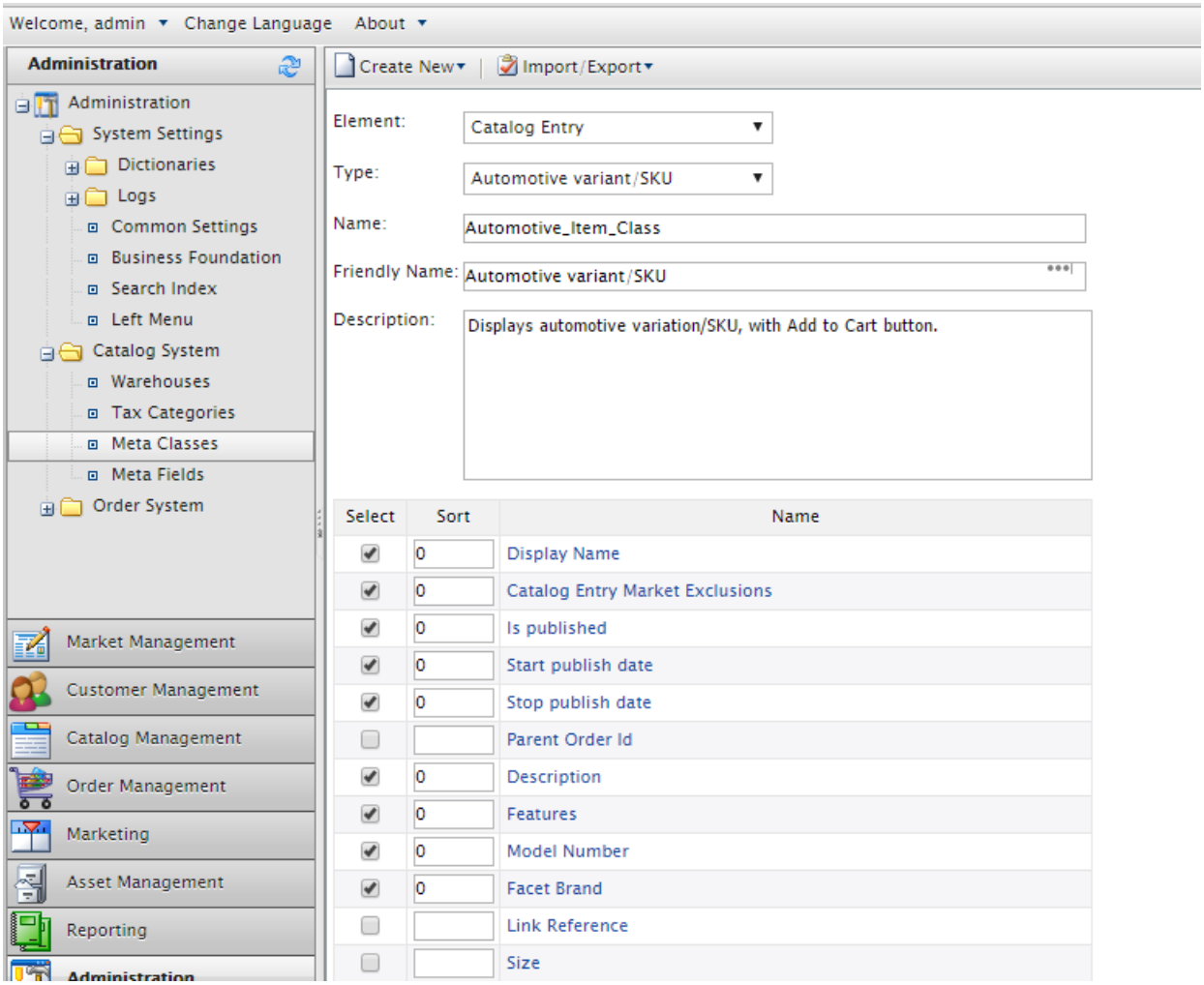
> The metadata system is used for both catalog and order. However, it is used much more intensively for catalog. The default catalog metaclasses are really simple and can't almost be used in anything. The default order metaclasses, in other hands, are much more comprehensive. In most implementations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

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

[18] (yes, it's not the best naming 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.)

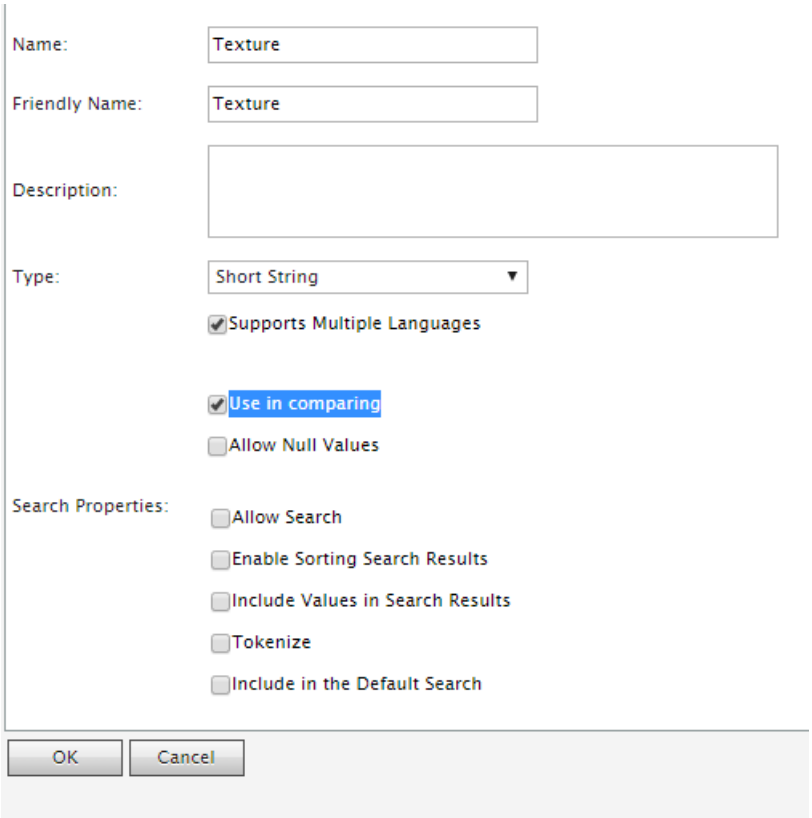
the `MetaClass` management in Commerce Manager (Commerce Manager/Administration/Catalog System/MetaClass), it's as close as bare metal you can get (without looking at database, of course)



A MetaClass

The key feature of MetaData system is the extensibility. You can attach or remove a metafield from a metaclass on-the-fly. (if you come from Commerce R3 and earlier, it was not on-the-fly, you'll have to restart the site for it to take effect. But hey, why would you still use that version?). You can share metafields between metaclasses (yes, pretty cool, I know). There is no hard limit of how many metafields can be in a metaclass, but my opinion is you should not exceed 50. (30 is even better, but there are products with complex specificiations).

It's metafield which is more interesting



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 for this metafield. For example, the `DisplayName` for this book can be "Pro Episerver Commerce" in "English", but can be "Professionell Episerver Commerce" in "svenska".
- 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. If this is set to false, the editor must set a value for it before saving.
- 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:

```
public enum MetaDataType
{
        Integer                         = 26,
        Boolean                         = 27,
        Date                          = 28,
        Email                          = 29,
        URL                                  = 30,
        ShortString                   = 31,
        LongString                   = 32,
        LongHtmlString        = 33,

        DictionarySingleValue = 34,
        DictionaryMultiValue = 35,
        EnumSingleValue                 = 36,
        EnumMultiValue                 = 37,
        StringDictionary        = 38,
        File                                 =        39,
        ImageFile                        =        40,

        MetaObject                        =        41
}
```

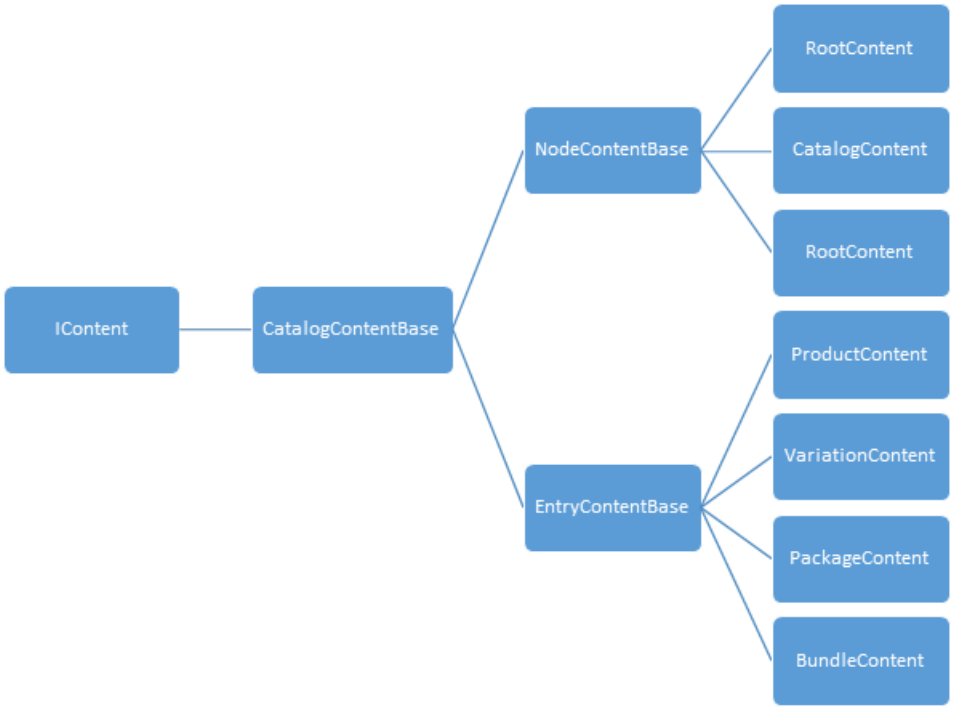
The only caveat of choosing a type is that you can’t change the type of a metafield after you saved it. So if you make a wrong move, it might be easiest to just delete the metafield and create again.

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` - when you only need to load the contents, not to save them), 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 that interface.

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`, which itself implements `IContent` and then add some properties, such as `ApplicationId`.

`RootContent` is a "virtual" content - it has no Mediachase eCF Catalog counterpart. You can only have one `RootContent` in the system, its `ContentLink` is fixed and it is always readonly. 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()`. (The only places where we need to use `GetRootLink` are when we need to work with `CatalogContent`, for example, to load all catalogs in the system.)

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, right, you'll usually do not write code like this in your daily work)

```
    //Get the dependencies
    var contentRepository = ServiceLocation.ServiceLocator.Current.GetInstance<I\
ContentRepository>();
    var referenceConverter = ServiceLocation.ServiceLocator.Current.GetInstance<\
ReferenceConverter>();

    //Create a CatalogContent
    var catalogContent = contentRepository.GetDefault<CatalogContent>(referenceC\
onverter.GetRootLink());
    catalogContent.Name = "Pro Episerver Commerce";
    catalogContent.DefaultCurrency = Currency.USD;
    catalogContent.DefaultLanguage = "en";
    catalogContent.WeightBase = "kgs";
    var catalogContentLink = contentRepository.Save(catalogContent, DataAccess.S\
aveAction.Publish, EPiServer.Security.AccessLevel.NoAccess);

    //Edit it to add Swedish
    catalogContent = contentRepository.Get<CatalogContent>(catalogContentLink).C\
reateWritableClone<CatalogContent>();
    catalogContent.CatalogLanguages.Add("sv");
    contentRepository.Save(catalogContent, DataAccess.SaveAction.Publish, EPiSer\
ver.Security.AccessLevel.NoAccess);

    //Delete it
    contentRepository.Delete(catalogContentLink, false, EPiServer.Security.Acces\
sLevel.NoAccess);
```

Well, it's easy, isn't it? If you worked with CMS before, then even without explaining, you should be able to understand the code. I told you, you'll like the new content way.

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 deratives.

This will simply does not work:

```
using EPiServer.Commerce.Catalog.ContentTypes;
using EPiServer.Commerce.Catalog.DataAnnotations;

namespace EPiServer.Commerce.Sample
{
    [CatalogContentType(GUID = "1FD3E34D-6476-40E2-8CB5-8EFB0DB79E3E")]
    public class MyCatalogContent : CatalogContent
    {
        public virtual string MyExtendedProperty { get; set; }
    }
}
```

`CatalogContentScannerExtension` will throw an `EPiServerException` exception during the initialization if it detects any content types inherit from `CatalogContent` [19], something like this:

The content type 'EPiServer.Commerce.Sample.MyCatalogContent' extends 'EPiServer.Commerce.Catalog.ContentTy which is not supported.

The code to get `IContentRepository` and `ReferenceConverter` as I showed above is just for convenience. In reality, you should use constructor dependencies pattern to supply dependencies to your classes. It's another best practice which you should follow (Episerver uses that approach heavily in their code, by the way)

Working with Nodes and Entries is more or less the same. Well, not entirely true. You'll have to model your metaclasses first.

Modeling catalog content types

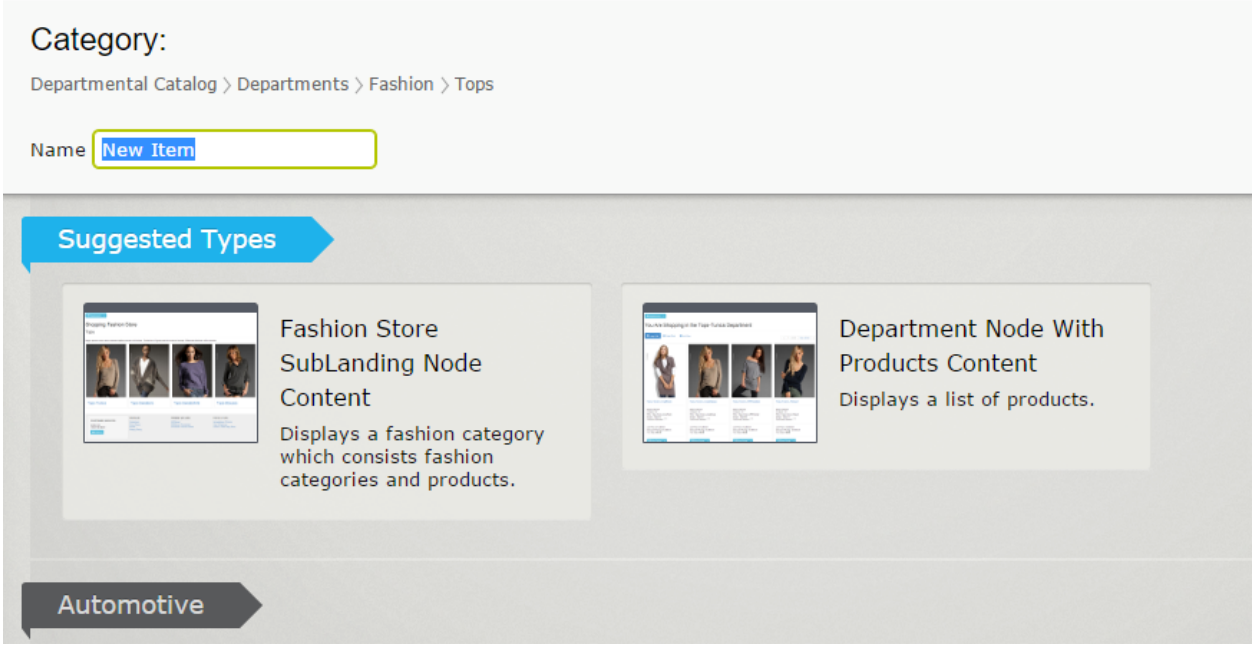
It's not required to model your metaclasses. You can use the defined classes such as `NodeContent` or `ProductContent`, and throw everything into Property properties, instead of `content.VeryLongPropertyName`, you can use `content.Property["VeryLongPropretyName"]`. (Remember? Any instance of `IContent` 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.

[19] The funny thing is the limitation only applies if you have the `CatalogContentType` attribute (which will be explained later). If you have no such attribute, it'll still work, but when you save `MyExtendedProperty`, it will be saved to nowhere.

```
/// <summary>
/// FashionProductContent class, map to Fashion_Product_Class metadata class
/// </summary>
[CatalogContentType(GUID = "18EA436F-3B3B-464E-A526-564E9AC454C7", MetaClassName\
 = "Fashion_Product_Class", GroupName = "Fashion",
    DisplayName = "Fashion product", Description = "Display fashion product with\
 Add to Cart button.")]
[ImageUrl("~/Templates/UX/gfx/page-type-thumbnail-fashion.png")]
public class FashionProductContent : ProductContent
{
    [Display(Name = "Description", Order = -15)]
    public virtual XhtmlString Info_Description { get; set; }

    [Display(Name = "Features", Order = -11)]
    public virtual XhtmlString Info_Features { get; set; }

    [Display(Name = "Model Number", Order = -3)]
    public virtual string Info_ModelNumber { get; set; }

    [Display(Name = "Facet Brand", Order = -1)]
    [Searchable]
    public virtual string Facet_Brand { get; set; }
}
```

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.
- `ImageUrl` 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 the metafield. Casing is not important. So if you have a string metafield named `facet_color`, your property must be string `Facet_Color`.

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

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

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

- The property must be declared as `virtual`.
- As a metafield can be used across multiple metaclass, the property which maps to it also has to be identical across content types. It does mean you can have a property named `Facet_Color` as string in one class and another named `Facet_Color` as double in another class. It also has to have the same attributes decorated. 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:

```
[BackingType(typeof(PropertyDictionaryMultiple))]
public virtual IEnumerable<string> MultipleValue { get; set; }

[BackingType(typeof(PropertyDictionarySingle))]
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

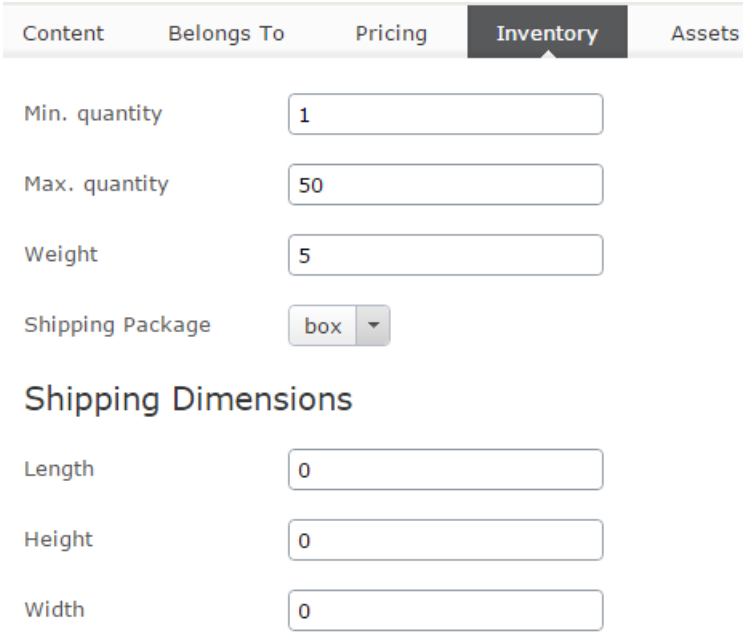
Most of properties of the base content types provided by Episerver Commerce (CatalogContent, NodeContent, ProductContent, etc) are flat, which means you can access directly, for example:

```
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. So to set the length of a variation, you'll have to do like this:

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

The reason? Unlike other fields which have existed since before Commerce 7.5, shipping dimensions were added quite recently (and by "Recently" I mean Commerce 8.7). 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 chosed the safe way and added a block instead. That's why the shipping dimensions are displayed a bit differently in Catalog UI:



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:

```
variationContent.ShippingDimensions = new ShippingDimensions();
```

Just use the block directly, as we showed above.

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.)

Prior to Commerce 9 (8+ and earlier), each metaclass has two tables to represent it [20]. Usually it's a table with the metaclass name, and another table with that name and _Localization suffix (But it does not always work that way, you can create a metaclass with a custom table name using MetaClass API:s, the only way to know for sure is to look at column `TableName` in `MetaClass` table)

As you might guess, the normal table is used to store non-localization metafields, while the _-Localization stores the other ones (the ones with `CultureSpecific` attribute decorated, or "Support multiple languages" option enabled). Each metafield will be on one table or the other, and they are added/removed dynamically.

Yes, everytime you add a metafield to a metaclass, a new column will be added to your table. And when you remove it, it will be dropped.

It sounds to be alarming if you have a big catalog with one million entries or so, but mostly the operations of adding and removing metafields will be done along deployments, so it might not as bad as it sounds.

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.

	MetaFieldName	LanguageName	Boolean	Number	FloatNumber	Money	Decimal	Date	Binary	String	LongString
1	DisplayName	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	Tops-Tunics-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 sit amet, co
7	Info_Features	en	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	<ul> <li> Feature 1 </li> <li> 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-Tunics-OffShoulder

CatalogContentProperty table, it's new in Commerce 9

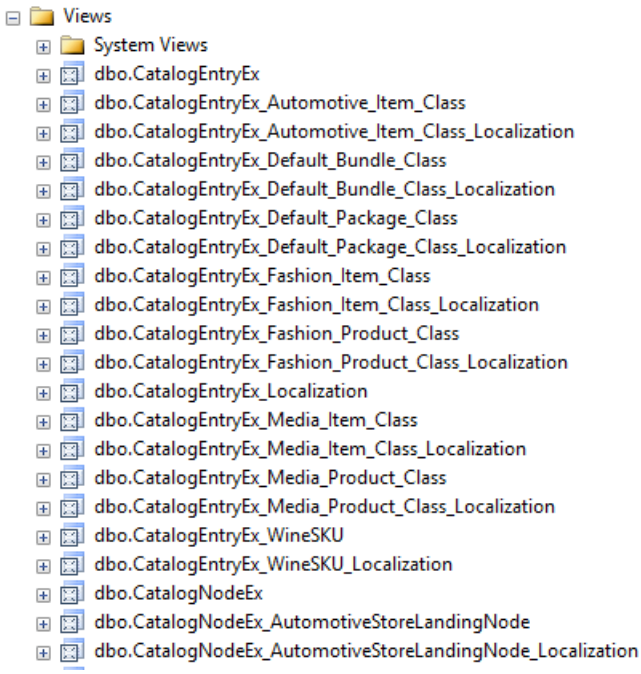
Commerce 9 has a new table - `CatalogContentProperty` to store property. Besides the identity columns, there are 10 columns for the datatypes `MetaDataPlus` supports: `Boolean`, `Number` (Integer), `FloatNumber`, `Money`, `Decimal` [21], `Date`, `Binary`, `String` (for strings shorter than 256 characters 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.

[20] If you look even further back, or your database is upgraded from a version prior to 7.5, there might be another table with _History suffix. This table was intended to store the history of metafields (which enables history support option). However, the feature had never really worked and was removed in later version of Episerver Commerce.

[21] Since Commerce 9, the `Money` column has been there. But it turned out to have not enough precisions in some case, so `Decimal` column was added later (Commerce 9.7) and all the data in Money column has been moved to the new column. Since 9.7, all `decimal` values are stored in Decimal column.

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 (and then can possibly yield better performance). 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 everytime 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 everytime you update your metaclasses (just like the way old metaclass tables did).



The new views to replace metaclass tables

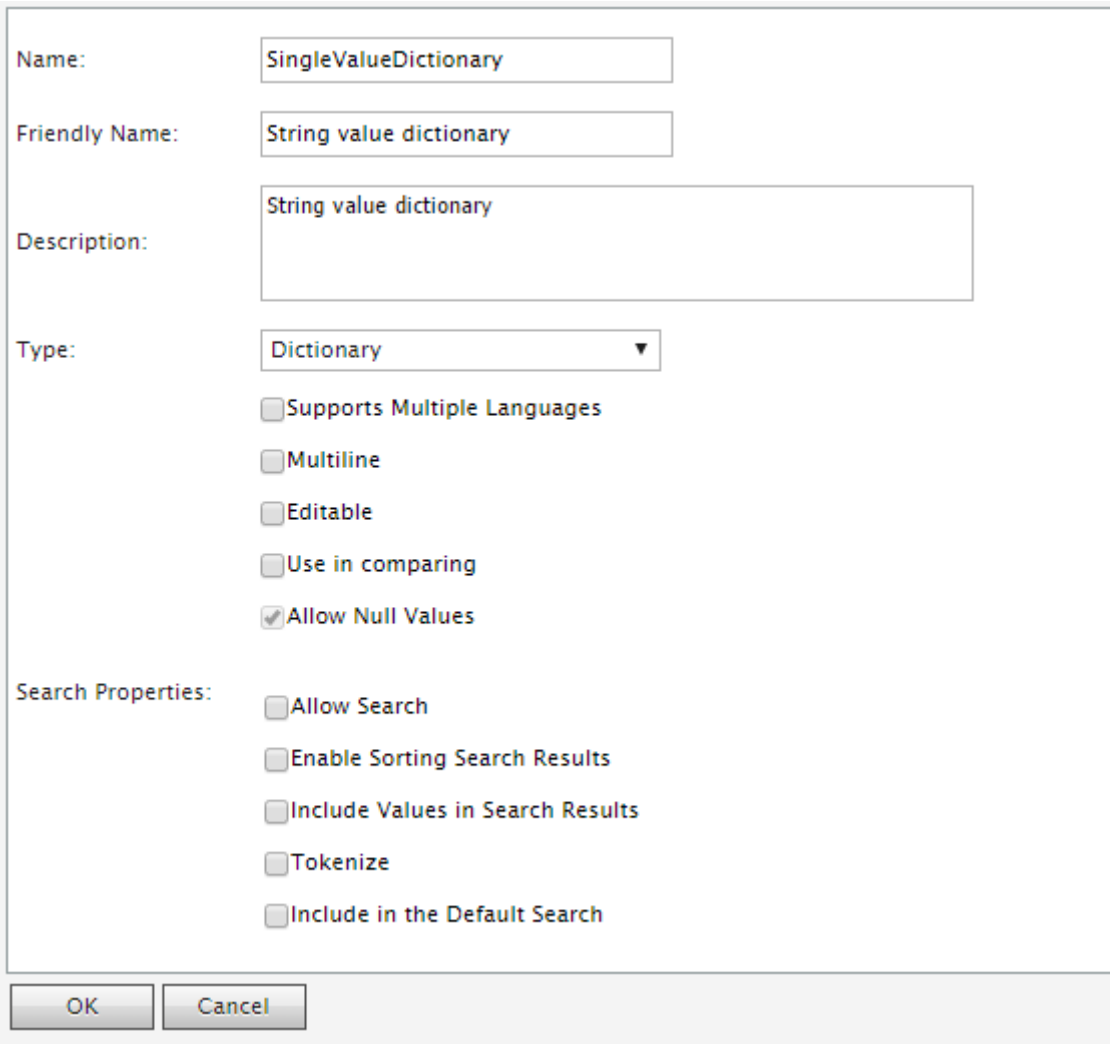
Those views are, of course, slower than the original tables, some tests shows that they can be 30% slower than the original tables. However, you'll not use them frequently enough for the difference to really make senses.

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.



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)`

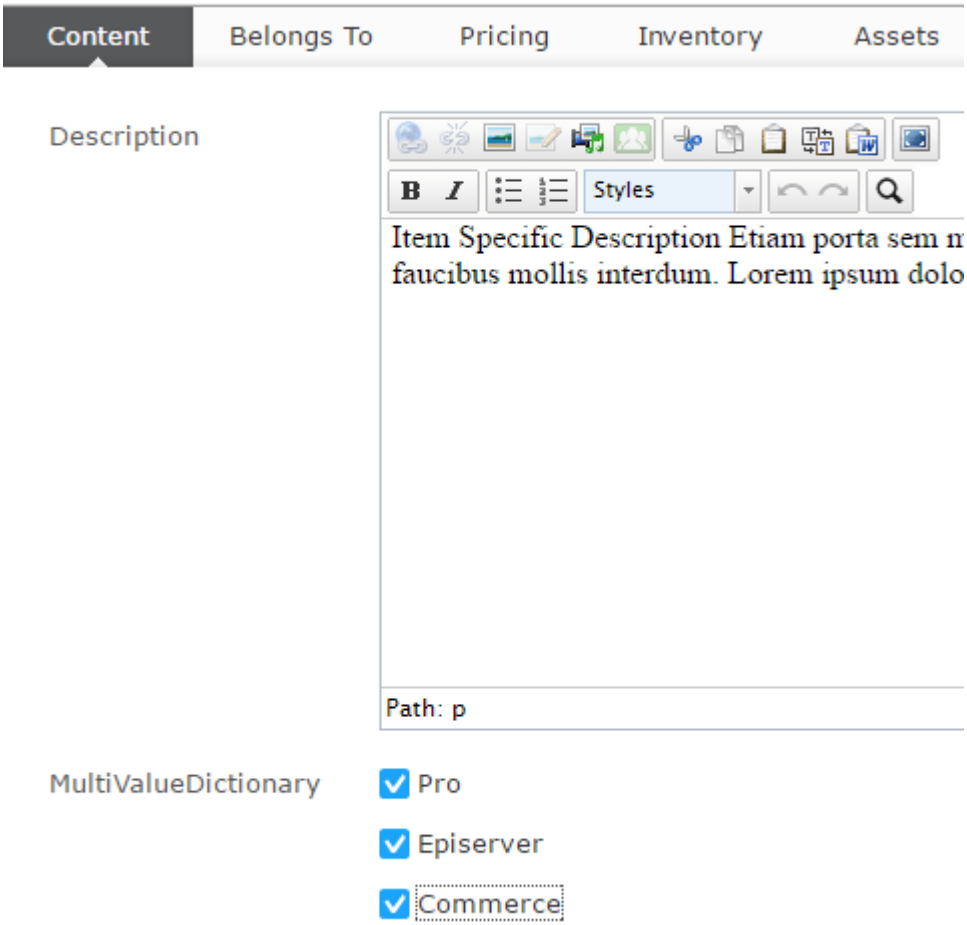
```
[BackingType(typeof(PropertyDictionarySingle))]
public virtual string Color { get; set; }
```

- 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.

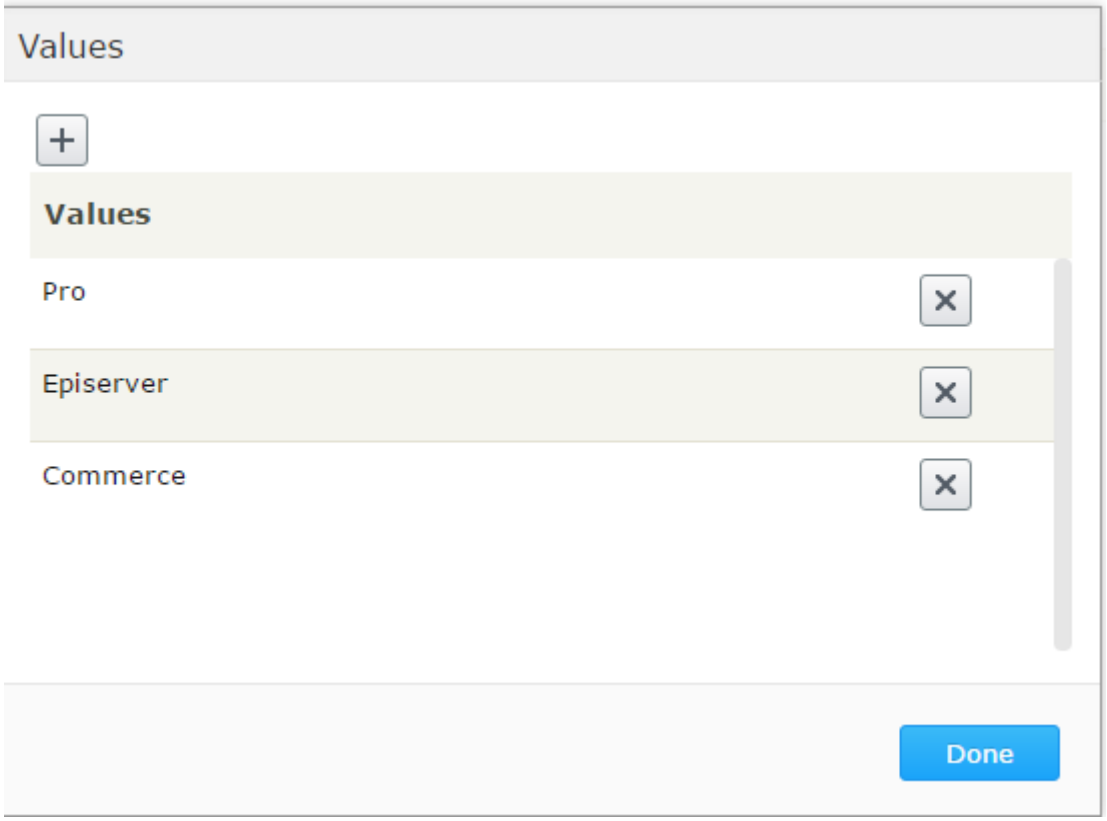
You can define a property for Multi value dictionary in content type by `IEnumeable<string>` and `typeof(PropertyDictionaryMultiple)` backing type

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

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



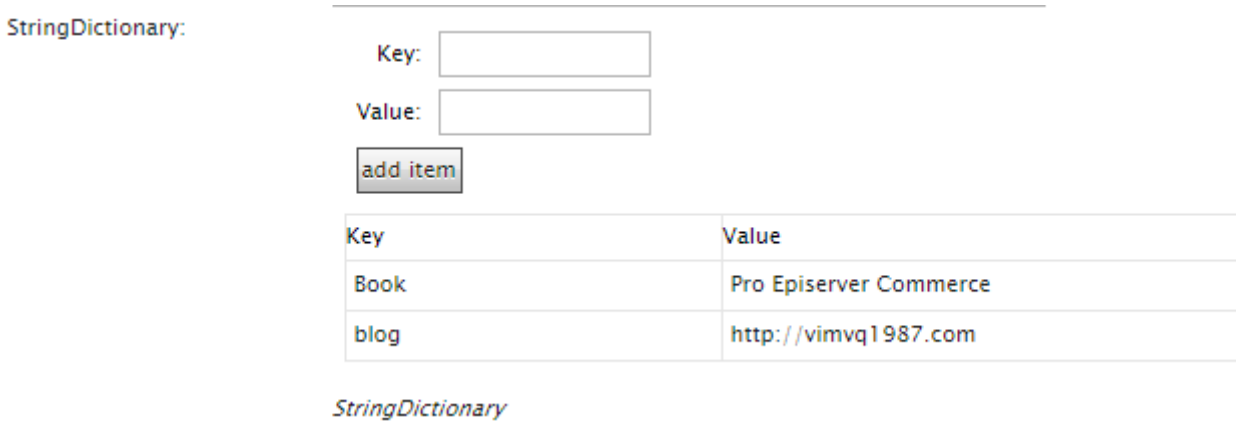
and administration:



Manage values for a dictionary field in Settings view

- 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



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.

Those information might not be really interesting to you - but they can be useful in some specific scenario. Let's consider some of those cases:

- You want to know which entries use a specific dictionary value. In previous example, we have a property named Color, we want to find all entries with Color is 'Blue'. For front-end site, it would be easy to find such entries by search feature (will be discussed later), but what if you want to create a report for that? Episerver Commerce does not provide such functionality out of the box, so we'll have to craft it ourselves. Time for some SQL then!

```
    DECLARE @MetaFieldId INT
    DECLARE @MetaDictionaryId INT

    SET @MetaFieldId = (SELECT MetaFieldId FROM MetaField WHERE Name = 'Color' A\
ND Namespace = 'Mediachase.Commerce.Catalog')
    SET @MetaDictionaryId = (SELECT MetaDictionaryId FROM MetaDictionary WHERE V\
alue = 'Blue' AND MetaFieldId = @MetaFieldId)

    SELECT ObjectId FROM CatalogContentProperty WHERE MetaFieldId = @MetaFieldId\
 AND Number = @MetaDictionaryId AND ObjectTypeId = 0
```

This script is quite simple - we need to get the Id of `Color` MetaField first, then the `MetaDictionaryId` of the 'Blue' color. When we have two values, we can simple query from table `CatalogContentProperty` to find which entries have that value. You can go even further to join with `CatalogEntry` table to get more information such as name or code - I'll leave that to you.

- Let's consider another case - we need to report which entries belong to a specific Market. To determine which entries belong to which markets, Episerver Commerce uses a special metafield, named `_ExcludedCatalogEntryMarkets` which is a multi-value dictionary. As its name might suggest, it contains list of the MarketId which the entry *does not* belong to, for example, if `_ExcludedCatalogEntryMarkets` contains 'US' then the entry is not available in 'US' market. So if we are to find entries which belong to 'US' market, we have to find entries which do not have 'US' value for `_ExcludedCatalogEntryMarkets`.

```
    DECLARE @MetaFieldId INT
    DECLARE @MetaDictionaryId INT

    SET @MetaFieldId = (SELECT MetaFieldId FROM MetaField
    WHERE Name = '_ExcludedCatalogEntryMarkets' AND Namespace = 'Mediachase.Comm\
erce.Catalog')
    SET @MetaDictionaryId = (SELECT MetaDictionaryId FROM MetaDictionary WHERE V\
alue = 'US' AND MetaFieldId = @MetaFieldId)

    SELECT ObjectId FROM CatalogContentProperty
    WHERE
    MetaFieldId = @MetaFieldId AND
    Number NOT IN
    (
    SELECT mk.MetaKey from MetaMultiValueDictionary mmv
    INNER JOIN MetaKey mk on mmv.MetaKey = mk.MetaKey
    WHERE mk.MetaFieldId = @MetaFieldId
    AND MetaDictionaryId = @MetaDictionaryId
    )
```

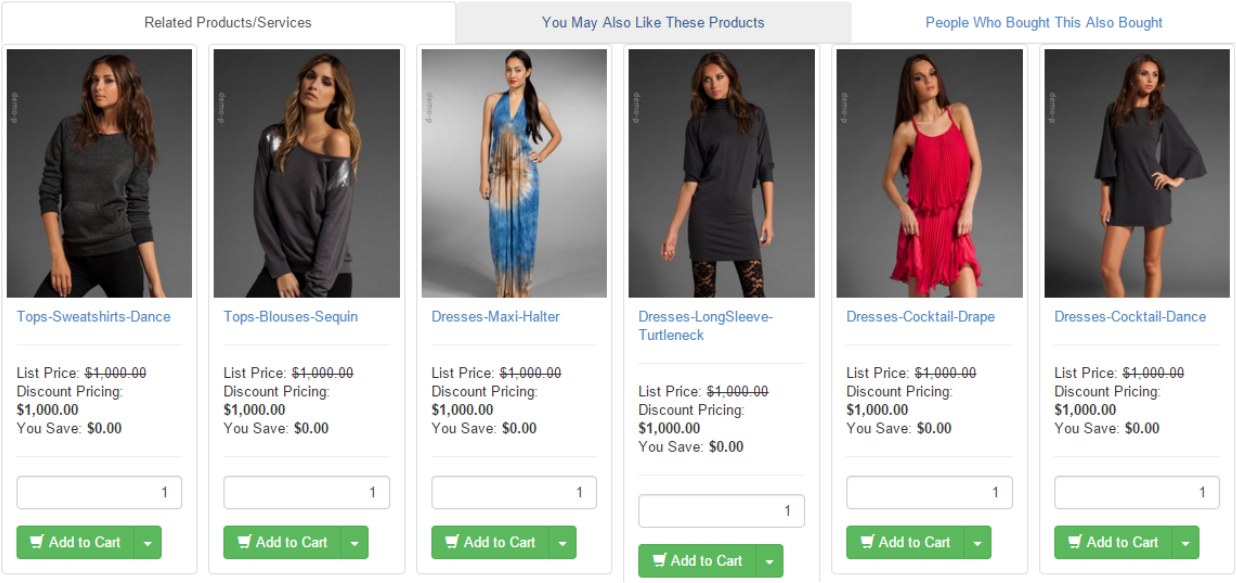
Same as previous script, we need to find the Id of `_ExcludedCatalogEntryMarkets` MetaField, then the `MetaDictionaryId` of 'US' market. The next statement is tricky - we need to find MetaKey which value matching value for 'US', then except them. Again, you can join with other tables for more data.

Chapter 3: Associations, relations and assets

Working with associations and relations

Managing associations and relations in `CatalogContentprovider` is done by `ILinksRepository` (which itself inherrits from `IAssociationRepository` and `IRelationRepository`). The default implementations of those interfaces are built on top of `ICatalogSystem`, while they are positioned in lower layer of `CatalogContentProvider`.

`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

```
public interface IAssociationRepository
{
    /// <summary>
    /// Gets the associations for the catalog content specified by the content l\
ink.
    /// </summary>
    IEnumerable<Association> GetAssociations(ContentReference contentLink);

    /// <summary>
    /// Removes the associations.
    /// </summary>
    void RemoveAssociations(IEnumerable<Association> associations);

    /// <summary>
    /// Updates matching associations and adds new associations for an entry.
    /// </summary>
    void UpdateAssociations(IEnumerable<Association> associations);
}
```

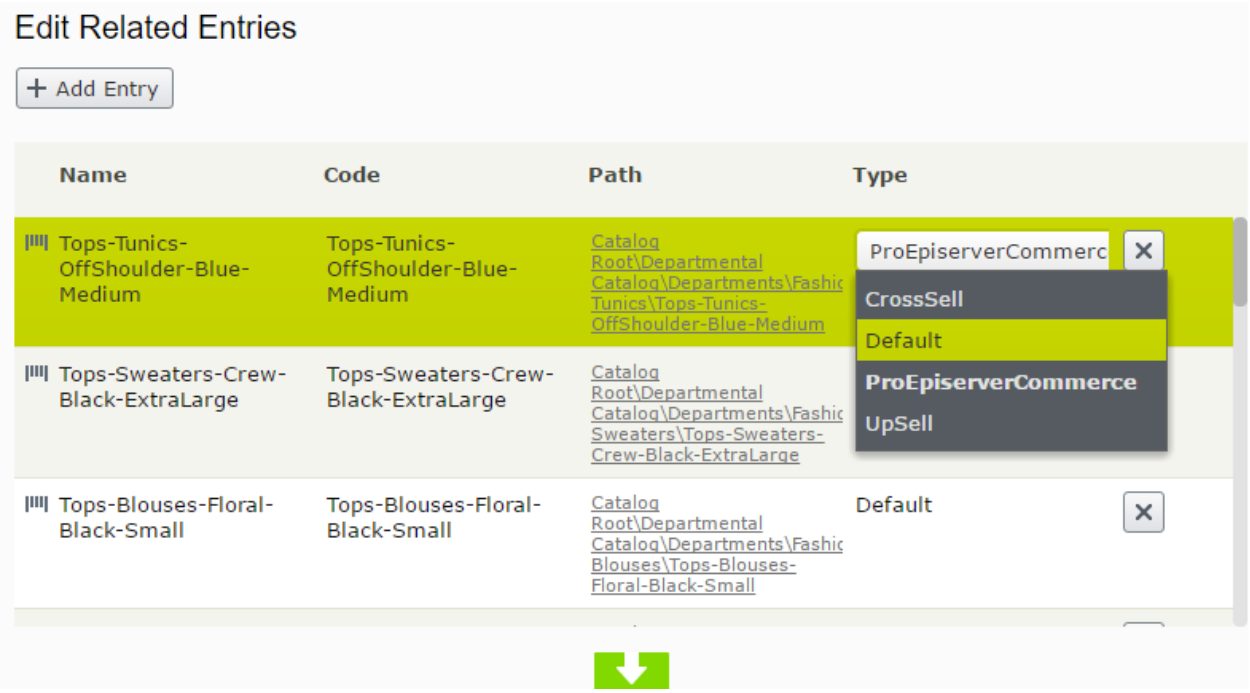
Associations and Relations groups.

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:

```
    var associationDefinitionRepository =
        context.Locate.Advanced.GetInstance<GroupDefinitionRepository<Associatio\
nGroupDefinition>>();
    associationDefinitionRepository.Add(new AssociationGroupDefinition { Name = \
"CrossSell" });
    associationDefinitionRepository.Add(new AssociationGroupDefinition { Name = \
"Upsell" });
    associationDefinitionRepository.Add(new AssociationGroupDefinition { Name = \
"ProEpiserverCommerce" });
```

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

`IRelationsRepository`, in the other hands, is much more complex and hard to follow. The problem with it, is, it tried to unified the relations between nodes, between entries, and between nodes and entries into one model. It'll be hard to come up with really good method names, and you'll have to stick with some abstractions. Here comes the Source and the Target.

```
public interface ILinksRepository
{
    IEnumerable<Relation> GetRelationsBySource(ContentReference contentLink);

    IEnumerable<T> GetRelationsBySource<T>(ContentReference contentLink) where T\
 : Relation;

    IEnumerable<Relation> GetRelationsByTarget(ContentReference contentLink);

    IEnumerable<T> GetRelationsByTarget<T>(ContentReference contentLink) where T\
 : Relation;

    void RemoveRelations(IEnumerable<Relation> relations);

    void UpdateRelations(IEnumerable<Relation> relations);

    void SetNodeParent(ContentReference contentLink, ContentReference newParentL\
```

```
ink);
}
```

You might read the documentation for each method to understand which one to use, but I doubt you will remember which is which. I've been there, staying confused to guess which one to use. However, it might be a little easier if you look at `Relation` types. It's a simple class with three properties: `SortOrder`, `Source` and `Target`.

And this is from the documentation:

- For a `NodeRelation`, `Source` is the `ContentReference` of the categorized item (aka children), while for an `EntryRelation` (`ProductVariation`, `PackageEntry`, `BundleEntry`), it is the product/package/bundle itself.
- And for a `NodeRelation`, `Target` is the `ContentReference` of the category, while for an `EntryRelation`, it is the entries (in the product/package/bundle).

You have `ProductVariation`, `PackageEntry`, `BundleEntry` and `NodeRelation` extend from `Relation` with extra information.

With all due respects, I don't think this is an example of good API:s - a good API:s should be easy to use without looking at the documentation and hard to misuse. I almost always forget which is Source and which is the Target. Fortunately, we can write our own extension methods, which have better naming and easier to remember and understand. This will be improved in Commerce 10.

A better ILinksRepository

Make no mistake, designing a good API:s is hard (probably as hard as writing a good book, which I am trying to do). We simply can't replace `ILinksRepository`, but we can create a bunch of extension methods which use it in a simple, easy understandable way to build up the functions we need. The following methods will return a list of `ContentReference` of the variations belong to a product, the entries belong to a package and bundle, respectively.

```
    public static IEnumerable<ContentReference> GetVariations(this ProductConten\
t productContent, ILinksRepository linksRepository)
    {
        return linksRepository.GetRelationsBySource<ProductVariation>(productCon\
tent.ContentLink).Select(r => r.Source);
    }

    public static IEnumerable<ContentReference> GetPackageEntries(this PackageCo\
ntent packageContent, ILinksRepository linksRepository)
    {
        return linksRepository.GetRelationsBySource<PackageEntry>(packageContent\
.ContentLink).Select(r => r.Source);
    }

    public static IEnumerable<ContentReference> GetBundleEntries(this BundleCont\
ent bundleContent, ILinksRepository linksRepository)
    {
        return linksRepository.GetRelationsBySource<BundleEntry>(bundleContent.C\
ontentLink).Select(r => r.Source);
    }
```

Those method will return the parent-children relations (The source is the "parent"). We can have reversed method to get the relations the way around. The following method will return the parent products of a variation, parent packages and bundles of an entry, respectively:

```
    public static IEnumerable<ContentReference> GetProducts(this VariationConten\
t variationContent, ILinksRepository linksRepository)
    {
        return linksRepository.GetRelationsByTarget<ProductVariation>(variationC\
ontent.ContentLink).Select(r => r.Source);
    }

    public static IEnumerable<ContentReference> GetPackages(this EntryContentBas\
e entryContent, ILinksRepository linksRepository)
    {
        return linksRepository.GetRelationsByTarget<PackageEntry>(entryContent.C\
ontentLink).Select(r => r.Source);
    }

    public static IEnumerable<ContentReference> GetBundles(this EntryContentBase\
 entryContent, ILinksRepository linksRepository)
    {
```

```
        return linksRepository.GetRelationsByTarget<BundleEntry>(entryContent.Co\
ntentLink).Select(r => r.Source);
    }
```

The `ILinksRepository` parameter is optional, it was added to allow the testability. You can add another overload without it even simpler to use.

The final thing is node-node relations, and node-entry relations. We only need to get parent nodes of a node and parent nodes of an entry, you can always get the children entries or nodes of a node by using `IContentRepository.GetChildren<T>`, which I believe to be easier way to remember.

```
    public static IEnumerable<ContentReference> GetParentCategories(this EntryCo\
ntentBase entryContent, ILinksRepository linksRepository)
    {
        return linksRepository.GetRelationsBySource<NodeRelation>(entryContent.C\
ontentLink).Select(r => r.Target);
    }

    public static IEnumerable<ContentReference> GetParentCategories(this NodeCon\
tent nodeContent, ILinksRepository linksRepository)
    {
        return linksRepository.GetRelationsBySource<NodeRelation>(nodeContent.Co\
ntentLink).Select(r => r.Target);
    }
```

Note that `GetParentCategories` will only return the parent categories which nodeContent is linked to, it does not include the true "parent" which is specified by `nodeContent.ParentLink`.

As I mentioned above, the `ParentLink` content of an entry currently points the node with lowest `SortOrder`. This is not entirely correct, the `SortOrder` should be used when you display the list of parent categories. This can be fixed in up coming versions of Commerce.

Assets

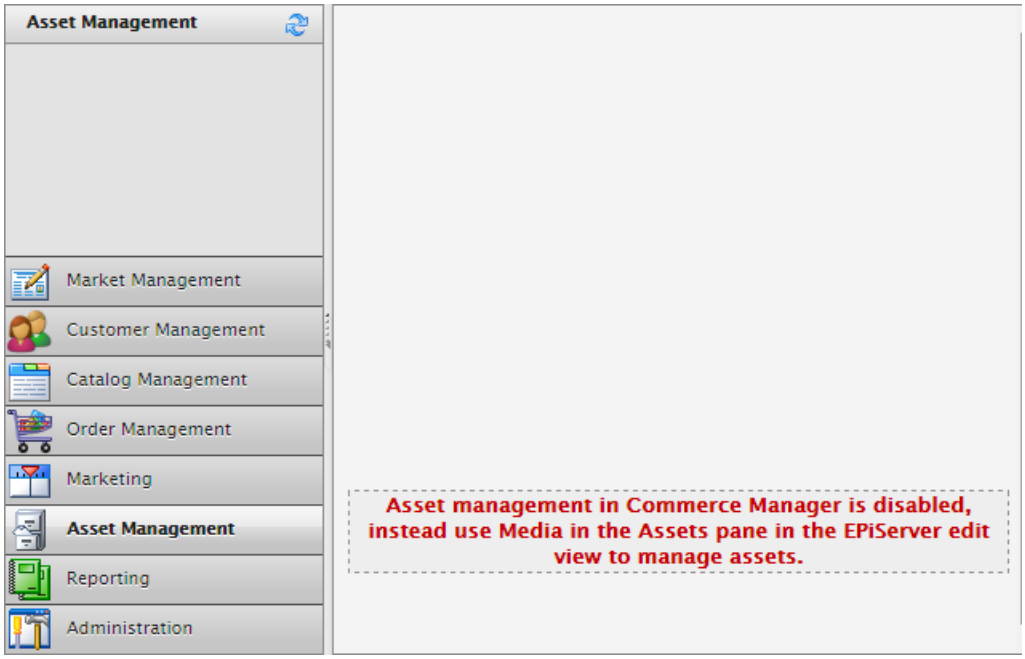
Products always need some assets. Images, videos, documentations such as specifications and manuals,...As a developer/an editor, you'll need a way to work with assets.

There are two ways to work with assets [22]. The old one sticks with `ICatalogSystem` - you'll have to access `CatalogItemAsset` datatable directly. The new one works via the `IAssetContainer`

[22] You'll start wondering yourself: Are there always two systems in Episerver Commerce? Not always, but for many parts, yes. There are places where the old systems do not fill the shoes, and new things must take over. However, Episerver takes it very seriously when it comes to backward compatibility. The features people are using will still be supported.

interface (which is implemented by `EntryContentBase` and `NodeContent`), and is accessible as `CommerceMediaCollection`.

The new way is, of course, the recommended way, and it's even enabled by default. There is a hidden flag - `UseLegacyAssetSystem` to let the system know which way you want to work with assets. If it's set to true, the Media tab in Catalog UI will be disabled. By default, the Assets tab in Commerce Manager is disabled.



This is how you see it by default

If you start a new site, there is no reason to work with old asset system. If you upgraded from R3 or earlier to 7.5 and later, you should migrate your site to use new asset system. There is a tool[23] to do that. There are two things the new one is definitely better:

- It's unified API:s, you can work with catalog content all the ways.
- The new asset system integrates with CMS asset system, which allows some nice features such as CDN.

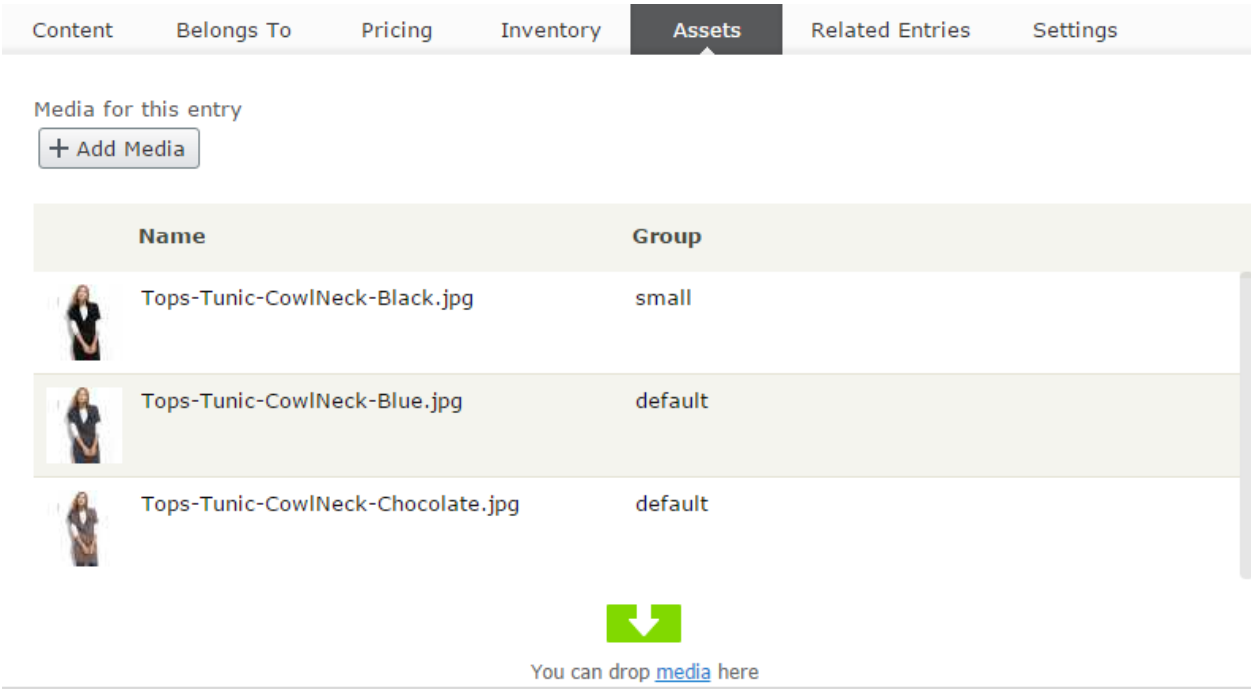
OK it's enough of theory, let's have some code.

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

[23] http://world.episerver.com/documentation/items/upgrading/episerver-commerce/75/migrating-assets-to-the-media-system/

```
public class CommerceMedia : ICloneable
{
    public ContentReference AssetLink { get; set; }

    public string AssetType { get; set; }

    public string GroupName { get; set; }

    public int SortOrder { get; set; }

    //Other methods and properties omitted for brevity.
}
```

Those properties should explain themselves. `AssetLink` is the `ContentReference` to the asset (which is supposed to be another CMS 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.



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:

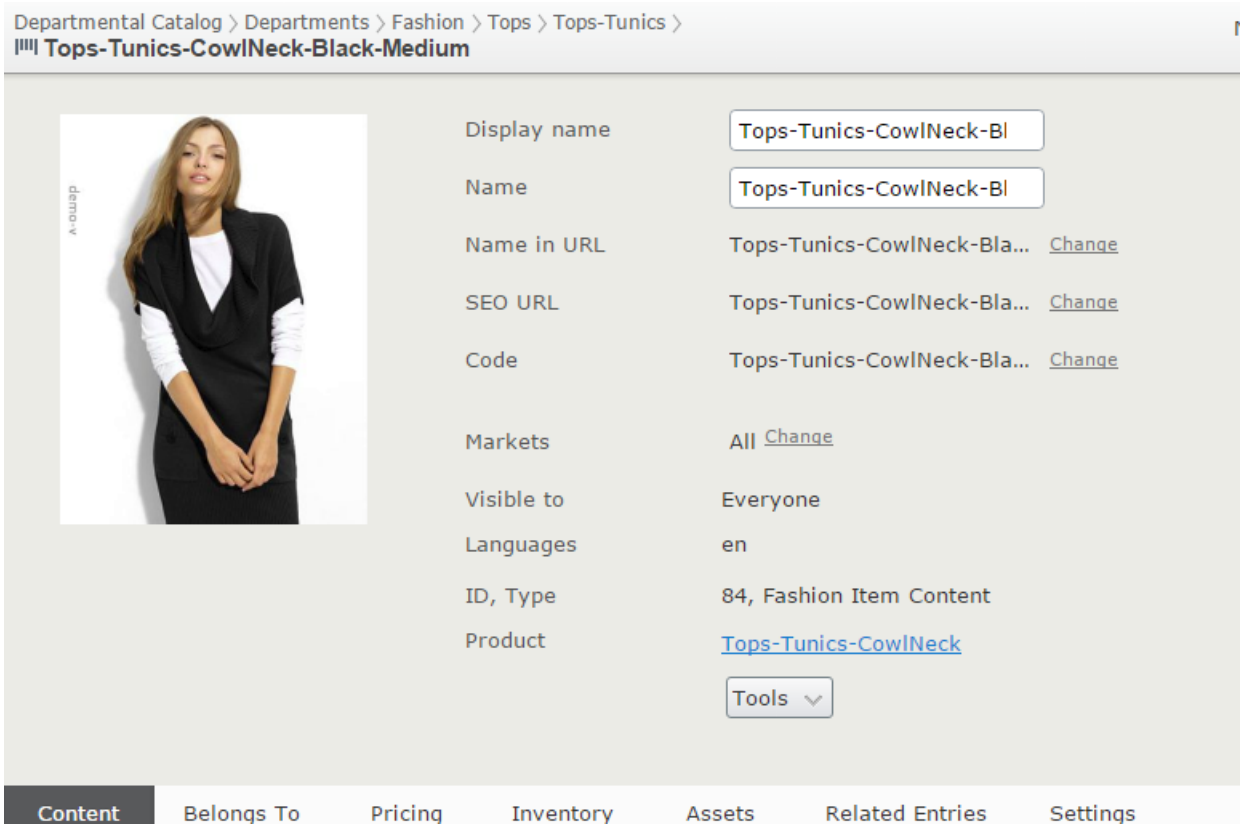
```
using System;
using EPiServer.Commerce.SpecializedProperties;
using EPiServer.DataAnnotations;
using EPiServer.Framework.DataAnnotations;

namespace EPiServer.Commerce.Sample.Models.Files
{
    [ContentType(GUID = "872AA39E-5B79-43BF-B7D5-F34D415553BD")]
    [MediaDescriptor(ExtensionString = "jpg,jpeg,jpe,ico,gif,bmp,png")]
    public class ImageFile : CommerceImage
    {
        /// <summary>
        /// Gets or sets the description.
        /// </summary>
        public virtual String Description { get; set; }
    }
}
```

This is CMS-related stuff. The most important thing in this example class is the `MediaDescriptor` attribute, which allows you to specific which extensions you want to handle by this class. So in this example, any jpg, jpeg, jpe 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:

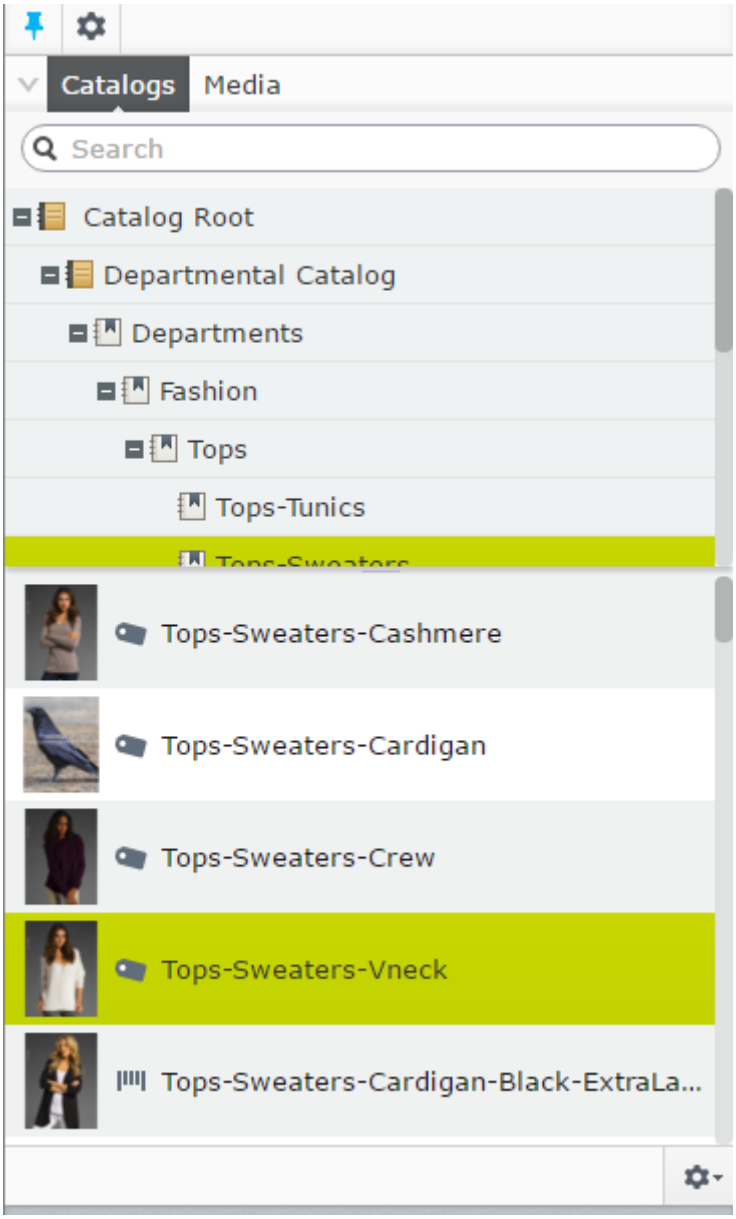
```
[ImageDescriptor(Width = 256, Height = 256)]
public virtual Blob LargeThumbnail { get; set; }

[ImageDescriptor(Width = 48, Height = 48)]
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

The second one is inherited from MediaData, it defines a `Blob` named `Thumbnail` with the size of 48x48. You can still guess it? 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:

```
[ContentType(GUID = "872AA39E-5B79-43BF-B7D5-F34D415553BD")]
[MediaDescriptor(ExtensionString = "jpg,jpeg,jpe,ico,gif,bmp,png")]
public class ImageFile : CommerceImage
{
    /// <summary>
    /// Gets or sets the description.
    /// </summary>
    public virtual String Description { get; set; }

    [Editable(false)]
    [ImageDescriptor(Width = 128, Height = 128)]
    public override Blob LargeThumbnail { get; set; }

    [Editable(false)]
    [ImageDescriptor(Width = 64, Height = 64)]
    public override Blob Thumbnail { get; set; }
}
```

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:

```
.Sleek .epi-thumbnailContentList.dgrid .dgrid-row .epi-thumbnail {
    width: 48px;
    height: 48px;
}
```

It might be interesting to look into how those Blobs work. For each Blob property of a `MediaData` content, CMS will create a file for it (as PNG for transparent). 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 orignal 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 examle, if the link to your asset is `/globalassets/catalogs/tops/sweaters/tops-sweaters-crew.jpg/`, then you can acces 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 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. `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 (which is supposed to have no custom content types), the asset mappings with entries and nodes will be skipped.

How versions work

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 a bit 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:

```
var parentLink = ContentReference.Parse("1073741845__CatalogContent");
var contentRepo = ServiceLocation.ServiceLocator.Current.GetInstance<IContentRep\
ository>();

//Unsaved content, should have status of NotCreated.
var variationContent = contentRepo.GetDefault<VariationContent>(parentLink);
variationContent.Name = "New variation";

//Save the content, now it is CheckoutOut
var variationLink = contentRepo.Save(variationContent, SaveAction.Save, AccessLe\
vel.NoAccess);

//A saved content is readonly. To edit it, we must create a "writable" clone
variationContent = contentRepo.Get<VariationContent>(variationLink)
    .CreateWritableClone<VariationContent>();
variationContent.Code = "New-varation";

```

```
//Check in, in the UI, it's Ready to Publish, which mean the edit was complete.
//The content status is now CheckedIn.
variationLink = contentRepo.Save(variationContent, SaveAction.CheckIn, AccessLev\
el.NoAccess);
variationContent = contentRepo.Get<VariationContent>(variationLink);

//Oops, made a typo. reject it.
variationContent = contentRepo.Get<VariationContent>(variationLink)
    .CreateWritableClone<VariationContent>();
//Now it's Rejected.
variationLink = contentRepo.Save(variationContent, SaveAction.Reject, AccessLeve\
l.NoAccess);

//Correct the mistake.
variationContent = contentRepo.Get<VariationContent>(variationLink).CreateWritab\
leClone<VariationContent>();
variationContent.Code = "New-variation";

//Publish it directly. Use ForceCurrentVersion flag so no new version will be cr\
eated
//it will overwrite the "rejected" version.
variationLink = contentRepo.Save(variationContent,
    SaveAction.Publish | SaveAction.ForceCurrentVersion, AccessLevel.NoAccess);
```

One thing to remember about the code above is that we used the versioned `ContentReference`:s (`WorkId > 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 "Previously Published" version in English:

```
var contentVersionRepository = ServiceLocator.Current.GetInstance<IContentVersio\
nRepository>();
var contentLink = ContentReference.Parse("83__CatalogContent");
var versions = contentVersionRepository.List(contentLink);
var previouslyPublished = versions.OrderByDescending(c => c.Saved)
        .FirstOrDefault(v => v.Status == VersionStatus.PreviouslyPublished && 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 Store[24] was the bottleneck. Firstly, 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 choosen because it looked quite straightforward. Each version (which was call `CatalogContentDraft`) is stored in one row, and except the "static" data which is supposed to be on all version (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.

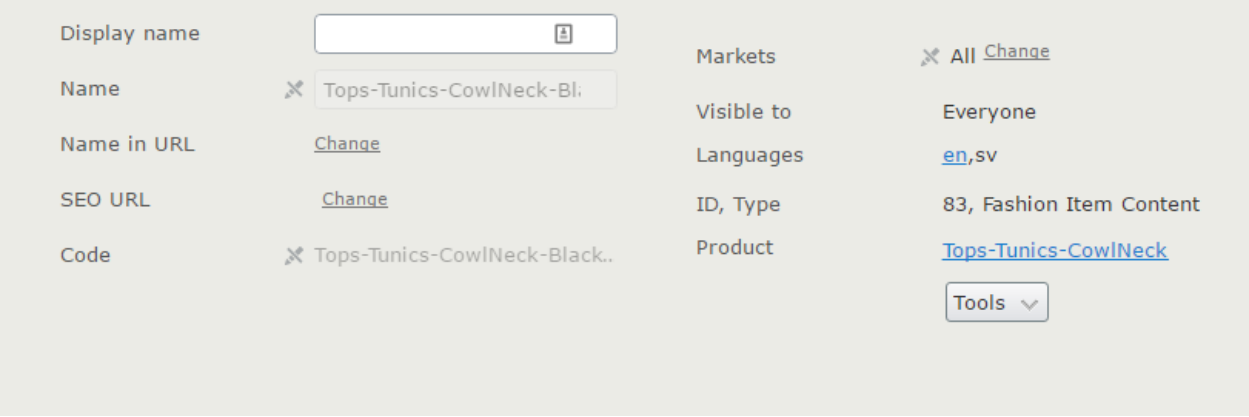
And having data in two places means you have to sync it everytime you save, which adds even more overheads to the system.

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

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:

[24] You 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.



Non CultureSpecific properties are grey out

By default, these properties are non-CultureSpecific: - `Code` - `Name` - `Markets` - `Assets` - `Prices`, `Inventories` are also uneditable 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:

```
var contentRepository = ServiceLocator.Current.GetInstance<IContentRepository>();
var contentLink = ContentReference.Parse("83__CatalogContent");
var content = contentRepository.Get<FashionItemContent>(contentLink, CultureInfo\
.GetCultureInfo("sv"))
    .CreateWritableClone<FashionItemContent>();
//Assuming Facet_Size is non CultureSpecific.
content.Facet_Size = content.Facet_Size + " edited";
contentRepository.Save(content, DataAccess.SaveAction.Publish, EPiServer.Securit\
y.AccessLevel.Publish);
```

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). This is, however, an extreme case and I don't expect you to do such action. Episerver Commerce team is aware of this issue and newer versions of Commerce might fix it.

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 rememeber: never mess up with ExistingLanguages until you absolutely know what you are trying to do. Episerver Commerce use ExistingLanguages internally to save versions in different languages, and if you change that it can lead to some unpredicted behaviors.

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 lower level than CatalogContentProvider, such as using ICatalogSystem directly, the update will also delete all other versions (only latest, published version is kept). This setting comes in handy when you don't not want to keep the old versions in the system, only latest, published one is needed. This is pretty much the same behavior with R3 where no versions existed. When you don't need versioning, this might be one way to improve performance.

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:

```
EPiServer.Configuration.Settings.Instance.UIMaxVersions = 1;
```

Or by adding `uiMaxVersions` (note it's not UIMaxVersions, the attributes in siteSettings are case-sensetive) 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:

```
var extendedAction = SaveAction.Publish.SetExtendedActionFlag(ExtendedSaveAction\
.ClearVersions);
```

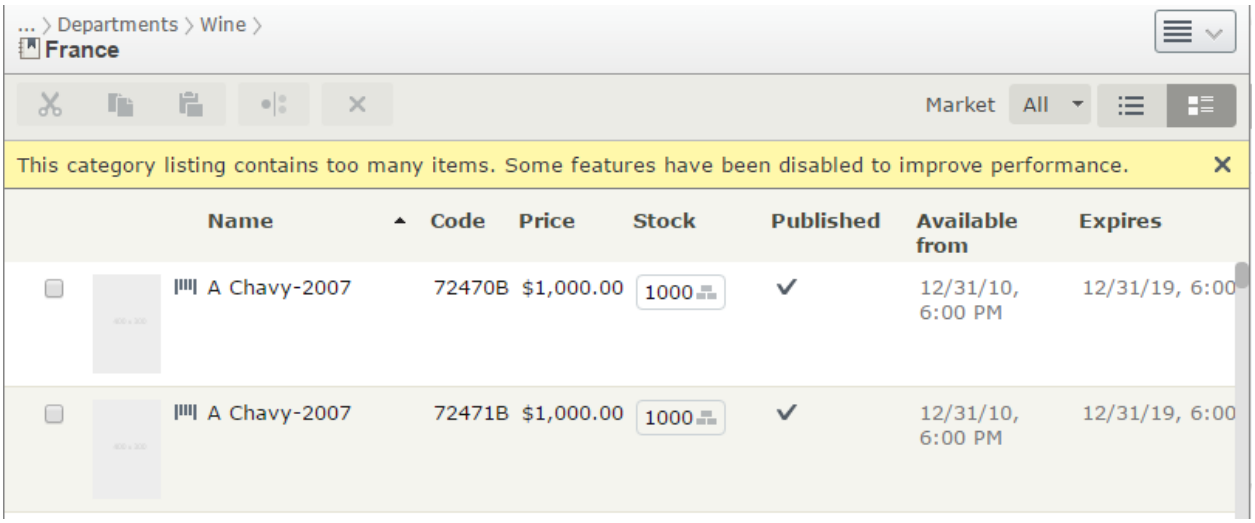
Note that compared to two previous options, this option is more flexible. You can apply it on certain catalog content which match specific criteria.

Optimizing your catalog

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.



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 having sub-category.

And it's not just CatalogUI. As we will learn later, 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.

So what's is the perfect number of entries per node? There is no definitive value, but I think you can keep it less than 500. Anything more than that and you should think about introducing sub-categories.

- Think about what metafields to put in a product, and what to put in its variations. It might be crystal clear to do so, but it's not always the case. Having multiple variations to share same

metafields will reduce the number of rows we have to load from CatalogContentProperty. 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.)

Part 1.1 Important concepts.

Now you have learned about catalog system - the heart of Episerver Commerce. If you knew about Commerce, you might be noticing some important concepts of the catalog were not mentioned. I want to keep them separated because they are not only used in the catalog, but affect entire the framework.

Chapter 3.1 Markets

Market is a relatively new concept in Episerver Commerce - it was included in Commerce R3a, but in hidden state. It was only fully implemented and visible later in Commerce 7.5. This is a new feature which affects entire Commerce site. So what's a market and why you should use it? Imagine you are a global company who sells products all over the world. Because of the regulations, taxes, logistics of regions, the availability of products might be different, and the prices might be different, too. To organize your site for such requirements, you would need to divide your areas of operation into Markets.

Market is an abstract concept, there is no constrain of what it can or can't be. Depends on your business, it can be a state, a country, a region, a continent, or you might not use Market at all (You have only one market). A Market is defined by `IMarket` interface:

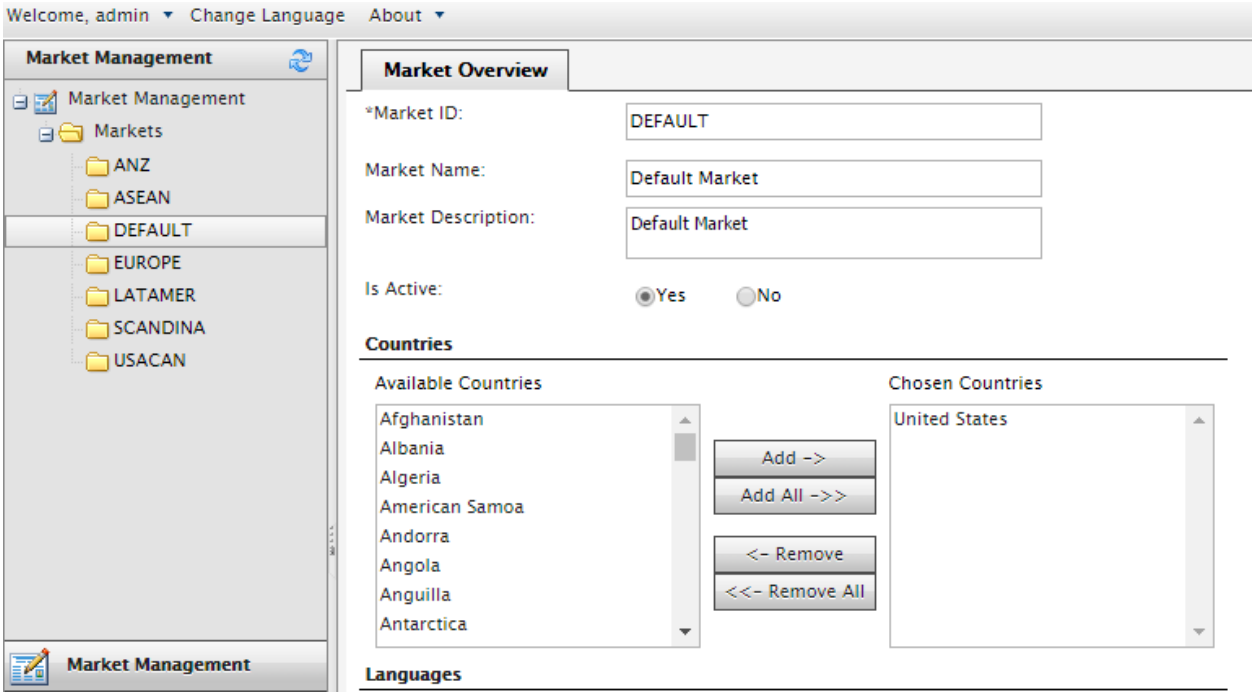
```
public interface IMarket
{
    /// <summary>
    /// Gets the market's unique identifier.
    /// </summary>
    MarketId MarketId { get; }

    /// <summary>
    /// True if the the market is enabled, false otherwise.
    /// </summary>
    bool IsEnabled { get; }

    /// <summary>
    /// Gets the market's name.
    /// </summary>
    string MarketName { get; }

    /// <summary>
    /// Gets the market's description.
    /// </summary>
    string MarketDescription { get; }

    /// <summary>
    /// Gets the default language for the market.
    /// </summary>
```

```
    CultureInfo DefaultLanguage { get; }

    /// <summary>
    /// Gets the available languages for the market.
    /// </summary>
    IEnumerable<CultureInfo> Languages { get; }

    /// <summary>
    /// Gets the default currency for the market.
    /// </summary>
    Currency DefaultCurrency { get; }

    /// <summary>
    /// Gets the available currencies for the market.
    /// </summary>
    IEnumerable<Currency> Currencies { get; }

    /// <summary>
    /// Gets the countries associated with the market.
    /// </summary>
    IEnumerable<string> Countries { get; }
}
```

The most important property of `IMarket` is `MarketId`, a simple `struct` which represents code of the market. For example, for United States market, you can have `MarketId` as "US", while for Europe market you can have "EUROPE". We will see later - how is this property used across Commerce system.

It's worth mentioning that there is a special, predefined `MarketId` in the default installation of Commerce: `MarketId.Default`, which the value of "DEFAULT". In practice, most of multi-markets Commerce websites do not use this market at all, it is usually disabled, but if you don't want to use the Market concept (your business is simply operated in a single market), just leave it there. There is effectively no way to work with Episerver Commerce without a market.

A market might have multiple `Languages` and `Currencies`, but it always has a `DefaultLanguage` and `DefaultConcurrency`. As mentioned earlier, a market is not bound to "Region", however, because it's the most common way to represent a market in real world, so `Countries` property was added for convenience.



Currently markets can be only edited in Commerce Manager. It's not the best UI, but it works

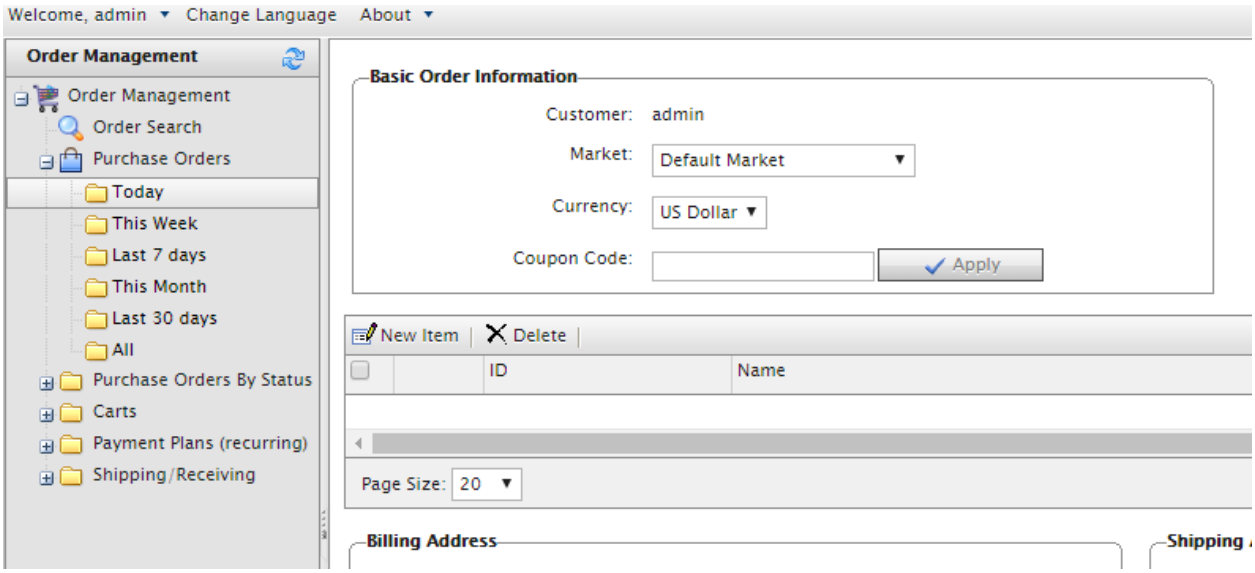
`Market` is the default implementation of `IMarket`, and it can be managed by `IMarketService` - it's not particularly interesting:

```
public interface IMarketService
{
    /// <summary>
    /// Gets all markets in the system.
    /// </summary>
    IEnumerable<IMarket> GetAllMarkets();

    /// <summary>
    /// Get a single market by MarketId.
    /// </summary>
    IMarket GetMarket(MarketId marketId);

    /// <summary>
    /// Adds a new market to the market system.
    /// </summary>
    void CreateMarket(IMarket market);

    /// <summary>
    /// Updates an existing market in the market system.
    /// </summary>
```

```
21      void UpdateMarket(IMarket market);
22
23      /// <summary>
24      /// Deletes a market by ID.
25      /// </summary>
26      void DeleteMarket(MarketId marketId);
27  }
```

So, enough of the interfaces. We'll see how markets are used. The market concept affects almost every aspect of Episerver Commerce.

- As from previous chapter, we've learned that an entry might or might not be available in a market. This is defined by a special metafield named `_ExcludedCatalogEntryMarkets`. This contains list of `MarketId` which the entry is *not* available in `_ExcludedCatalogEntryMarkets` metafield.
- Markets also control the prices: a price must belong to a market. Warehouses and inventories, however, are not tied to any markets (which are reasonable because a warehouse might serve more than 1 markets)
- Orders must belong to specific markets. Shipping methods and payment gateways also need to be available in a specific, however, they are still grouped by languages. This was because of backward compatibility reasons, and in the future versions shipping methods and payment gateways might be changed to be grouped by markets.



A market must always be set for an order

All market-specific information are determined by `MarketId`. It's important to determine which market a customer is currently in - this is handled by `ICurrentMarket`:

```
public interface ICurrentMarket
{
    /// <summary>
    /// Gets the current market.
    /// </summary>
    IMarket GetCurrentMarket();

    /// <summary>
    /// Sets the current market.
    /// </summary>
    void SetCurrentMarket(MarketId marketId);
}
```

The only method which needs to be implemented is `GetCurrentMarket` - you can simply "ignore" `SetCurrentMarket` - for example if you determine the market of customer based on the geography location. You can find a simple implementation of `ICurrentMarket` in both Commerce Sample site (`MarketStorage`) and QuickSilver site (`CurrentMarket`). Both use cookies to store the selected `MarketId`.

Chapter 3.2 Application-level settings

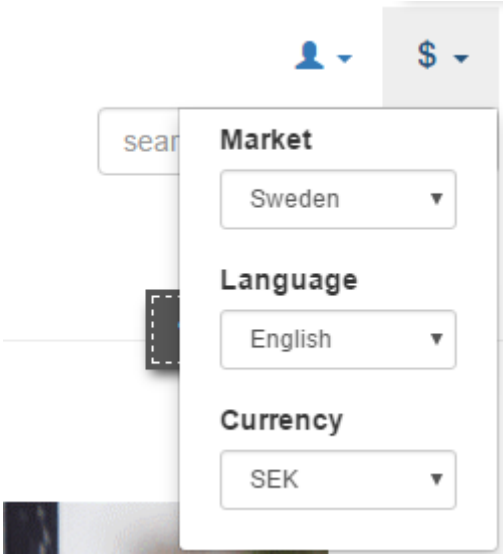
Beside `Market`, there are several other important setting which are shared across an Episerver Commerce site. Those settings are available in `SiteContext`, `IApplicationContext` and `AppContext` objects.

`SiteContext` and `AppContext` are both accessible in singleton objects: `SiteContext.Current` and `AppContext.Current`. `IApplicationContext` is registered via `ServiceLocator` as singleton but there is no property available.

> This is of course not optimal structure, but it comes with history reasons. `SiteContext` and `AppContext` have been there since Mediachase-era, and while `SiteContext` was intended to be used in site and `AppContext` was for both site and other applications (console, for example), the distinction has never been clear. `ApplicationContext` was introduced later in `EPiServer.Commerce` to handle content-related settings. There should only be one global-level settings object, however doing such thing will certainly result in breaking changes. We will have to live with these three objects, at least for now.

The most important setting in `SiteContext` is `CurrentCurrency`. If your site allows customers to select billing currency, this is the place to go

```
SiteContext.Current.CurrentCurrency = selectedCurrency;
```



Selecting currency, simple implementation in QuickSilver

There're some things to remember when you set the `CurrentCurrency` setting:

- This setting is stored by cookie. Other places only see the new `CurrentCurrency` in the next request. So it's required to refresh the page after setting the value.
- Prices are currency-dependant (We'll talk more about this in next part), so make sure to fresh prices of all items, *especially* items in the cart.

`AppContext` comes with several settings. The one you need to use the most is `ApplicationId`, a `Guid` to construct the `CatalogKey` which is used in many features of Episerver Commerce:

```
var catalogKey = new CatalogKey(AppContext.Current.ApplicationId, entryCode);
```

However, `ApplicationId` is going to be obsolete soon - which is planned for Commerce 11. It's a good thing - it has not been serving its true purpose, to allow multitenancy in Commerce. One less class to care about, in Commerce 11 you will be able to use `Code` of the entry directly. Another setting in `AppContext` is Azure-specific. `AzureCompatible` is a boolean flag to indicate if this site is `compatible` with Microsoft Windows Azure environment. It's not entirely true, however, it only shows if the metafield encryption is disabled or not in the database - because at this time of writing, SQL Azure Database still does not support encryption. Both `ApplicationId` and `AzureCompatible` are read-only, they're determined automatically by the state of the site.

`IApplicationContext` is probably the most important of all. It has two properties:

- `HasContentModelTypes` - indicates if the current site context contains the catalog content types, in other words - it separates the front-end site and anything else (Commerce Manager, ServiceAPI, unless you have your catalog content types in your ServiceAPI site). This setting is used by several parts of the system - for example, Find.Commerce, to see if the it should index the content or not. `HasContentModelTypes` is read-only and can't be set - it depends on the assemblies in the bin folder.
- `DisableVersionSync` - we've learned in previous chapter how it works and how can it be configured in `ecf.catalog.config`, the other way is via `IApplicationContext`.

The default implementation of `IApplicationContext` - `ApplicationContext` has another extra flag - `SkipContentModelCheck`. It allows the site to safely ignore the `HasContentModelTypes` flag - in other words, a workaround to set a readonly flag.

Part 2: Inventories and Prices

Chapter 4: Prices.

In Commerce R2 and earlier, prices were a part of entries. If you remember the structure of CatalogEntryDto in first chapter, you can still see the SalePrice table there. At first, it seems to be reasonable as the prices seems to be naturally belong to SKUs. However, this design has a serious flaw, it makes it harder to integrate to the system, such as ERP, to update prices regularly and independently (Updating entries will fire certain events, which we will mention later).

Pricing

Add Item

Edit Command	Market	Sale Type	Sale Code	Currency	Price	Min. Quantity	Start Date	End Date
	Australia and New Zealand	All Customers		AUD	1042.00	0	9/1/2010 3:00:00 PM	
	ASEAN	All Customers		USD	1000.00	0	9/1/2010 3:00:00 PM	
	Default Market	All Customers		USD	1000.00	0	9/1/2010 3:00:00 PM	
	Europe	All Customers		EUR	775.20	0	9/1/2010 3:00:00 PM	
	Central and South America	All Customers		USD	1000.00	0	9/1/2010 3:00:00 PM	
	Scandinavia	All Customers		EUR	775.20	0	9/1/2010 3:00:00 PM	
	USA and Canada	All Customers		USD	1000.00	0	9/1/2010 3:00:00 PM	

1 Page 1 of 1 (7 ite

Price editing in Commerce Manager

Commerce R3 came with a new design - the prices are now separated and no longer tightly coupled with the entries. Of course, they still tie to the entries in certain way - by matching codes.[25]

The new design also came with two interfaces - `IPriceValue` and `IPriceService`. The former, as its name might suggest, is for the price value itself, and the latter provides actual methods to work with those values.

Let's see what an `IPriceValue` has

```
public interface IPriceValue
{
    /// <summary>
    /// Gets the catalog entry key.
    /// </summary>
    CatalogKey CatalogKey { get; }

    /// <summary>
    /// Gets the market.
```

[25] There are three ways to identify an entry (or a node): An Id (`int`), a code (`string`) or an unique identifier (`Guid`, introduced in Commerce 7.10). An Id is used internally in the catalog, and a `Guid` is used to communicate with Episerver CMS (if you have worked with Episerver CMS, you might know the concept of *permanent link*), and a code is used to work with external, such as ERP or PIM. Almost every feature which might work with external system can be identified by a code.

```
    /// </summary>
    MarketId MarketId { get; }

    /// <summary>
    /// The customer pricing.
    /// </summary>
    CustomerPricing CustomerPricing { get; }

    /// <summary>
    /// The date and time in UTC when this price data became or becomes active.
    /// </summary>
    DateTime ValidFrom { get; }

    /// <summary>
    /// Gets the valid until date and time in UTC. Null means no end date.
    /// </summary>
    DateTime? ValidUntil { get; }

    /// <summary>
    /// The minimum quantity that this pricing applies to.
    /// For example, if MinQuantity = 4 then customer must by at least 4 to
    /// get the UnitPrice.
    /// </summary>
    decimal MinQuantity { get; }

    /// <summary>
    /// The price per unit of quantity.
    /// </summary>
    Money UnitPrice { get; }
}
```

Almost all the properties explain for themselves, however, there are two things which might be interesting here: CatalogKey and CustomerPricing.

As we mentioned earlier, the code is the way to identify an entry when you work with external systems. CatalogKey is a little more than that - it has one ApplicationId [26] as GUID and a code as string. You can construct a `CatalogKey` by this:

[26] Why `ApplicationId`? In the early days of Episerver Commerce, there was an idea of multitenant in just one Commerce database, meaning you can have multiple "Applications". Each application will be identified by an `ApplicationId`, which why if you look at most important tables in Commerce database, such as Catalog, CatalogEntry or CatalogNode, you will see that column there. However, the idea has never been fully implemented, and it has never really worked. It's safe to always to AppContext.Current.ApplicationId. The idea of multitenant might be removed in Commerce 10, which will be a big breaking change. Just don't take my words for it.

```
var catalogKey = new CatalogKey(AppContext.Current.ApplicationId, entryCode);
```

`CatalogKey` is an important little class and is used intensively in price and inventory system.

The second interesting thing is `CustomerPricing`. What is it for?

You might have different prices for markets or currencies for an Entry, but you might also want to have different prices for different customers. CustomerPricing allows you to set such prices. It's a simple class with just two properties.

```
public class CustomerPricing : IEquatable<CustomerPricing>
{
    public static readonly CustomerPricing AllCustomers = new CustomerPricing(Pr\
iceType.AllCustomers, string.Empty);

    public enum PriceType
    {
        /// <summary>
        /// Gets prices valid for all customers. PriceCode is supposed be an emp\
ty string when AllCustomers is specified.
        /// </summary>
        AllCustomers = 0,

        /// <summary>
        /// Gets prices valid for a specific customer. PriceCode is supposed be \
the customer's username.
        /// </summary>
        UserName = 1,

        /// <summary>
        /// Gets prices valid for a specific price group. PriceCode is supposed \
be the price group identifier.
        /// </summary>
        PriceGroup = 2
    }

    /// <summary>
    /// Gets the price type id.
    /// </summary>
    public PriceType PriceTypeId { get; private set; }

    /// <summary>
    /// The price code. The meaning of this depends on the value of PriceTypeId.
```

```
    /// </summary>
    public string PriceCode { get; private set; }
}
```

You can have either price for `AllCustomer` (which is the default value), for a specific customer (`PriceType` = UserName and `PriceCode` is the username), or for a specific group (`PriceType` = `PriceGroup` and `PriceCode` is the name of that group, usually a name of an organization).

There is one important characteristic of the default implementation of `IPriceService` - it's optimized. The prices saved in the system might not be the price you input. How? It suppose to make the best price wins. How?

Assuming you have those prices, all have the same market, min quantity and currency:

- Unit Price 100$, Valid From: 1/1/2016 to 12/31/2016
- Unit Price 80$, Valid From: 4/1/2016 to 6/30/2016
- Unit Price 60$, Valid Form: 5/1/2016 to 5/31/2016

Input those price in the price editing tab in Commerce Manager. Guess what will you get back after you save the Entry (which will trigger the save for the prices)?

It might be surprising for you to find out, those are the results you get:

- Unit Price 100$, Valid From: 1/1/2016 to 3/31/2016
- Unit Price 80$, Valid From: 4/1/2016 to 4/30/2016
- Unit Price 60$, Valid Form: 5/1/2016 to 5/31/2016
- Unit Price 80$, Valid From: 6/1/2016 to 6/30/2016
- Unit Price 100$, Valid From: 1/1/2016 to 3/31/2016

So if there are overlapped prices, the lowest price will win in that certain period. This applies to the Min Quantity, too. The lower price with lower Min Quantity will override the higher price with higher Min Quantity. In short, the most beneficial prices for customers will be the final prices.

Well, that's the behavior of the default implementation. We'll look into implementing another price service with act differently in later chapters.

Another important thing about `IPriceService`: it is optimized for batch processing. It works really well when you want to load the price to display to customers, but it comes with a caveat. When you save, you actually save the entire prices of an SKU, you can't just save exactly one price, because there is no way to identify it. It makes editing exactly one price less than optimized.

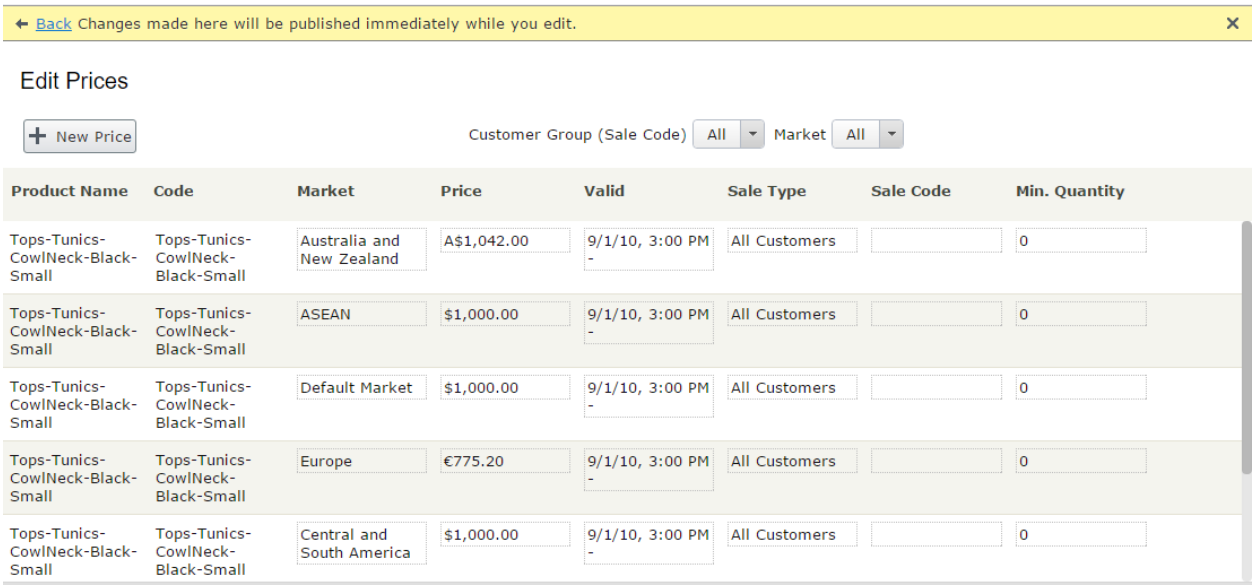
Let's see how IPriceService has to offer:

```
public interface IPriceService
{
    /// <summary>
    /// Gets a value indicating if this implementation can modify prices.
    /// </summary>
    bool IsReadOnly { get; }

    /// <summary>
    /// Gets a default price (price with min quantity = 0 and is valid at the gi\
ven time) of given CatalogKey in given market and currency.
    /// </summary>
    IPriceValue GetDefaultPrice(MarketId market, DateTime validOn, CatalogKey ca\
talogKey, Currency currency);

    /// <summary>
    /// Gets filtered pricing for a single catalog entry.
    /// </summary>
    IEnumerable<IPriceValue> GetPrices(MarketId market, DateTime validOn,
        CatalogKey catalogKey, PriceFilter filter);

    /// <summary>
    /// Gets filtered pricing for multiple catalog entries.
    /// </summary>
    IEnumerable<IPriceValue> GetPrices(MarketId market, DateTime validOn,
        IEnumerable<CatalogKey> catalogKeys, PriceFilter filter);

    /// <summary>
    /// Gets filtered pricing for multiple catalog entries, in varying quantitie\
s.
    /// </summary>
    IEnumerable<IPriceValue> GetPrices(MarketId market, DateTime validOn,
        IEnumerable<CatalogKeyAndQuantity> catalogKeysAndQuantities, PriceFilter\
 filter);

    /// <summary>
    /// Gets all price values for a single catalog entry.
    /// </summary>
    IEnumerable<IPriceValue> GetCatalogEntryPrices(CatalogKey catalogKey);

    /// <summary>
    /// Gets all price values for one or more catalog entries.
    /// </summary>
```

```
    IEnumerable<IPriceValue> GetCatalogEntryPrices(IEnumerable<CatalogKey> catal\
ogKeys);

    /// <summary>
    /// Sets the price values of a catalog entry.
    /// </summary>
    void SetCatalogEntryPrices(CatalogKey catalogKey,
        IEnumerable<IPriceValue> priceValues);

    /// <summary>
    /// Sets the price values for multiple catalog entries.
    /// If a catalog entry is listed in catalogKeys but there is no prices
    /// in priceValues, it prices will be deleted.
    /// </summary>
    void SetCatalogEntryPrices(IEnumerable<CatalogKey> catalogKeys,
        IEnumerable<IPriceValue> priceValues);

    /// <summary>
    /// Sets the price values for catalog entries to reflect changes
    /// in IPriceDetailService.
    /// This should ONLY be used by implementations of IPriceDetailService.
    /// </summary>
    void ReplicatePriceDetailChanges(IEnumerable<CatalogKey> catalogKeys,
        IEnumerable<IPriceValue> priceValues);
}
```

In my opinion - this interface is too “thick”. The interface should have “minimal viable” signature, and other methods which can be extension methods instead. Bigger interfaces take more time to implement. Sadly, this trend is across the price and inventory systems, where many interfaces contain way more methods than they should.

When Episerver Commerce 7.5 was being developed, there was a requirement to edit price inside the new Catalog UI. Prices need to be edited individually. To solve that problem, `IPriceDetailValue` and `IPriceDetailService` were introduced.



Price Editing in Catalog UI

There are two price views in Catalog UI [27]. The first one shows the price values of a variant, which is equivalent with the price editing tab in Commerce, but readonly. The second one actually allows you to edit the prices, use `IPriceDetailValue` and `IPriceDetailService`.

What is the difference?

Well, not much.

```
public interface IPriceDetailValue : IPriceValue
{
    long PriceValueId { get; set; }
}
```

Yay, `IPriceDetailValue` is just `IPriceValue`, with an Id. This will allow you to identify a specific price and edit or delete it individually, without saving the entire set price of an Entry. Let's see what `IPriceDetailService` provides:

[27] Strictly speaking, you have totally 3 price views in Catalog UI. The third one is the view for entire prices in a Category (Node), or a Product: it will list prices of all variants in that Category or Product.

```
public interface IPriceDetailService
{
    /// <summary>
    /// Gets a value indicating if this implementation can modify prices.
    /// </summary>
    bool IsReadOnly { get; }

    /// <summary>
    /// Gets a specific price value.
    /// </summary>
    IPriceDetailValue Get(long priceValueId);

    /// <summary>
    /// Gets all price details of a catalog content.
    /// </summary>
    IList<IPriceDetailValue> List(ContentReference catalogContentReference);

    /// <summary>
    /// Gets all price details for a catalog content with paging support.
    /// </summary>
    IList<IPriceDetailValue> List(ContentReference catalogContentReference,
        int offset, int count, out int totalCount);

    /// <summary>
    /// Gets all price details for a catalog content with paging
    /// support and filter for market, currencies and CustomerPricing:s.
    /// </summary>
    IList<IPriceDetailValue> List(ContentReference catalogContentReference,
        MarketId marketId, PriceFilter filter, int offset, int count, out int to\
talCount);

    /// <summary>
    /// Saves the specified price values.
    /// </summary>
    IList<IPriceDetailValue> Save(IEnumerable<IPriceDetailValue> priceValues);

    /// <summary>
    /// Deletes the specified price values.
    /// </summary>
    void Delete(IEnumerable<long> priceValueIds);

    /// <summary>
```

```
    /// Sets the price detail values to reflect changes in IPriceService.
    /// This should ONLY be used by implementations of IPriceservice>.
    /// </summary>
    void ReplicatePriceServiceChanges(IEnumerable<CatalogKey> catalogKeySet,
        IEnumerable<IPriceValue> priceValuesList);
}
```

There's one thing to remember about the `List` method. If the `catalogContentReference` is a variant or a package, it will return the its prices - as usual. However if it is a product or bundle, the combined prices of all children variants will be returned. This is also true if it is a category, but only prices of direct children entries(no recursive).

There are three important characteristics of the default implementation of `IPriceDetailService`:

- First, it saves whatever you tell it to save. So no optimization at all, you'll get what you give. As-is.
- Second, it does not cache. For a system which is designed for editing, it's less than effective to have the cache floating around when you won't hit it the next time.
- And last one, it does sync with `IPriceService`. So if you save prices by `IPriceDetailService`, it saves the values as-is, but it also synchronizes the changes to `IPriceService`. Of course, next time you load prices by `IPriceService`, it will still return optimized prices.

So there are some rules of thumbs when you're working with prices:

- If you are loading prices to do calculation or to show to customers, use `IPriceService`.
- If you are saving prices, use `IPriceDetailService`.

Those are not absolute rules, nothing prevent you to use `IPriceService` to save prices, but it'll be less effective.

You can argue that the two price system can cause confusion, and I can agree to some extend - it might be not the best design. The argument is that was for better performance. For other interfaces, such as for inventories which we will exam shortly, the updates usually follow the reads, but for pricing, the updates are much less frequent than the reads. But still, the two intefaces are confusing. Perhaps the better design would have been following CQRS - Command Query Responsibility Segregation pattern more strictly, and make `IPriceLoadingService` and `IPriceSavingService` instead.

Enough of the theory, now let's get our hands dirty with a simple example: How to store the MSRP (Manufacturer suggested retail price), usually known as list price for SKU. This price is unlikely to change (it'll be changed, like 1 or 2 times in the entire lifetime of the product). If you remember from chapter 1 - Catalog system, there was a Display price attached to each variation. However, this is

not enough, because a product can be sold in multiple markets and for each market, there should be a list price. This is a pretty frequently asked feature from Episerver World forums, and we're going to build it today.

How to store the MSRP can be a little tricky - the price system - as well as many parts in the system - is extentable in a API:s level, not in database level. So if we want to store a special price, we need a small hack - use PriceGroup for what it was *not* intended for. We will store those prices with `PriceType = 2` and `PriceGroup = "MSRP"`.

To make the new prices easy to access, we will try not to implement the interface ourselves, but instead, use a very useful feature in C# - extensions methods. Let's start with saving the price:

```
    public static void SaveMsrp(this IPriceDetailService priceDetailService, Con\
tentReference contentLink, MarketId marketId, Currency currency, decimal amount)
    {
        int totalCount;
        var priceFilter = new PriceFilter();
        priceFilter.Currencies = new List<Currency>() { currency };
        var msrp = (PriceDetailValue)priceDetailService.List(contentLink, market\
Id, priceFilter, 0, int.MaxValue, out totalCount).
            FirstOrDefault(p => p.UnitPrice.Currency == currency && p.CustomerPr\
icing.PriceTypeId == CustomerPricing.PriceType.PriceGroup && p.CustomerPricing.P\
riceCode == "MSRP");

        if (msrp != null)
        {
            msrp.UnitPrice = new Money(amount, currency);
        }
        else
        {
            msrp = new PriceDetailValue();
            var contentRepository = ServiceLocator.Current.GetInstance<IContentR\
epository>();
            var entryContent = contentRepository.Get<EntryContentBase>(contentLi\
nk);
            msrp.CatalogKey = new CatalogKey(new Guid(entryContent.ApplicationId\
), entryContent.Code);
            msrp.CustomerPricing = new CustomerPricing(CustomerPricing.PriceType\
.PriceGroup, "MSRP");
            msrp.UnitPrice = new Money(amount, currency);
            msrp.MarketId = marketId;
            msrp.MinQuantity = 0;
            msrp.ValidFrom = entryContent.StartPublish.Value;
        }
```

```

        priceDetailService.Save(msrp);
    }
```

The code is pretty straighforward - we try to load the existing MSRP, if any, otherwise we create one. Perhaps the most notable part is when we set the `ValidFrom`. This field is required to be not null, and the value should be the time when the entry becomes available. However with the abstraction of `entryContent`, StartPublish can be null. In this case we use the current time instead.

We needed to convert the existing price to PriceDetailValue because the values returned by `IPriceDetailService.List` are `IEnumerable<IPriceDetailService>`, and `IPriceDetailService` is read-only. And then we can extract that long statement to an extention method on its own - to load the MSRP:

```
    public static IPriceDetailValue LoadMsrp(this IPriceDetailService priceDetai\
lService, ContentReference contentLink, MarketId marketId, Currency currency)
    {
        int totalCount;
        var priceFilter = new PriceFilter();
        priceFilter.Currencies = new List<Currency>() { currency };
        return priceDetailService.List(contentLink, marketId, priceFilter, 0, in\
t.MaxValue, out totalCount).
            FirstOrDefault(p => p.UnitPrice.Currency == currency && p.CustomerPr\
icing.PriceTypeId == CustomerPricing.PriceType.PriceGroup && p.CustomerPricing.P\
riceCode == "MSRP");
    }
```

And then the SaveMrsp become a little shorter (and cleaner):

```
    public static void SaveMsrp(this IPriceDetailService priceDetailService, Con\
tentReference contentLink, MarketId marketId, Currency currency, decimal amount)
    {
        var msrp = (PriceDetailValue)LoadMsrp(priceDetailService, contentLink, m\
arketId, currency);

        if (msrp != null)
        {
            msrp.UnitPrice = new Money(amount, currency);
        }
        else
        {
            msrp = new PriceDetailValue();
```

```
            var contentRepository = ServiceLocator.Current.GetInstance<IContentR\
epository>();
            var entryContent = contentRepository.Get<EntryContentBase>(contentLi\
nk);
            msrp.CatalogKey = new CatalogKey(new Guid(entryContent.ApplicationId\
), entryContent.Code);
            msrp.CustomerPricing = new CustomerPricing(CustomerPricing.PriceType\
.PriceGroup, "MSRP");
            msrp.UnitPrice = new Money(amount, currency);
            msrp.MarketId = marketId;
            msrp.MinQuantity = 0;
            msrp.ValidFrom = entryContent.StartPublish.Value;
        }

        priceDetailService.Save(msrp);
    }
```

However, this is not entirely optimal. As we talked earlier - we should use `IPriceDetailService` for editing and `IPriceService` for loading. We did with editing, but we should implement an extension method for loading:

```
    public static IPriceValue LoadMsrp(this IPriceService priceService, ContentR\
eference contentLink, MarketId marketId, Currency currency)
    {
        var contentRepository = ServiceLocator.Current.GetInstance<IContentRepos\
itory>();
        var entryContent = contentRepository.Get<EntryContentBase>(contentLink);
        var catalogKey = new CatalogKey(new Guid(entryContent.ApplicationId), en\
tryContent.Code);
        var priceFilter = new PriceFilter();
        priceFilter.Currencies = new List<Currency>() { currency };
        return priceService.GetPrices(marketId, DateTime.UtcNow, catalogKey, pri\
ceFilter)
            .FirstOrDefault(p => p.CustomerPricing.PriceTypeId == CustomerPricin\
g.PriceType.PriceGroup && p.CustomerPricing.PriceCode == "MSRP");
    }
```

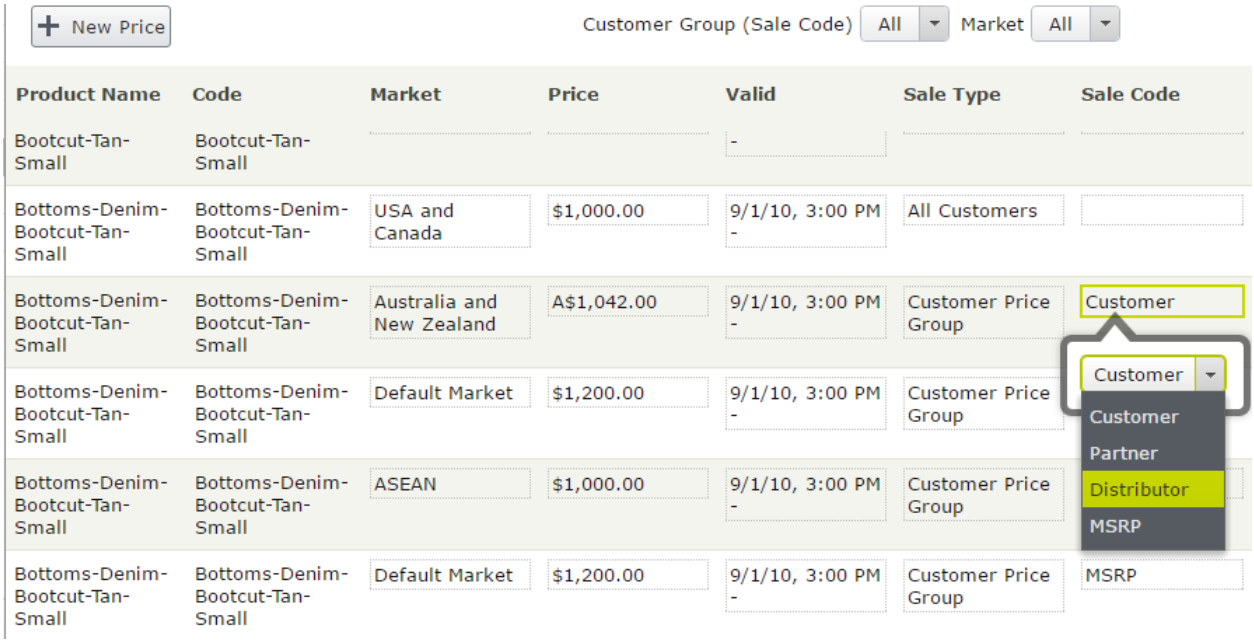
Of course, there are still improvements can be made - such as making “MSRP” a constant that shared across the methods, or you can pass an `EntryContentBase` parameter instead, so you won’t have to load it up just for the code. I’ll leave that for you. This approach comes with a limitation - you won’t be able to edit a list price Catalog UI, because the price editor there only allows three predefined `PriceGroups` (Customer, Partner and Distributor) and we can’t change it back in the Catalog UI.

Product Name	Code	Market	Price	Valid	Sale Type	Sale Code
Bootcut-Tan-Small	Bootcut-Tan-Small			-		
Bottoms-Denim-Bootcut-Tan-Small	Bottoms-Denim-Bootcut-Tan-Small	Europe	€775.20	9/1/10, 3:00 PM -	All Customers	
Bottoms-Denim-Bootcut-Tan-Small	Bottoms-Denim-Bootcut-Tan-Small	Central and South America	$1,000.00	9/1/10, 3:00 PM -	All Customers	
Bottoms-Denim-Bootcut-Tan-Small	Bottoms-Denim-Bootcut-Tan-Small	Scandinavia	€775.20	9/1/10, 3:00 PM -	All Customers	
Bottoms-Denim-Bootcut-Tan-Small	Bottoms-Denim-Bootcut-Tan-Small	USA and Canada	$1,000.00	9/1/10, 3:00 PM -	All Customers	
Bottoms-Denim-Bootcut-Tan-Small	Bottoms-Denim-Bootcut-Tan-Small	Default Market	$1,200.00	9/1/10, 3:00 PM -	Customer Price Group	MSRP

The price is here, just remember to not touch it!

Can we make it editable even with the new custom MSRP type? In this case, fortunately, yes. `PriceGroup` types are just another `MetaTypes`, and they can be registered. We can have this code to one of the initialization module, the code will check if the MSRP type existed or not, then add it (to the end of the list) if necessary

```
    private void AddMSRPGroup()
    {
        var metaFieldType = DataContext.Current.MetaModel.RegisteredTypes["Conta\
ctGroup"];
        var metaEnumItems = MetaEnum.GetItems(metaFieldType);
        var msrpGroup = metaEnumItems.Any(item => string.Equals(item.Name, "MSRP\
", StringComparison.OrdinalIgnoreCase));
        if (!msrpGroup)
        {
            var lastIndex = metaEnumItems.Count();
            MetaEnum.AddItem(metaFieldType, "MSRP", ++lastIndex);
        }
    }
```



Hooray, the price type is here

Now let's consider another example of pricing. Prices can be changed quite frequently - on a supply and deman basis, or because the items are near EOL (End of Line), or because simply you get a better deal from suppliers. In such case, it's not technically promotions, but you still want to show the discounted price to the customer. It's pretty strong encourage to say "Was $1200, only $1000" now. This is a pretty way to make customers decide to buy.

```
    public static IPriceValue GetPreviousPrice(this IPriceService priceService, \
ContentReference contentLink, MarketId marketId, Currency currency)
    {
        var contentRepository = ServiceLocator.Current.GetInstance<IContentRepos\
itory>();
        var entryContent = contentRepository.Get<EntryContentBase>(contentLink);
        var catalogKey = new CatalogKey(new Guid(entryContent.ApplicationId), en\
tryContent.Code);
        priceFilter.Currencies = new List<Currency>() { currency };
        var prices = priceService.GetCatalogEntryPrices(catalogKey);
        var previousPrice = prices.FirstOrDefault(p => p.MarketId == marketId
            && p.UnitPrice.Currency == currency && p.ValidUntil.HasValue && p.Va\
lidUntil.Value < DateTime.UtcNow);
        var currentPrice = prices.FirstOrDefault(p => p.MarketId == marketId
            && p.UnitPrice.Currency == currency && p.ValidFrom < DateTime.UtcNow\
 && (!p.ValidUntil.HasValue || p.ValidUntil.Value > DateTime.UtcNow));
        if (previousPrice.UnitPrice.Amount < currentPrice.UnitPrice.Amount)
        {
```

```
            return previousPrice;
        }
        return currentPrice;
    }
```

The last line is a small but important thing to check - if the price was increased, it's better to silently ignore the difference. We only show information if the item is cheaper.

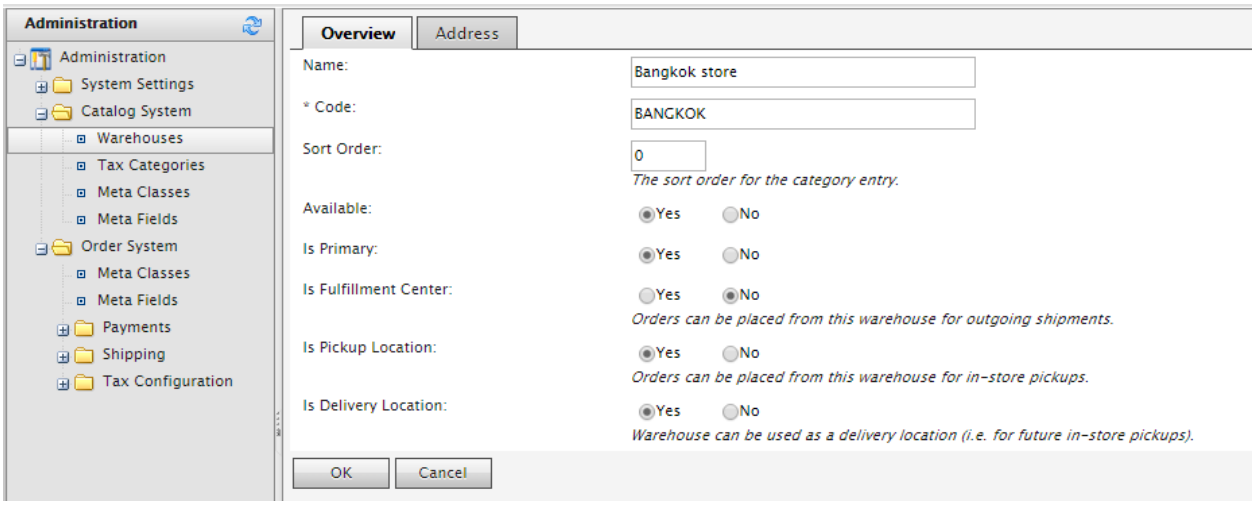
Chapter 5: Inventories

Inventory is another "detachable" system in Episerver Commerce, and yet as important. In real world scenario, Inventories are usually updated from external systems, such as ERP. So they were designed with "interface" approach in mind - they are easy to decouple from the system and be replaced by your own implementation.

Let's examine two concepts: warehouse and inventory.

Warehouse

Warehouse is the place you store the goods, it might be used as a distribution center, or a pickup location.



Editing a warehouse in Commerce Manager

Same as entries, the code of warehouses are supposed to be unique, and it's used to map between warehouse and inventories. You'll work with `Warehouse` via `IWarehouse` and `IWarehouse-Repository`. There is nothing special about these interfaces - you can check the documentation at http://world.episerver.com/documentation/Class-library/?documentId=commerce/9/8BCAE8CB and http://world.episerver.com/documentation/Class-library/?documentId=commerce/9/4A181B5. Normally, you we work with `Mediachase.Commerce.Inventory.Warehouse`, the default implementation of `IWarhouse`.

Inventory

While you will hardly work with warehouse API:s, you are supposed to work with inventories, a lot. Unlike warehouses which are rarely changed, the inventories are adjusted continously - when a customer places and order, when a supply comes to a warehouse, etc.

There are again - two API:s for manipulating inventories in Episerver Commerce. The first - and older one is `IWarehouseInventoryService`, provides methods to work with `IWarehouseInventory`. The newer one is `IInventoryService`, provides method around `InventoryRecord`. You might ask, why.

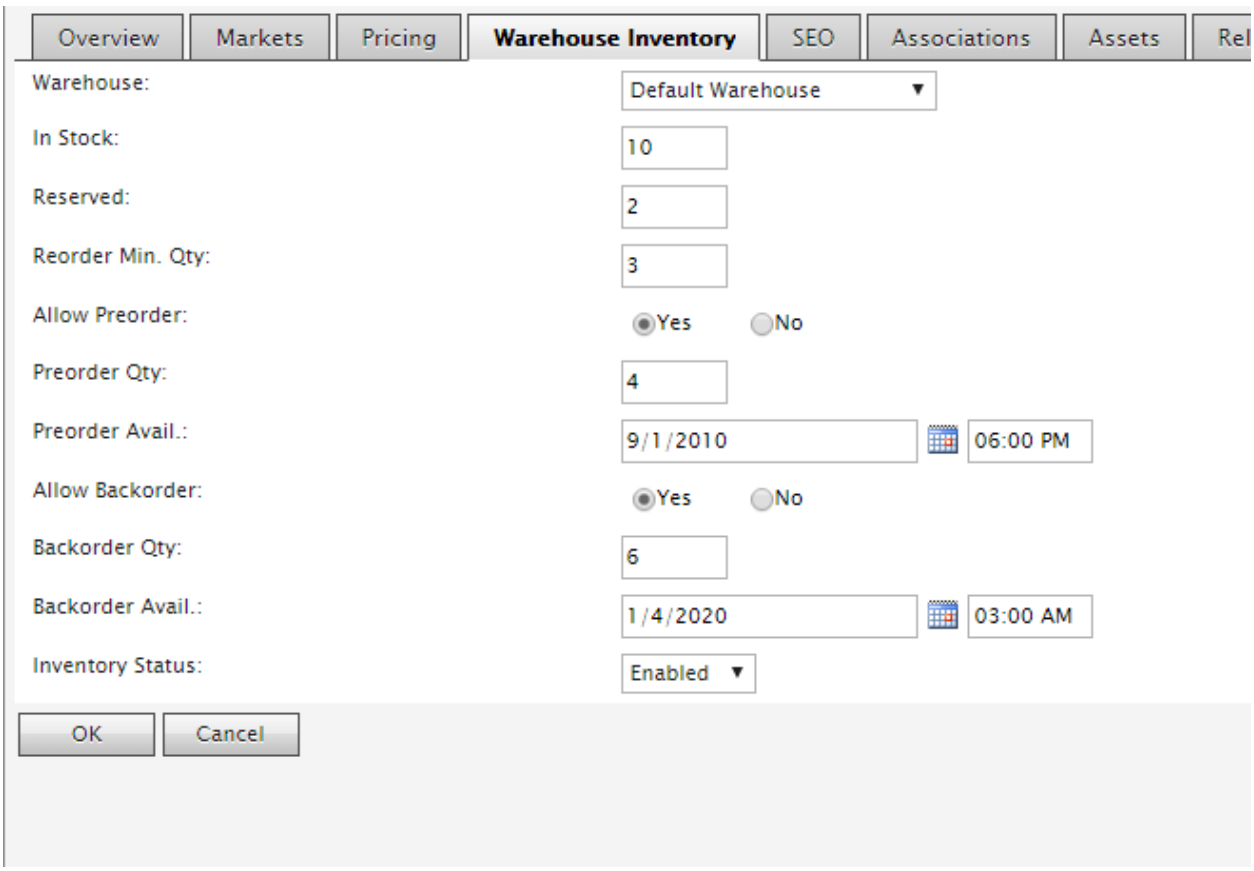
`IWarehouseInventoryService` has been there since Commerce 7.5. Its concept is simple: if a SKU is stored in a warehouse, a record of `IWarehouseInventory` will be saved for it. The record itself is quite self-explainary:

In Commerce 9.24, (which is under development at the time of this writing), `IWarehouseInventory` and `IWarehouseInventoryService` will be obsoleted. However they are not going to "disappear" anytime soon. Episerver Commerce has a policy of supporting obsoleted APIs for at least 12 months since the version the `[Obsolete]` attribute is added. This section will be kept in this book for reference - but you are recommended to switch to new inventory API:s.

```
public interface IWarehouseInventory
    {
        CatalogKey CatalogKey { get; }

        string WarehouseCode { get; }

        decimal InStockQuantity { get; }

        decimal ReservedQuantity { get; }

        decimal ReorderMinQuantity { get; }

        decimal PreorderQuantity { get; }

        decimal BackorderQuantity { get; }

        bool AllowBackorder { get; }

        bool AllowPreorder { get; }

```

```
        InventoryTrackingStatus InventoryStatus { get; }

        DateTime? PreorderAvailabilityDate { get; }

        DateTime? BackorderAvailabilityDate { get; }
    }
```

This maps to the Warehouse Inventory tab in Commerce Manager



This might not be the best UI in the world, but it works

However it's not editable in Catalog UI:

Tracked

Inventory locations

Location	Quantity	Reserved	Allow Preorder	Preorder Availability	Allow Backorder	Inventory Status
BANGKOK	10	0	⊘	1/1/00, 1:00 AM	⊘	Enabled
BEIJING	22	0	⊘	1/1/00, 1:00 AM	⊘	Enabled
BERLIN	15	0	⊘	1/1/00, 1:00 AM	⊘	Enabled
default	10	2	✓	9/1/10, 6:00 PM	✓	Enabled

This is read-only

At this time of writing, a new UI for editing inventories in Catalog UI is being worked on. It should be released in Commerce 9.24

If you are new to Commerce, perhaps it's worth spending sometimes to explain the terminologies with the inventory:

- `InStockQuantity` is the quantity you have in stock - it's the amount of goods that your warehouse can fulfill, minus `ReservedQuantity`. For example, if you have 100 items in store, but the reserved level set to 5, you can only sell 95 items.
- `ReorderMinQuantity` is the minimum quantity for re-order. Once you your instock quantity reaches this level, the entry will be shown in the "Low stock quantity" report.
- `AllowBackorder` is a flag indicating if you allow customers to order this item, even if it's currently out of stock, if the `BackorderAvailabilityDate`, if set, is passed. The order will be fulfilled later when the goods arrive. The number of items ordered by back-order will be added up to `BackorderQuantity`.
- `AllowPreorder` is a flag indicating if you allow customers to order this item before the day it's available, if the `PreorderAvailabilityDate`, if set, is passed. For example, if you set `AllowPreorder` to true, then your customer still to wait until `PreorderAvailabilityDate` to pre-order. The order will be fulfilled later - the number of items ordered by pre-order will be added up to `PreorderQuantity`.
- `InventoryTrackingStatus` is a flag indicating if the workflows - should care about the inventory at all. If this is set to `Disabled`, then the workflows won't check if there is any item left in store when a customer adds it to cart, or won't adjust the inventory when they place the order. If you are selling digital goods - like this book itself - it makes sense to set it to `Disabled`.

Then how the "old" inventory system work?

If you use a vanilla Episerver Commerce system - meaning you don't have any customizations to the workflows, then usually you work around `IWarehouseInventoryService` - a pretty big interface

which provides CRUD operations to `IWarehouseInventory`, as you can find its full documention [^here](http://world.episerver.com/documentation/Class-library/?documentId=commerce/9/56465654).

- You can use `IWarehouseInventoryService.List` overloads to load the information for entries shown to customers. Depends on your website's policies, you might want to show something like "7 available", "Less than 5 left", etc. However, it's generally a good practice to at least indicate if an item is out of stock or not, and if it should be refilled soon.
- When a customer add an item to cart, you'll normally call `CartValidate` workflow to do some checks (applying promotions etc), and the system will verify if the inventories have enough quantity for the goods. In cases when the quantity can't be fulfilled, the quantity might be reduced, or the item even removed from the cart. This is done in `CheckInventoryActivity`.

These steps are not changed in the new system. However the next step was.

- When the customer finally places an order, the value of `InStockQuantity` (and in some cases, `PreorderQuantity`, if it's a pre-order, or `BackorderQuantity`/a combination of `InStockQuantity` and `BackorderQuantity`) will be updated to subtract (or add, in case the warehouse is refilled.) the quantity has been ordered.

This design works to a point. It serves basic cases quite well, however, it has some major limitations which led to the new inventory system:

- The quantities are set to a calculated value, such as if it was 10 and a customer places an order of 2, then it's set to 8. Normally this is non-issue, but if your item is being sold like hot cake (and of course, you would want it to be), it's possible that two or more customers are placing orders at the same time - then all the threads will try to set the `InStockQuantity` to 8.
- After a purchase order is placed, there is no way to know the status of inventory - is it ready for packaging, is it released to shipping, how should we do if a customer cancels the order - should we simply revert back the change because the goods were still on the shelf, or should we record them as outlet because they were packaged and were ready for shipping. Once the changes are made, there are no real way to "revert" them.

Then come the new inventory system - which is built around `IInventoryService` - it's supposed to be smarter and more flexible than the old one. It was introduced in Episerver Commerce 7.19, and was sitting there silently until 8.7, when it was fully integrated into the workflows. (Before that, the new APIs were there but the new features were not used by any parts of the system. Any inventory operations were done in a "compatible layer", the calls to `IWarehouseInventoryService` are forwarded to work with `IInventoryService` internally.)

The new inventory system is much more soliphicated and powerful than the old one, so we will examine it piece by piece. Firstly, by the basic "InventoryRecord" class to represent a record of an inventory:

```
public class InventoryRecord : InventoryKey, IReadOnly<InventoryRecord>
{
    /// <summary>
    /// Gets a value indicating if inventory values are tracked for this item.
    /// </summary>
    public virtual bool IsTracked { get; set;}

    /// <summary>
    /// Gets the quantity of items available for purchase.
    /// </summary>
    public virtual decimal PurchaseAvailableQuantity { get; set;}

    /// <summary>
    /// Gets the quantity of items available for preorder.
    /// </summary>
    public virtual decimal PreorderAvailableQuantity { get; set;}

    /// <summary>
    /// Gets the quantity of items available for backorder.
    /// </summary>
    public virtual decimal BackorderAvailableQuantity { get; set;}

    /// <summary>
    /// Gets the quantity of items requested for purchase and not yet completed.
    /// </summary>
    public virtual decimal PurchaseRequestedQuantity { get; set;}

    /// <summary>
    /// Gets the quantity of items requested for preorder and not yet completed.
    /// </summary>
    public virtual decimal PreorderRequestedQuantity { get; set;}

    /// <summary>
    /// Gets the quantity of items requested for backorder and not yet completed.
    /// </summary>
    public virtual decimal BackorderRequestedQuantity { get; set;}

    /// <summary>
    /// Gets the purchase availability date for the item and warehouse, in UTC.
    /// </summary>
    public virtual DateTime PurchaseAvailableUtc { get; set;}

```

```
    /// <summary>
    /// Gets the preorder availability date for the item and warehouse, in UTC.
    /// </summary>
    public virtual DateTime PreorderAvailableUtc { get; set;}

    /// <summary>
    /// Gets the backorder availability date for the item and warehouse, in UTC.
    /// </summary>
    public virtual DateTime BackorderAvailableUtc { get; set;}

    /// <summary>
    /// Gets a quantity representing additional quantity at the warehouse that i\
s not available
    /// for any transactions. This value is not used in business logic, and exis\
ts only to
    /// support backwards compatibility with the reserved quantity field.
    /// </summary>
    public virtual decimal AdditionalQuantity { get; set;}

    /// <summary>
    /// Gets the reorder minimum quantity. This value is not used in business lo\
gic, and exists
    /// only for backwards compatibility.
    /// </summary>
    public virtual decimal ReorderMinQuantity {get; set; }
}
```

This is more or less map to `IWarehouseInventory`, which some differences. It's the easiest to explain is to look at how an `InventoryRecord` is converted to an `IWarehouseInventory`:

```
    private IWarehouseInventory RecordToInventory(Guid applicationId, InventoryR\
ecord source)
    {
        return WarehouseInventory()
        {
            CatalogKey = new CatalogKey(applicationId, source.CatalogEntryCode),
            WarehouseCode = source.WarehouseCode,
            InStockQuantity = source.PurchaseAvailableQuantity + source.Addition\
alQuantity,
            ReservedQuantity = source.AdditionalQuantity,
            ReorderMinQuantity = source.ReorderMinQuantity,
            PreorderQuantity = source.PreorderAvailableQuantity,
```

```
            BackorderQuantity = source.BackorderAvailableQuantity,
            AllowBackorder = source.BackorderAvailableQuantity > 0,
            AllowPreorder = source.PreorderAvailableUtc > new DateTime(1900, 1, \
1, 0, 0, 0, DateTimeKind.Utc),
            InventoryStatus = source.IsTracked ? InventoryTrackingStatus.Enabled\
 : InventoryTrackingStatus.Disabled,
            PreorderAvailabilityDate = source.PreorderAvailableUtc,
            BackorderAvailabilityDate = source.BackorderAvailableUtc
        };
    }
```

We can see that:

- `InventoryTrackingStatus` only has `Enabled` and `Disabled` now as the other flag - `Ignored` is obsolete. This is mapped to `IsTracked`.
- `InStockQuantity` equals to `PurchaseAvailableQuantity` plus `AdditionalQuantity`, while `AdditionalQuantity` is the `ReservedQuantity`.
- `AllowBackOrder` will be true if `BackorderAvailableQuantity` is greater than 0, and `AllowPreorder` will be true if `PreorderAvailableUtc` is greater than a specific time (the beginning of 20th century, only 116 years ago)

So the way data is stored is changed, but not much. The biggest change is in how it works.

When the new inventory system was introduced, the data from old system was migrated to new one, and the old interface was moved to a new proxy class (`WarehouseInventoryProxy`), and the old class `WarehouseInventoryServiceDatabase` was obsoleted - and it's not just obsolete in term of API:s - the old class works with the old way of storing data and that means those data are not used anywhere else. This is why you should avoid using concrete class from Episerver unless absolutely necessary. Sticking with interfaces is always the safer way.

Into requests

As a developer, you will have to adjust the inventories. In the old system, it's "Setting the value" - and it can be troublesome in some situations. In the new inventory system, it's "requesting a change". So whenever you need to adjust quantities, you send a request to the system - you no longer tell how should the system do it, you only tell the system what you want to do and it will figure out how to do it (remember, it's supposed to be smarter!). This is done with `InventoryRequest`

```
public class InventoryRequest
{
    /// <summary>
    /// Gets or sets the request date, in UTC.
    /// </summary>
    public virtual DateTime RequestDateUtc { get; set; }

    /// <summary>
    /// Gets or sets a collection of <see cref="InventoryRequestItem"/> values.
    /// </summary>
    public virtual IList<InventoryRequestItem> Items { get; set; }

    /// <summary>
    /// Gets or sets additional provider-specific data for the request.
    /// </summary>
    public virtual object Context { get; set; }
}
```

This class is quite simple and easy to follow: `RequestDateUtc` is of course the time of the request in Utc, while `Items` is a collection of items being requested (customer might be placing an order with several items, and you only have to send one request for the entire order. `Context` is an extra object for customization - this property is not used by Episerver itself, but it is reserved for custom implementation of inventory system. So what do we have in an `InventoryRequestItem`?

```
public class InventoryRequestItem
{
    /// <summary>
    /// Gets or sets the item index. Used to correlate request items with respon\
se items.
    /// </summary>
    public virtual int ItemIndex { get; set; }

    /// <summary>
    /// Gets or sets the request type. This may be <see cref="InventoryRequestTy\
pe.Purchase"/>
    /// or <see cref="InventoryRequestType.Backorder"/>.
    /// </summary>
    public virtual InventoryRequestType RequestType { get; set; }

    /// <summary>
    /// Gets or sets the requested catalog entry code.
    /// </summary>
```

```
    /// <remarks>
    /// The catalog entry code is ignored in completion, cancellation, and backo\
rder redemption requests.
    /// </remarks>
    public virtual string CatalogEntryCode { get; set; }

    /// <summary>
    /// Gets or sets the code of the warehouse to request inventory from.
    /// </summary>
    /// <remarks>
    /// The warehouse code is ignored in completion, cancellation, and backorder\
 redemption requests.
    /// </remarks>
    public virtual string WarehouseCode { get; set; }

    /// <summary>
    /// Gets or sets the requested quantity.
    /// </summary>
    /// <remarks>
    /// The quantity is ignored in completion, cancellation, and backorder redem\
ption requests.
    /// </remarks>
    public virtual decimal Quantity { get; set; }

    /// <summary>
    /// Gets or sets an optional operation key to correlate this request with a \
previous request.
    /// </summary>
    public virtual string OperationKey { get; set; }

    /// <summary>
    /// Gets or sets additional provider-specific data for the request.
    /// </summary>
    public virtual object Context { get; set; }

    ...

}
```

Each `InventoryRequestItem` maps to a `LineItem` in an order, and beyond that, there are several properties which might interest you:

- `ItemIndex`: So when we get the response back from the system, we will know which response

is for which request.

- `OperationKey`: This is a specialized string which contains almost everything about the request: the catalog entry code, the warehouse code, the values of requested purchase/preorder/backorder/ quantities, the values of available purchase/preorder/backorder quantities. This value plays a very important role in how the system should react in case there is another request. For certain RequestType(s), you must have the `OperationKey` set in `InventoryRequestItem`.
- `RequestType`: A flag indicates how the system should treat the request. There are several types of request:

```
public enum InventoryRequestType
{
    /// <summary>
    /// The request is for a normal purchase.
    /// </summary>
    Purchase,
    /// <summary>
    /// The request is for a preorder.
    /// </summary>
    Preorder,
    /// <summary>
    /// The request is for a backorder.
    /// </summary>
    Backorder,
    /// <summary>
    /// The request is for a purchase or a preorder. The system will compare the\
 order date
    /// with the purchase and preorder availability dates to determine the appro\
priate operation.
    /// </summary>
    PurchaseOrPreorder,
    /// <summary>
    /// Complete an existing allocation. The operation keys of an existing
    /// allocation must be specified.
    /// </summary>
    Complete,
    /// <summary>
    /// Cancel an existing allocation. The operation keys of an existing
    /// allocation must be specified.
    /// </summary>
    Cancel,
    /// <summary>
```

```
    /// Splits an existing allocation into two parts. The operation keys of an
    /// existing allocation must be specified.
    /// </summary>
    Split,
    /// <summary>
    /// Executes a custom, provider-specific operation. The operation should be \
further
    /// described in the context property.
    /// </summary>
    Custom,
}
```

If you look into first 4 `InventoryRequestTypes` (`Purchase`, `Preorder`, `Backorder`, and `PurchaseOr-Preorder`, you can see they are for placing order (aka they are the "first requests"). The next three ones are for subsequent requests, when you make some changes to previous requests, such as commit it (`Complete`), or cancel it (`Cancel`), or make it into two shipments (`Split`). The last one is reversed for custom implementation, it is not used by default.

- `Context`: same as `InventoryRequest`, it's for the custom implemenation.

So how do the requests work?

When a customer places an order, the workflow (specifically `AdjustInventoryActivity` activity) will create `InventoryRequestItems` for each lineitem in the cart:

```
private InventoryRequestItem CreateRequestItem(int lineItemIndex, InventoryReque\
stType requestType,
                                                string entryCode, string warehou\
seCode, decimal delta)
{
    return new InventoryRequestItem(index, requestType, entryCode, warehouseCode\
, delta, string.Empty, null);
}
```

`requestType` can be `InventoryRequestType.Preorder`, `InventoryRequestType.Purchase` or `InventoryRequestType.Backorder`, depends on the type of order. Note that it does not have to be strictly one "request item" per line item. By default, the activity might create two request items for one item if it must be fulfilled with both order and backorder. Now we have request item(s) for each `LineItem`, time to add up an `InventoryRequest` and make the request to the system:

```
var request = new InventoryRequest(DateTime.UtcNow, requestItems, null);
var inventoryService = ServiceLocator.Current.GetInstance<IInventoryService>();
var response = inventoryService.Request(request);
```

With the old system, when you're done with adjusting the inventory, you are ... done. No other operations needed. However, the new system requires extra processing - it processes your request and returns something. You'll need to act on that 'response' somehow - so the next time you want to make a change on the request you made earlier. Let's see what in that `InventoryRespose`:

```
public class InventoryResponse
{
    /// <summary>
    /// Gets or sets a value indicating if the overall operation was a success.
    /// </summary>
    public virtual bool IsSuccess { get; set; }
    /// <summary>
    /// Gets or sets the request date, in UTC. This will be the same request dat\
e as specified
    /// in the request.
    /// </summary>
    public virtual DateTime RequestDateUtc { get; set; }
    /// <summary>
    /// Gets or sets a collection of responses for each item in the request.
    /// </summary>
    public virtual IList<InventoryResponseItem> Items { get; set; }
    /// <summary>
    /// Gets or sets additional provider-specific data for the response.
    /// </summary>
    public virtual object Context { get; set; }
}
```

So we only interest with what're in `InventoryResponseItem`. It's a rather big class, which you can find its definition here[28]. However, we only need to care about a special property - `OperationKey`. It contains almost every thing for later use.

What is an `OperationKey`? Here's an example of an operation key:

ZZZSZPAU7GKDBPAPWK29X4KW8WQQ7VS7MPJ99Y6SWJT9XBKW6W99SVAMZZA98KYXMLB9ECZ3SARTX

It might look cryptic to you (and you don't really have to care about its value), but it's actually simple than you would think - It's just a serialized string contains information about a request, and is handled by `OperationKeySerializer`:

[28] http://world.episerver.com/documentation/Class-library/?documentId=commerce/9/D5082996

```
public class OperationKeySerializer
{
    public virtual string Serialize(InventoryRequestType requestType, InventoryC\
hange change);
    public virtual bool TryDeserialize(string value, out InventoryRequestType re\
questType, out InventoryChange change);
}
```

So an `OperationKey` contains information about an `InventoryRequestType`, and an `InventoryChange`, and it can be converted back and forth from those values. `InventoryChange` is a rather simple class, contains the code of the entry, the code of the warehouse and the "changes" of inventory values:

```
public class InventoryChange : InventoryKey, IReadOnly<InventoryChange>
{
    public virtual string CatalogEntryCode { get; set; } //from InventoryKey
    public virtual string WarehouseCode { get; set; } //from InventoryKey
    public virtual decimal PurchaseAvailableChange { get; set; }
    public virtual decimal PreorderAvailableChange { get; set; }
    public virtual decimal BackorderAvailableChange { get; set; }
    public virtual decimal PurchaseRequestedChange { get; set; }
    public virtual decimal PreorderRequestedChange { get; set; }
    public virtual decimal BackorderRequestedChange { get; set; }
}
```

After the `InventoryResponse` is returned to the workflow, it needs to store the information - `OperationKeys` somewhere. Naturally, it belongs to the `Shipment`. The `Shipment` has a "secret" field `OperationKeys`, and here's it looks like:

0:1#ZZZSZPAU7GKDBPAPWK29X4KW8WQQ7VS7MPJ99Y6SWJT9XBKW6W99SVAMZZA98KYXMLB9ECZ3SA
1:1#ZZZS7PAU7GKDB2VQMPTQ8KYD7M92XEYNWG9964NLLPGXEEYAWM926RA27FT2EEYDMCDTZGV3MF

This value can be accessed by `Shipment.OperationsKeyMap` property. As you'll learn in next chapter, a `Shipment` has to keep track of its `LineItems`. It will now need to know the inventory request information of each `LineItem`. In the above example of `OperationKeys`, we have a `Shipment` with two `LineItems` - they are identified by their "index" in the `LineItem` collection of the OrderForm. Each `LineItem` has one `OperationKey`, although it's possible to have more than one `OperationKey`. Those `OperationKey` are required for later processing, such as shipping or cancelling a shipment. Normally, we don't change each item individually, but they are processed in

We just need to send requests with the saved `OperationKey` and `InventoryRequestType` - depends on the type of action we want to do, either `Complete` or `Cancel`.

```
private void SendRequests(Shipment shipment, InventoryRequestType requestType)
{
    var itemIndexStart = 0;
    var requestItems = shipment.OperationKeysMap.SelectMany(c => c.Value).Select\
(key =>
            new InventoryRequestItem()
            {
                ItemIndex = itemIndexStart++,
                OperationKey = key,
                RequestType = requestType
            }).ToList();

    if (requestItems.Any())
    {
        InventoryService.Request(new InventoryRequest(DateTime.UtcNow, requestIt\
ems, null));
        shipment.ClearOperationKeys();
    }
}
```

Note that after sending the request, we need to call `ClearOperationKeys`. The data stored in `OperationKeys` is no longer needed in the future, and we should prevent any wrongdoing with that data.

So what actually happens under the hood?

When a request is made to the inventory system, depends on the type of the request, let's say in the most common case, `Purchase`, the values stored in `InventoryService` table are updated. `PurchaseAvailableQuanity` is decreased, while `PurchaseRequestedQuantity` is increased. (In cases multiple requests are made, for example, both `Purchase` and `BackOrder`, the respected values will be all updated). There is a substainable different compared to the old inventory system here. In the old system, the values are set by the computed values from application layer. As we mentioned earlier, this will cause issues in concurrent cases where two or more threads want to set the values at the same time. In the new system, the values are "adjusted" - the API:s only let the database know which changes are needed, and the database will handle concurrent itself. I would doubt that we can write better, more robust concurrent handling code than SQL Server engine itself.

When a shipment is shipped, the `Complete` request will make sure the change is permanent - the value in `PurchaseRequestedQuantity` will be decreased. However if the shipment is cancelled before it is shipped, the change will be reverted back -

When should you cancel a request? It only makes senses to cancel when the shipment has not been shipped, or even the goods have not been leaving the shelf. In case the goods have been packaged, or shipped, you will need to handle it separately - to treat the goods as outlet, for example. That's the reason when we ship the goods, we use the `Complete` request: once the shipment is shipped, the inventory system has done its parts.

Part 3: The order system - selling is what (really) matters

No matter how shiny your website is, no matter how cool your URL is, and no matter how fancy the technologies you used to build it - in the end, it's the orders you take that really matter - the most important measure to a success of a e-Commerce website.

This part will be focus on how the order system works, and how you can do better.

Chapter 6: The Order system.

Overview

Order system, as same as the catalog system, is based on the MetaDataPlus MetaClass system. That means, you can extend your classes with additional attributes to match with your business need. By default, it can be seen as even more complicated than the catalog system. With Catalog system, you only have two base classes (CatalogEntry and CatalogNode, you can consider CatalogEntryEx and CatalogNodeEx are default classes, but they are rarely used), but for the order system, there are quite many. Here's the list of default order metaclasses from the default installation:

26	26	Mediachase.Commerce.Orders.System	OrderForm	Order Form	1	0	0	OrderForm
27	27	Mediachase.Commerce.Orders.System	Shipment	Shipment	1	0	0	Shipment
28	28	Mediachase.Commerce.Orders.System	LineItem	LineItem	1	0	0	LineItem
29	29	Mediachase.Commerce.Orders.System	OrderGroupAddress	Order Group Address	1	0	0	OrderGroupAddress
30	30	Mediachase.Commerce.Orders.System	OrderFormPayment	Order Form Payment	1	0	0	OrderFormPayment
31	31	Mediachase.Commerce.Orders.System	OrderFormEx	Order Form Extended	0	0	26	OrderFormEx
32	32	Mediachase.Commerce.Orders.System	ShipmentEx	Shipment Extended	0	0	27	ShipmentEx
33	33	Mediachase.Commerce.Orders.System	LineItemEx	LineItem Extended	0	0	28	LineItemEx
34	34	Mediachase.Commerce.Orders.System	OrderGroupAddressEx	OrderGroup Address E...	0	0	29	OrderGroupAddressEx
35	35	Mediachase.Commerce.Orders.System	CreditCardPayment	Credit Card Payment	0	0	30	OrderFormPayment_CreditCard
36	36	Mediachase.Commerce.Orders.System	CashCardPayment	Cash Card Payment	0	0	30	OrderFormPayment_CashCard
37	37	Mediachase.Commerce.Orders.System	InvoicePayment	Invoice Payment	0	0	30	OrderFormPayment_Invoice
38	38	Mediachase.Commerce.Orders.System	GiftCardPayment	Gift Card Payment	0	0	30	OrderFormPayment_GiftCard
39	39	Mediachase.Commerce.Orders.System	OtherPayment	Other Payment	0	0	30	OrderFormPayment_Other
40	40	Mediachase.Commerce.Orders.System	ExchangePayment	Exchange Payment	0	0	30	OrderFormPayment_Exchange
41	41	Mediachase.Commerce.Orders.System	PurchaseOrder	Purchase Order	0	0	25	OrderGroup_PurchaseOrder
42	42	Mediachase.Commerce.Orders.System	ShoppingCart	Cart Order Class	0	0	25	OrderGroup_ShoppingCart
43	43	Mediachase.Commerce.Orders.System	PaymentPlan	Payment Plan	0	0	25	OrderGroup_PaymentPlan

How stuffs are structured.

At the center of the Order system, is the `OrderGroup`. It's base class of three most important classes in the system: `PurchaseOrder`, `PaymentPlan` and `ShoppingCart` (Well, it's a little inconsistent here, the class itself is `Cart`, but the corresponding metaclass is `ShoppingCart`. We'll use the terms exchangeably). `OrderGroup` is a big class - like most of other classes in order system - and here are some of the most important

```
public abstract class OrderGroup : OrderStorageBase, ISerializable, IOrderGroup
{
    /// <summary>
    /// Gets the order group id.
    /// </summary>
    public int OrderGroupId { get; }

    /// <summary>
    /// Gets the order forms.
    /// </summary>
    public OrderFormCollection OrderForms { get; private set; }

    /// <summary>
    /// Gets the order addresses.
    /// </summary>
    public OrderAddressCollection OrderAddresses { get; private set; }

    /// <summary>
    /// Gets or sets the order notes.
    /// </summary>
    public OrderNoteCollection OrderNotes { get; set; }

    /// <summary>
    /// Gets or sets the name.
    /// </summary>
    public string Name { get; set; }

    /// <summary>
    /// Gets or sets the customer id.
    /// </summary>
    public Guid CustomerId { get; set; }

    /// <summary>
    /// Gets or sets the total.
    /// </summary>
    public decimal Total { get; set; }

    /// <summary>
    /// Gets or sets the billing currency.
    /// </summary>
    public string BillingCurrency { get; set; }

```

```
    /// <summary>
    /// Gets or sets the status.
    /// </summary>
    public string Status { get; set; }

    /// <summary>
    /// The market this order belongs to.
    /// </summary>
    public MarketId MarketId { get; set; }

    ...
}
```

You can refer to this link[29] for detail information of properties and methods.

Those classes share almost the same major characteristics:

- They all have a CustomerId (which maps to a customer, either registered or anonymous), cost-related properties (`Total`, `SubTotal`, `HandlingTotal` etc.) and a collection of addresses.
- They all have `OrderForm`(s). In most of the cases, there will be only one `OrderForm`, but in some B2B systems, you might see two or more. Each `OrderForm` will have information about payments, promotions attached to that form. Each `OrderForm` might have one or more `Shipments`, a list of `OrderAddress`(es):

```
    public class OrderForm : OrderStorageBase, ISerializable, IOrderForm
    {
        /// <summary>
        /// Gets the order form id.
        /// </summary>
        public int OrderFormId { get; }

        /// <summary>
        /// Gets the shipments.
        /// </summary>
        public ShipmentCollection Shipments { get; }

        /// <summary>
        /// Gets the payments.
        /// </summary>
        public PaymentCollection Payments { get; }
```

[29] http://world.episerver.com/documentation/Class-library/?documentId=commerce/7.5/193A3DA9

```
17
18          /// <summary>
19          /// Gets the line items.
20          /// </summary>
21          public LineItemCollection LineItems { get; }
22
23          /// <summary>
24          /// Gets the discounts.
25          /// </summary>
26          public OrderFormDiscountCollection Discounts { get; }
27
28          ...
29      }
```

- As the unit to be shipped, each shipment will have one Shipping address (It'll be the address of the store, in case it's an in-store pickup shipment), shipping method information, shipping cost and so on. Each `Shipment` has a list of `LineItems` belong to it[30].
- Each `LineItem` has information about the the entry it represents, the quantity, the price when it was added to cart (`PlacedPrice`) , the price considering quantity and discounts (`ExtendedPrice`), and then discounts for itself.

It's `PaymentPlan` which is a little different, because it has cycle information attached.

However, those classes are different in the places they are used, and therefore, they are processed with different workflows, resulting in different behaviors. You can save a `Cart` in Commerce Manager without any payments attached, but you will need to have a valid payment amount (which match the order's total) to be able to save a `PurchaseOrder`.

And as `OrderGroup` and its derived classes are all `MetaObject`, you can always call `.AcceptChanges()` on an instance of them, to get it saved to the database. And what if you want to delete it? `.Delete()` followed by `.AcceptChanges()` will do the job.

Let's explore those classes one by one.

Cart

A cart is supposed to be unique per market, name and customer. If the customer already has another cart in same market, same name in the system, then when he or she try to save another cart, it would result in an exception. Episerver Commerce has a check to prevent such thing from happening - the default behavior when a customer browses the site is, the existing cart will be loaded if he already has one, a new one is only created if there is none in existent.

[30] This is not entirely true. The lineitems collection belong to the `OrderForm`, and the lineitems in a shipment is only a subset mapped to that shipment. This will be explained in later section.

Of course, `PaymentPlan` and `PurchaseOrder` have no such restriction.

`Cart` extends `OrderGroup` by adding certain const strings for the name of the carts:

```
public class Cart : OrderGroup
{
    /// <summary>
    /// Gets the default name for the cart.
    /// </summary>
    public static string DefaultName { get; }

    /// <summary>
    /// Gets the name for the Wishlist.
    /// </summary>
    public static string WishListName { get; }

    /// <summary>
    /// Gets the exchange name for the cart.
    /// </summary>
    public static string ExchangeName { get; }

    ...
}
```

How to create a cart? Sadly, there is no way such as:

```
var cart = new Cart();
```

Instead, this is the way we do today:

```
var cartHelper = new CartHelper();
```

`CartHelper` is a helper class, provides convenient way to access and manipulate `Cart`. It's open sourced in `Mediachase.Commerce.Website` package. For a very long time, it's the way to create and update the shopping cart. `CartHelper` has another constructors which take the name of the cart, the customer Id and the market he is in. The pamameterless constructor above will use all default values - the default cart name (Can you guess it? It's "Default"), the current customer Id (in form of a `Guid` - even anonymous customer has one) and the current market:

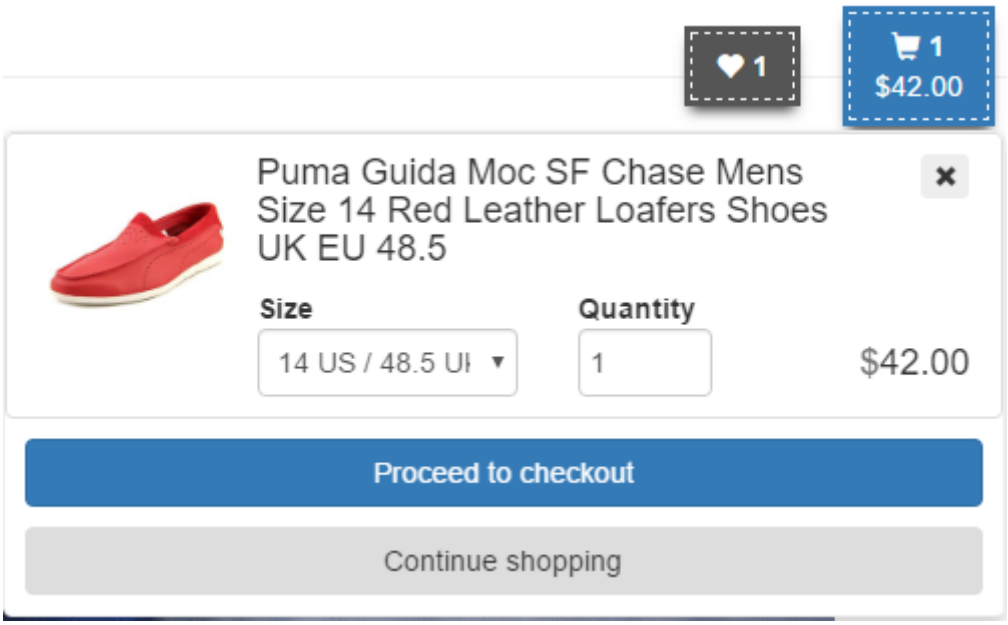
```
var cartHelper = new CartHelper(name, userId, market);
```

What is so bad about `CartHelper`? Mostly, because it had become, over the time, the GOD class which does everything about the `Cart`, new functions had been added without much thinking about architecting the class - it is doing too much. `CartHelper` is almost impossible to mock, so it's painful to write unit tests for cart-related functionalities.

And then, we can access the `Cart` by:

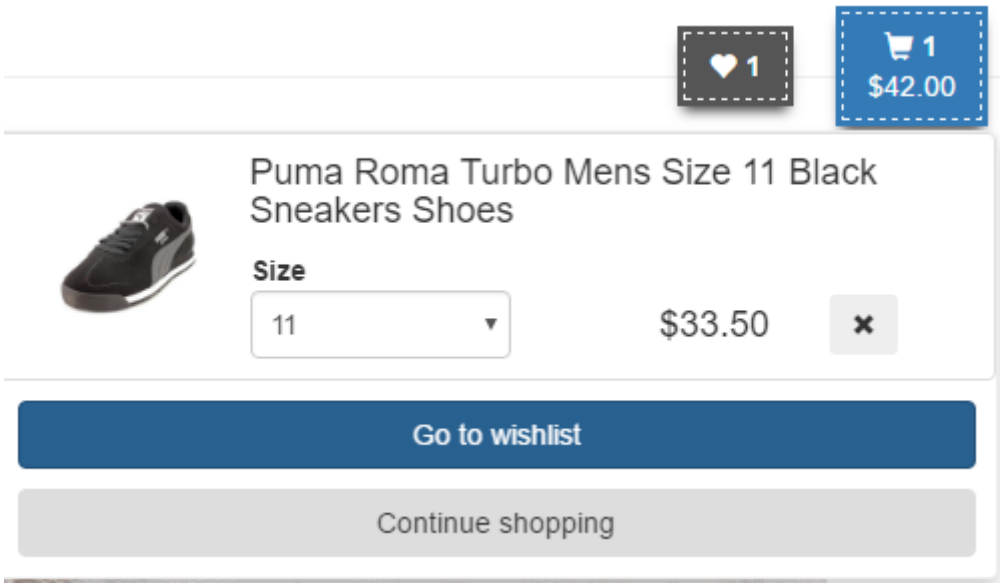
```
var cart = cartHelper.Cart;
```

`CartHelper` constructor will return the existing, or create a new one if none existed.



The cart implementation in QuickSilver

Another special form of `Cart` is wishlist - it is just a `Cart` with defined name by `Cart.WishListName` constant ("WishList", if you wonder). Again, you can choose a different name for your wishlist, just make sure to stick with it.



The wishlist implemenation on QuickSilver, note that it displays no quantity

There are several known "best practices" for wishlist, such as you should not care about the quantity of items in a wishlist, or you should not make a check for inventories when an item was added to it, or only registered, logged in users have wishlists, when a customer adds an item from wishlist to cart, it should be removed from wishlist. However, these are not definitive rules, and you can of course implements your wishlist as your business requires.

By default, when a customer logs in, his current carts will be merged with his existing carts in the system (assuming both carts have line item(s)). We will look into how to change that behavior later.

PaymentPlan

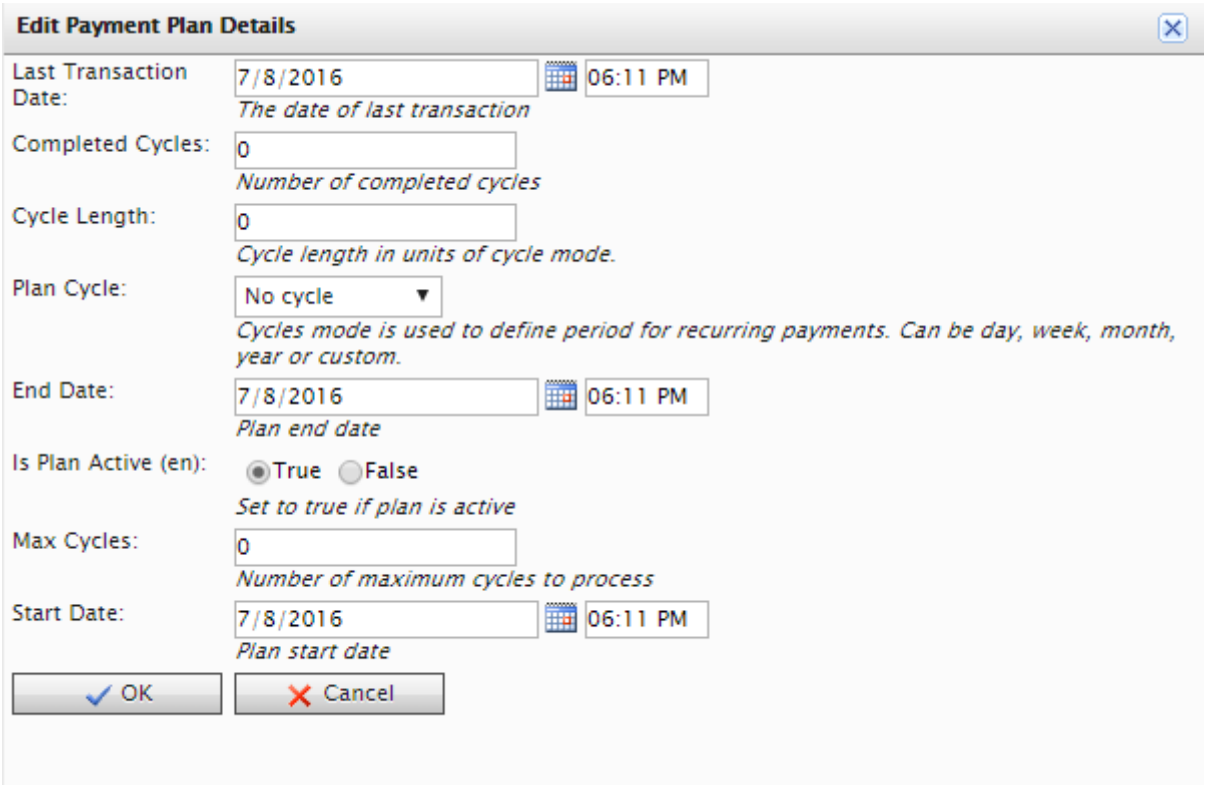
`PaymentPlan` is a special form of `OrderGroup` where you can create and manage the cycle of recurring order. For simplest example, think about a subscription. Your customer wants to order a magazine to be delivered to his home the 5th of every month, for six months. Instead of having manually create an order for him every month (trust me, it's a tedious work, even if you have only few customers), you can create a `PaymentPlan` to handle such requirements:

```
public class PaymentPlan : OrderGroup
{
    public virtual PaymentPlanCycle CycleMode { get; set;}
    public virtual int CycleLength { get; set;}
    public virtual int MaxCyclesCount { get; set;}
    public virtual int CompletedCyclesCount { get; set;}
    public virtual DateTime StartDate { get; set;}
    public virtual DateTime EndDate { get; set;}
    public virtual DateTime LastTransactionDate { get; set;}
    public virtual bool IsActive { get; set;}
    ...
}
```

`PaymentPlan` is really flexible in term of cycle, it allows you to set whatever cycle your customers would like:

```
public enum PaymentPlanCycle
{
    /// <summary>
    /// Unknown Plan Cycle
    /// </summary>
    None = 0,
    /// <summary>
    /// Daily Cycle
    /// </summary>
    Days = 1,
    /// <summary>
    /// Weekly Cycle
    /// </summary>
    Weeks,
    /// <summary>
    /// Monthly Cycle
    /// </summary>
    Months,
    /// <summary>
    /// Yearly Cycle
    /// </summary>
    Years,
    /// <summary>
    /// Custom1 Cycle, use your own time frame.
    /// </summary>
    Custom1,
```

```
    /// <summary>
    /// Custom2 Cycle, use your own time frame.
    /// </summary>
    Custom2
}
```



Cycle settings for payment plan, it is simpler than it seems

Note that if `CycleMode` is `None` then the `PaymentPlan` will only happen once. It will behave like one-time `PurchaseOrder`.

What if you want an order to be created every 3 days, or every 2 weeks - that's when `CycleLength` comes to help - you can set `CycleMode` to `Days` and `CycleLength` to 3, or `CycleMode` to `Weeks` and `CycleLength` to 2, respectively. Combine `CycleMode` and `CycleLength` and your payment plans can fit into even most complicated subcriptions. You can control the number of subscription to be made, by setting the value of `StartDate` and `EndDate`, and `MaxCyclesCount`.

`StartDate` and `EndDate` can be used to set the limit of the cycle if you know the time it would end (when calculating the number of cycle is not trivial, like having an order every week, for 6 months), while `MaxCyclesCount` can be used when you know exactly the number of times you should convert the `PaymentPlan` to purchase order (having 30 orders, every other week). Those are not exclusive, they can be used in conjunction and which condition is reached first will be used.

You will usually not create a `PaymentPlan` by code directly, but to create a `Cart` first, and then convert it to a `PaymentPlan`:

```
var paymentPlan = cart.SaveAsPaymentPlan();
paymentPlan.AcceptChanges();
```

To turn a `PaymentPlan` to a `PurchaseOrder`, it's as simple as call:

```
var purchaseOrder = paymentPlan.SaveAsPurchaseOrder();
```

However, you normally don't have to do it - the point of `PaymentPlan` is to make sure it can be automated once it's created. Episerver Commerce comes with a scheduled job, named "Subscription Payment Plans", which will convert any `PaymentPlans` which statisfies the conditions (`.IsDue` returns true) to `PurchaseOrder`. You can set this scheduled job to run daily, and let it does it job.

PurchaseOrder

`PurchaseOrder` is what you want an `OrderGroup` to be - it's the cart which has been converted successully to an order. `PurchaseOrder` extends `OrderGroup` with some notable properties:

```
public class PurchaseOrder : OrderGroup
{
    /// <summary>
    /// Gets or sets the parent order group id. This can be used to relate purch\
ase orders to a payment plan or other order.
    /// </summary>
    public int ParentOrderGroupId { get; set;}

    /// <summary>
    /// Tracking number. Automatically generated or can be generated using provi\
ded function.
    /// </summary>
    public string TrackingNumber { get; set;}

```

```
    /// <summary>
    /// Expiration date can be used for subscription type of orders.
    /// </summary>
    public DateTime ExpirationDate { get; set;}

        /// <summary>
    /// The collection of Return merchandise authorization (RMA).
        /// </summary>
    public OrderFormCollection ReturnOrderForms { get; }
}
```

There are several ways to create an order, but normally you will call `SaveAsPurchaseOrder` method on a `Cart` or `PaymentPlan`. For example, this is the normal way to create a `PurchaseOrder`:

```
OrderGroupWorkflowManager.RunWorkflow(Cart, OrderGroupWorkflowManager.CartCheckO\
utWorkflowName, true, dic);
Cart.CustomerId = PrincipalInfo.CurrentPrincipal.GetContactId();
var po = Cart.SaveAsPurchaseOrder();

if (_currentContact != null)
{
    Cart.CustomerName = _currentContact.FullName;
}
purchaseOrder.ProviderId = "FrontEnd";
OrderNotesManager.AddNoteToPurchaseOrder(purchaseOrder, noteDetail, OrderNoteTyp\
es.System, PrincipalInfo.CurrentPrincipal.GetContactId());
po.AcceptChanges();

// Remove old cart
Cart.Delete();
Cart.AcceptChanges();
```

Note that this code has been simplified. But the idea is simple: You run the `CartCheckout` workflow, as it will process payments, adjust inventories, and calculate the totals of the order. Then you save the `Cart` as a `PurchaseOrder`, add some more information to it (in this case, adding an `OrderNote`), then save it again. Finally, you delete the old `Cart`.

It's worth noting that `.SaveAsPurchaseOrder()` (as well as `.SaveAsPaymentPlan()`) does not really "transform" the `Cart` into a `PurchaseOrder`, instead, it creates a new `PurchaseOrder` and copy the data over. Now we have both the `Cart` and the `PurchaseOrder` existing in the system. That's why we need to call the last two lines - the old `Cart` needs to be deleted.

As you might see, we set the `ProviderId` of the purchase order. What's it for? In simplest scenario, you can use it to distinguish between orders placed by front-end site (end customers) and backend (Commerce Manager - by editors/managers). It also can be used to see which site customer placed orders on, in case you have multiple sites. Certain activities, such as `ValidateLineItemsActivity`, will validate the prices based on this value - it only allows price of an item to be set manually if `ProviderId` is not "FrontEnd".

OrderForm - Shipment - LineItem

We learned about `OrderGroup`, about `PaymentPlan` and `PurchaseOrder`. However, there was a lie when you save a cart as a purchase order or a payment plan. If the cart is "empty" - aka it does not contain any items, the saving will fail with an `InvalidOperationException`. Therefore, we will have to add `LineItems` to cart.[^foo361]

There is usually one `OrderForm` per `OrderGroup`, however for B2B scenarios, it's possible to have multiple `OrderForm`.

[^foo361] Only `LineItem`(s) are needed. If the cart has no shipment by default, the default implementation will add it for you. However, you might want to add shipments to your cart, as we discussed later.

A `LineItem` represents an item in the order. When you add an item to the cart - it is turned to a `LineItem`. As we are using the old order system, we'll use `CartHelper` for that:

```
var cartHelper = new CartHelper();
var wishlist = new CartHelper(Cart.WishListName).Cart;
var entry = variationContent.LoadEntry();
cartHelper.AddEntry(entry, 1, false, new Cart[] {wishlist});
```

If you're new to Episerver Commerce, we have plenty thing to talk about. Here we are loading two carts: The first one is the default cart ("Default") and the second one is wishlist ("WishList"). We are using an overload which removes the item from other carts (wishlist) when it is added to a cart. Simplify things a lot for us, right?

It's also worth noted that `CartHelper` does not work with neither `CatalogEntryDto` nor `VariationContent`, but `Entry`. `Entry` is a POCO object represents an entry. It has been there since the first version of Episerver Commerce, and was supposed to be an object-approach alternative of `CatalogEntryDto`. It's a big class - you can find the full properties and methods of it here[31]. Outside of being used by `CartHelper` for adding items to cart like the example above, it's hardly useful for anything else - with the fact that an `Entry` is readonly. If you are using Commerce 7.5 and later,

[31] http://world.episerver.com/documentation/Class-library/?documentId=commerce/9/A421552

you should be using `EntryContentBase` - we're lucky that there are plenty of extensions methods for `EntryContentBase` to load the corresponding `Entry`.

There are several overloads of `AddEntry`- you can have the full list here[32]. In this example, we are adding an entry with `quantity` = 1, with no fixed quantity[^foo362], and we are also removing the item from wishlist, if it is already in that list.

[^foo362] Fixed quantity means if you want to set the quantity of the item to be a new value. So if you have the quantity to be 3, and fixed quantity is set to true, then the quantity of the cart will be 3, no matter the quantity you currently have the the cart. In the above example, we have fixed quantity = false, so the quantity (1) will be added to the current quantity of the item.

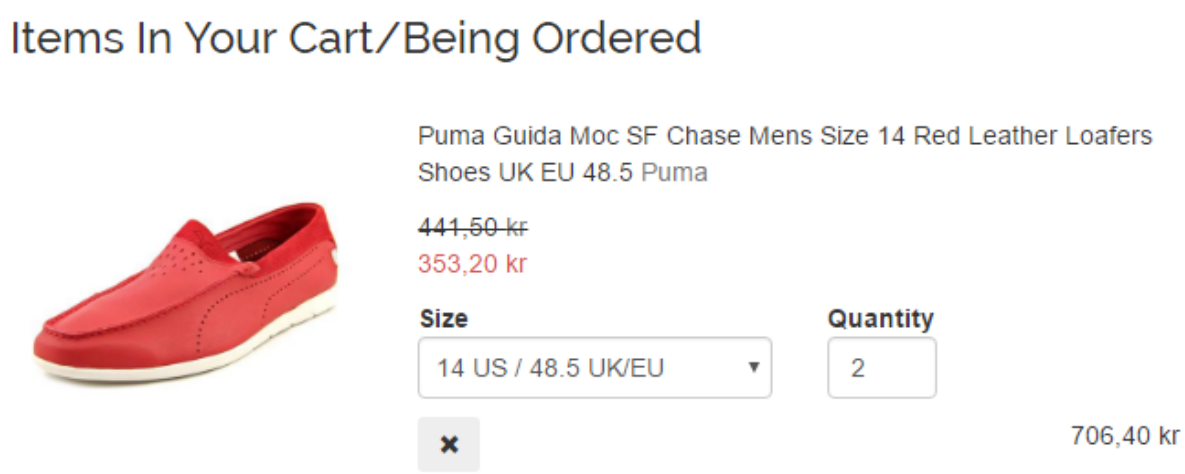
So what's in a `LineItem`? As other classes in order system, it's a big class with a lot of properties which have been added over versions - you can find the full class definition here[33], but here the most important properties:

```
public class LineItem
{
    /// <summary>
    /// The catalog entry code.
    /// </summary>
    public string Code { get; set;}

    /// <summary>
    /// Display name, usually the display name of the entry.
    /// </summary>
    public string DisplayName { get; set;}

    /// <summary>
    /// The quantity of the item.
    /// </summary>
    public decimal Quantity { get; set;}

    /// <summary>
    /// The price per unit of the item when it's added.
    /// </summary>
    public decimal PlacedPrice

    /// <summary>
    /// The price, including quantity and discounts.
    /// </summary>
    public decimal ExtendedPrice
```

[32] http://world.episerver.com/documentation/Class-library/?documentId=commerce/9/8324683B
[33] http://world.episerver.com/documentation/Class-library/?documentId=commerce/9/F1B0EB53

```
27
28      /// <summary>
29      /// The line item discount amount.
30      /// </summary>
31      public decimal LineItemDiscountAmount { get; set;}
32
33      /// <summary>
34      /// The order level discount amount.
35      /// </summary>
36      public decimal OrderLevelDiscountAmount { get; set;}
37  }
```

Here's an example of lineitem in cart in Quicksilver:



A lineitem in cart

You can see clearly how a lineitem is presented:

- `DisplayName` is shown to customer - however, it's the `Code` (which is also the entry's code) which is used by the system.
- The `PlacedPrice` is 441.50kr, but there are discount(s), so each item only costs 353.20kr. With the quantity of 2, so we have the `ExtendedPrice` of 706.40kr. The total of `LineItemDiscountAmount` and `OrderLevelDiscountAmount` is 176.60kr.

Normally, value such as `DisplayName`, `ExtendedPrice`, `LineItemDiscountAmount` and `OrderLevelDiscountAmount` will be set by other parts of system: `DisplayName` is copied from the `DisplayName` of the entry, while other values are set by the "calculators" when workflows run. You should only have to care about `Code` and `Quantity`.

It's important to keep in mind that all quantities in Episerver Commerce are (treated as) decimal: lineitem quantity or values of the inventories. However, depends on your implementation, it might make sense to treat the cart's items quantities as `integer`. For example, if your site sells clothes, the `ViewModel` for items in cart should use `integer` for quantities. It's funny to allow customers buying 3.6 shirts, right?

A lineitem is not shipped on its own. It's shipped via `Shipment`. It's time to create a `Shipment` and add it to the `OrderForm`.

```
var shipment = cartHelper.Cart.OrderForm[0].Shipments.AddNew();
```

This will create a shipment and add it to the `Shipment` of the `OrderForm` - we are working in a normal B2C case so there should be only one `OrderForm`. Until now, the lineitems we added to `cartHelper` belong to the `OrderForm`, and can be accessed via `OrderForm.LineItems`. Time to give them to the shipment. How?

```
var index = orderForm.LineItems.IndexOf(lineItem);
shipment.AddLineitemIndex(index);
```

This seems ... harmless. But it can be dangerous!

The quirk of Shipment and LineItem relationship

At its core, it's the `OrderForm` which owns the collection of `LineItems`. Assume we have lineitem A, B, C with the quantity of 3,4,2, respectively. The indices of A, B, C in the `OrderForm.LineItems` collection are 0,1,2. In case you have only one shipment, then the `LineItems` collection belong to that shipment entirely, all is good. Let's see when you have more than one shipment, which is a quite common scenario.

The collection of `LineItems` is still there, and the `OrderForm` still owns it. However, to distribute the lineitems across the shipments, there is the concept of index-quantity mapping. The distribution in the first shipment is A:1, B:2, C:1, and in the second shipment is A:2, B:2, C:1.

To store that information, the indices of the lineitems was used, so they will be 0:1, 1:2, 2:1 and 0:2, 1:2, 2:1, respectively. All is still good, at this point.

The problem arises when you want to remove one lineitem, change it quantity or move it around (to another shipment, for example). That's when the index-quantity max shows it true face: fragile. You can easily get the wrong quantity, or even worse, get an IndexOutOfRangeException just because a lineitem was removed but the map was not updated.

It's something in design and you just can't avoid it entirely. But you can still avoid those problems by using methods provided in `PurchaseOrderManager`, instead of manipulating the `LineItems` collection your own.

Trust me, a good amount of power brain has been spent to make sure those methods work correctly and reliably.

So if you want to remove a lineitem from the cart, this is NOT the correct way:

```
var lineItem = orderForm.LineItems.FirstOrDefault(l => l.Code == code);
orderForm.LineItems.Remove(lineItem);
lineItem.Delete();
```

but instead: {lang=C#} var lineItem = orderForm.LineItems.FirstOrDefault(l ⇒ l.Code == code); PurchaseOrderManager.RemoveLineItemFromOrder(orderGroup, lineItem.LineItemId);

`PurchaseOrderManager.RemoveLineItemFromOrder` will do a lot of things: it will remove the lineitem from shipments which contain it, then remove the lineitem from the orderform, finally, it will re-index the lineitem index-quantity mapping, to make sure everything is in the correct and stable state.

`RemoveLineItemFromOrder` is not the only thing you can use in `PurchaseOrderManager`, there are other useful methods, such as `RemoveLineItemFromShipment` or `UpdateShipmentLineItemQuantity`. Just let me re-say this: use `PurchaseOrderManager` to manage your shipment-lineitems relations, do not do it yourself.

Until when the new Order system comes, which might happen later this year (2016), it's something we have to live with, for now.

Addresses and Shipping methods

So we added `LineItems` to the cart, then add them to the shipment. To make the shipment ship-able, we still have works to do: to know where and how to ship it!

Addresses.

Each `OrderGroup` has a collection of addresses - which can be accessed via `OrderGroup.OrderAddresses`. Usually, you have one or more *shipping addresses* and one *billing address*.

```
public class OrderAddress
{
    public string Name { get; set;}

    public int OrderGroupId { get; set;}

    public string FirstName { get; set;}

    public string LastName { get; set;}

    ...
}
```

Manipulating addresses in an order can be a complicated task. Here's how the Episerver Commerce works to resolve that:

- If the current user is registered and logged in, then at checkout page, he or she can select from `CustomerContact.PreferredBillingAddress` and `CustomerContact.PreferredShippingAddress`, if those settings have values. Otherwise, the customer can choose from *address book*, or input a new address.
- If the current user is anonymous, then he or she must input at least one address (if shipping and billing addresses are the same)
- If customer choose In Store pickup (Handoff) shipping method, then the address of selected warehouse will be copied to an `OrderAddress`, then add this `OrderAddress` to `OrderGroup.OrderAddresses`.

```
    public static OrderAddress ConvertWarehouseContactInfoToOrderAddress(IWareho\
use warehouse)
    {
        OrderAddress address = new OrderAddress();
        address.City = warehouse.ContactInformation.City;
        address.CountryCode = warehouse.ContactInformation.CountryCode;
        address.CountryName = warehouse.ContactInformation.CountryName;
        address.DaytimePhoneNumber = warehouse.ContactInformation.DaytimePhoneNu\
mber;
        address.Email = warehouse.ContactInformation.Email;
        address.EveningPhoneNumber = warehouse.ContactInformation.EveningPhoneNu\
mber;
        address.FirstName = warehouse.ContactInformation.FirstName;
        address.LastName = warehouse.ContactInformation.LastName;
        address.Line1 = warehouse.ContactInformation.Line1;
```

```
        address.Line2 = warehouse.ContactInformation.Line2;
        address.Name = OrderHelper.GetSafeAddressName(warehouse.Name);
        address.PostalCode = warehouse.ContactInformation.PostalCode;
        address.RegionName = warehouse.ContactInformation.RegionName;
        address.RegionCode = warehouse.ContactInformation.RegionCode;
        address.State = warehouse.ContactInformation.State;
        return address;
    }
```

- The address will then be saved to `OrderGroup.OrderAddresses`. If the address was an `CustomerAddress`, it will be copied over to an `OrderAddress` (the simplest way to do so is to use `CustomerAddress.CopyCustomerAddressToOrderAddress(AddressEntity customer-Address, OrderAddress orderAddress)`),then added to `OrderGroup.OrderAddresses`.
- `OrderForm.BillingAddressId`,`Payment.BillingAddressId` and each `Shipment.ShippingAddressId` will point to `Name` of one `OrderAddress` in `OrderGroup.OrderAddresses`. At checkout page, here's what our code should look like:

```
var shippingAddress = new OrderAddress();
//Set the fields of shippingAddress
purchaseOrder.OrderAddresses.Add(shippingAddress);
shipment.ShippingAddressId = shippingAddress.Name;
```

During the checkout, the customer can create many addresses, however, only the selected shipping addresses and billing address will be saved, all other unused addresses will be deleted (If you recall from previous section, the cart gets deleted, a new purchase order gets created, so un-references objects like the addresses will be deleted).

Shipping methods

The only crucial information missing for Shipment, is the shipping method customer selected to ship the goods. We'll talk about the shipping methods in depth in Part 8 - Extending the system, but now we'll only limit to how to get the shipping methods and how to set that information to the shipment.

One way to show available shipping methods to customers

```
ShippingMethodDto shippingMethodDto = ShippingManager.GetShippingMethodsByMarket\
(CurrentMarket.MarketId.Value, false);
//Construct the model to display shipping methods to end user
```

Here we're getting the shipping methods available in the current market. `ShippingMethodDto` is another `DataSet`, but we should only care about its `ShippingMethod` table now. Each `ShippingMethodRow` has a `ShippingMethodId`, which we can use to bind to the radio buttons like the screenshot above. Now when the customer selected the shipping method, we need to set it to the shipment:

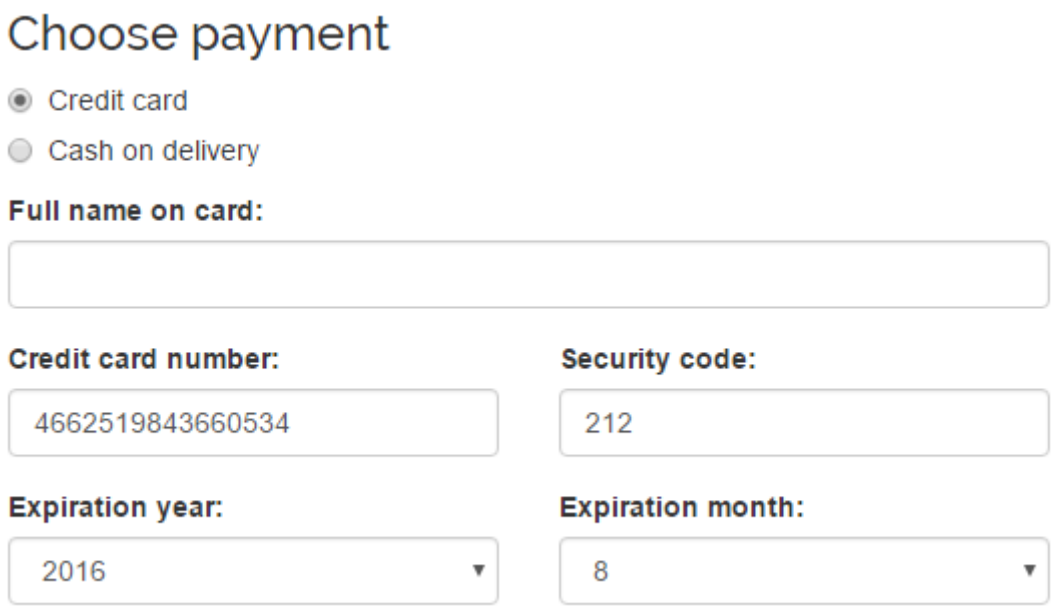
```
ShippingMethodDto shippingMethodDto = ShippingManager.GetShippingMethodsByMarket\
(CurrentMarket.MarketId.Value, false);
var selectedShippingMethod = shippingMethodDto.ShippingMethod.FirstOrDefault(m =\
> m.ShippingMethodId == selectedShippingMethodId);
shipment.ShippingMethodId = selectedShippingMethod.ShippingMethodId;
shipment.ShippingMethodName = selectedShippingMethod.Name;
```

Payments

Now it's time for customer to ... pay. (If your site does not require that, please send me the link). First, we need to show the available payment methods to the customer:

```
 var payments = PaymentManager.GetPaymentMethodsByMarket(_currentMarket.MarketId\
.Value, SiteContext.Current.LanguageName, false).PaymentMethod.Select()
foreach (var paymentRow in payments)
{
    var paymentMethod = new PaymentMethod(paymentRow);
    //Create the Model for the payment
}
```

Here we are getting the *active* payment methods in the current market and current site language (the `false` parameter indicates that we don't want *inactive* payment methods). Then for each `PaymentMethodRow` found, we create a `PaymentMethod`. The most important thing in a `PaymentMethod` is the `PaymentMethodId`, which must be used for identifying the payment method.



The simplifed payment methods selection in Quicksilver. Shhh, please don't tell anyone my credit card number

then add the selected one to the `OrderForm`.

```
var payment = selectedPayment.PreProcess(orderForm);
if (payment != null)
{
    payment.PaymentMethodId = paymentMethodRow.PaymentMethodId;
    payment.PaymentMethodName = paymentMethodRow.Name;
    payment.BillingAddressId = orderForm.BillingAddressId;
    payment.Amount = orderForm.Total;
    orderForm.Payments.Add(payment);
}
```

It's required to call `CartValidate` workflow before adding payment(s) to the `OrderForm` - it does all the hard jobs - apply all applicable promotions, calculate the taxes and shipping costs. The amount of money you charge your customer must perfectly match the total of the order.

`selectedPayment` is a component (User control, Partial view) which implements `IPaymentOption` interface. In `PreProcess`, you can do some checks to verify that the information customer provided is valid (for example, the credit card is not expired), and return a `Payment` object. The final thing is to set some properties to that `Payment` object and add it to `OrderForm.Payments`. Those payments will be finally processed in later stage (hint: when you run `CartCheckout` workflow).

Like shipping methods, we'll discuss morre about payment methods in Part 8. See you there!

AcceptChanges()

It's important to understand how saving in order system works. As we mentioned in above section, to save an `OrderGroup`, including `Cart`, `PurchaseOrder` or `PaymentPlan`, simply call `AcceptChanges()` on the object. What does it save? Everything.

So this is not necessary (in fact, it does not work, the order of saving here is not correct, the "parent" objects must be saved first):

```
lineItem.AcceptChanges();
shipment.AcceptChanges();
orderForm.AcceptChanges();
cart.AcceptChanges();
```

this is already enough:

```
cart.AcceptChanges();
```

Calling `AcceptChanges()` on an `OrderGroup` will also save all the children properties, including collections. It's also pretty smart - only changed objects (object with status of `MetaObjectState.Added`, `MetaObjectState.Deleted`, `MetaObjectState.Modified`) are saved. Unchanged objects, well, they remains unchanged.

It's also important to remember that calling `AcceptChanges()` on an `OrderGroup` is an expensive operation. Many calls to database will be done: for each and every object inside an `OrderGroup`, there is at least one separate save to database. That's why you should postpone saving `Carts` as late as possible, and save as few times as possible. If your siute is doing multiple cart saves in one `HttpRequest` - it's time to review the code to remove unwanted saves. You probably need only the last one.

Workflows

When a customer places an order, there are several steps that require extra processing - those can be done by calling certain workflows. Workflows, or more precisely, `ActivityFlows`, have been an essential part of Episerver Commerce since the first version, and play an important part in checkout process.

However, with the raise of new promotion system, it's not a mandatory thing to run anymore. For websites which sell downloadable contents and use the new promotion system, there is almost no reason to use workflows. We will learn more about workflows in Part 8, but here we are talking about the basic of workflows - what are they, and how can they be used.

When a customer adds an item to cart - you should call `CartValidate`. This workflow will check if the items in cart can be fulfilled (if this is a normal purchase order, if this is pre-order, or backorder, then the inventories must have special settings to allow that) and if the cart is entitled to any promotions. If an item does not satisfy the check, it will be removed from the cart. If there is a promotion which can be applied, discounts will be made to the price and/or gift items will be added to cart. `CartValidate` also records those discount usages - so the system can keep tracks of promotion redemption.s

There are several ways to call the workflow. The most common way in this stage is propably:

```
cartHelper.RunWorkflow("CartValidate");
```

the other way is to use `OrderGroupWorkflowManager`:

```
OrderGroupWorkflowManager.RunWorkflow(cartHelper.Cart, OrderGroupWorkflowManager\
.CartValidateWorkflowName);
```

When they finish adding items to cart, and proceed to checkout, it's time to call another workflow `CartPrepare`. It's almost as same as `CartValidate`, except it will calculate *taxes*. `CartPrepare` also calculate discounts, but it does not record the promotion usage again.

Finally, when they added shipping/billing addresses and payment information, the final step is to run `CartCheckout` workflow. It will process the payments, make requests to inventory system, and finally record the promotion usage as "Used".

There are most important workflows which are used on the frontend site. There are several others, but you rarely use them directly by code - they are mostly used by funtionalities in Commerce Manager.

There is a small exception - if you're using your own system to process order, you might need to use `CompleteShipmentWorkflow` workflow.

Here's the full list of all builtin workflows in Episerver Commerce, defined by names:

```
public const string CartValidateWorkflowName = "CartValidate";
public const string CartPrepareWorkflowName = "CartPrepare";
public const string CartCheckOutWorkflowName = "CartCheckout";

public const string OrderRecalculateWorkflowName = "RecalculatePurchaseOrderWork\
flow";
public const string OrderSaveChangesWorkflowName = "SaveChangesWorkflow";

public const string OrderCompleteShipmentWorkflowName = "CompleteShipmentWorkflo\
w";
public const string OrderSplitShipmentsWorkflowName = "SplitShipmentsWorkflow";
public const string OrderCalculateTotalsWorkflowName = "RecalculateTotalsWorkflo\
w";
public const string ReturnFormCompleteWorkflowName = "ReturnFormCompleteWorkflow\
";
public const string ReturnFormRecalculateWorkflowName = "ReturnFormRecalculateWo\
rkflow";

public const string CheckAndReserveInstorePickupWorkflowName = "CheckAndReserveI\
nstorePickupWorkflow";
```

It's important to remember that a `Cart`, or a `PurchaseOrder`, needs to be saved prior before calling workflows. Otherwise, some activities will try to load the information from the database and the results will not be correct. In other hands, it does make senses to save as little as possible. Each call to `.AcceptChanges` on an `OrderGroup` will save everything on that instance - `OrderForm`, `Shipment`, `LineItem`, etc. to database, then duplicated calls will result on performance hit. As a best practice, try to save only *once* per request - for example, when a user add or remove one item from the cart, it should only be saved only *once*, nothing more.

Handling order statuses

If you are using Commerce Manager to handle orders, it does not really make sense to understand order statuses, programmatically. However, if you want to to handle orders by external systems, it's a must.

Here's the list of available statuses for orders. Remember that `enum` is not extensible in C#, so you have to stick with those values [34]

[34] Technically, you can. Those values are stored as a string column in database, so you can set any values you want. However, the existing parts of system won't be able to handle the custom values.

```
[Flags]
public enum OrderStatus
{
        OnHold = 1,
        PartiallyShipped = 2,
        InProgress = 4,
        Completed = 8,
        Cancelled = 16,
        AwaitingExchange = 32
}
```

Let's examine the statuses, one by one.

In this image above, we can see that the order is in `InProgress` state. This is the default status when a purchase order is made. You can see that there are button to put this order to be `OnHold` and `Cancelled`.

To put an order to `OnHold` state, we can call:

```
OrderStatusManager.HoldOrder(purchaseOrder);
```

When an order is in `OnHold` state - it's read only, for example when you are waiting for customer's confirmation - no changes can be made to the order until you release it from hold.

```
if (OrderStatusManager.GetPurchaseOrderStatus(purchaseOrder) == OrderStatus.OnHo\
ld)
{
    OrderStatusManager.ReleaseHoldOnOrder(purchaseOrder);
}
```

`ReleaseHoldOnOrder` will throw an exception if `purchaseOrder` is not already `OnHold` - so it's best for check for the status before actually releasing it from holding.

Althought it's not required, it's best practice to add a note to the order whenever you change the order status, to save history of why did the order get changed. Here's how we should do it

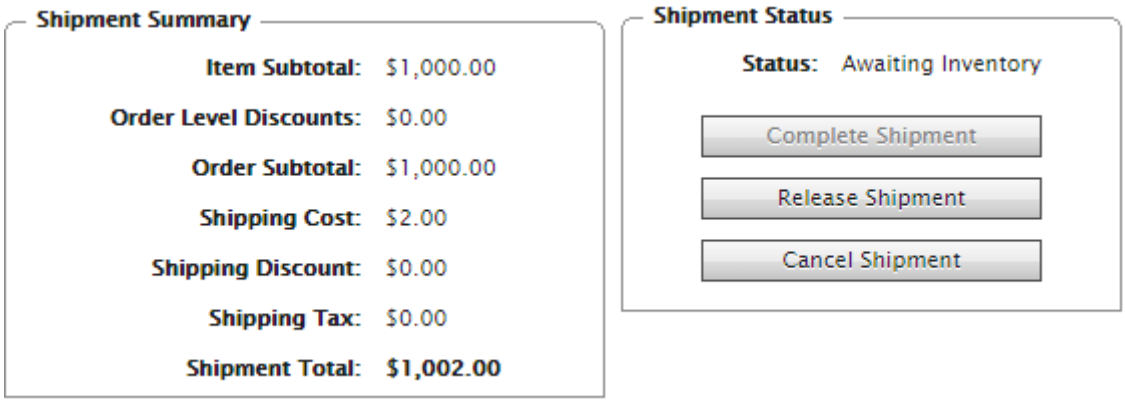
```
OrderStatusManager.HoldOrder(purchaseOrder);
OrderNotesManager.AddNoteToPurchaseOrder(purchaseOrder, "Order put on hold to wa\
it for customer's confirmation", OrderNoteTypes.System,
                                                PrincipalInfo.CurrentPrincipal.\
GetContactId());
```

While holding a purchase order only makes it readonly - canceling an order does a lot more: it does not only set the status of the order to `Cancelled`, it also cancels all the shipments in the orders - so any inventories reserved for the order, if the shipments haven't been shipped, will be reverted back, payments made by the orders will be processed, this time with `TransactionType.Credit` flag (to refund the money, if possible, to customer).

However, as the seller, you won't want the order to be put on hold or be cancelled - you want the order to be completed - the goods to be shipped and the customers to be happy with their purchase. In such case, you need to dig into `OrderShipmentStatus`:

```
[Flags]
public enum OrderShipmentStatus
{
        AwaitingInventory = 1,
        Cancelled = 2,
        InventoryAssigned = 4,
        OnHold = 8,
        Packing = 16,
        Released = 32,
        Shipped = 64
}
```

You can guess when some statuses are used - `OnHold` is when the entire order is put on hold, and `Cancelled` is when the order is cancelled. However, in case the shipment has multiple shipments, each shipment can be cancelled separately.



Each shipment can be cancelled separately

To cancel a shipment:

```
OrderStatusManager.CancelOrderShipment(shipment);
```

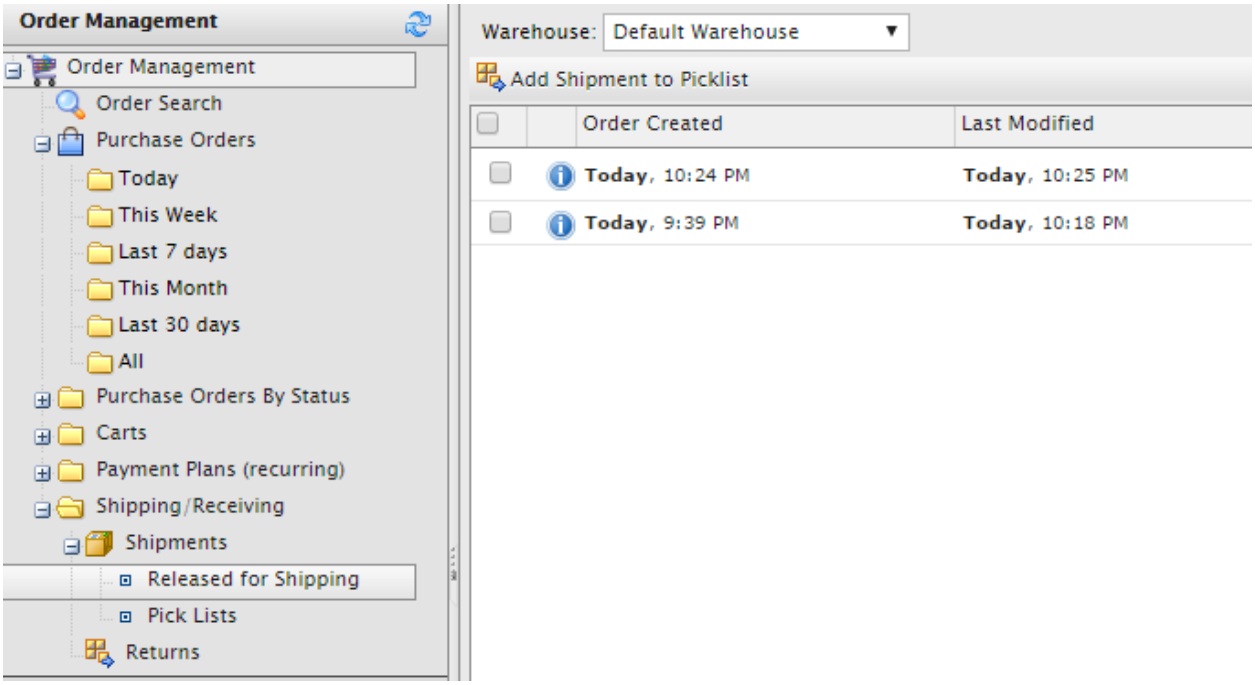
In case the order has only one shipment, or if all other shipments in the order have been cancelled, it does make sense to cancel the order itself:

```
OrderStatusManager.CancelOrderShipment(shipment);
if (purchaseOrder.OrderForms[0].Shipments.All(c=>c.Status.Equals(OrderShipmentSt\
atus.Cancelled.ToString()))
{
    OrderStatusManager.CancelOrder(purchaseOrder);
}
```

By default, when an order is placed, the shipment is in `AwaitingInventory` state. We must now release it from inventory to make it ready to delivery:

```
if ((!ShippingManager.IsHandoffShippingMethod(shipment.ShippingMethodName) ||
    OrderStatusManager.IsShipmentReleaseable(shipment)) &&
    OrderStatusManager.IsOrderPaid(purchaseOrder))
    {
        OrderStatusManager.ReleaseOrderShipment(shipment);
    }
```

Here we are doing some additional checks - the shipment should not be “in store pickup” delivery, it should be “releasable”, and the order has been paid. Lucky for us, `OrderStatusManager` provides all required methods to do those checks. Once released, the shipment will be in `Released` state (of course), and ready to be picked up for delivery.



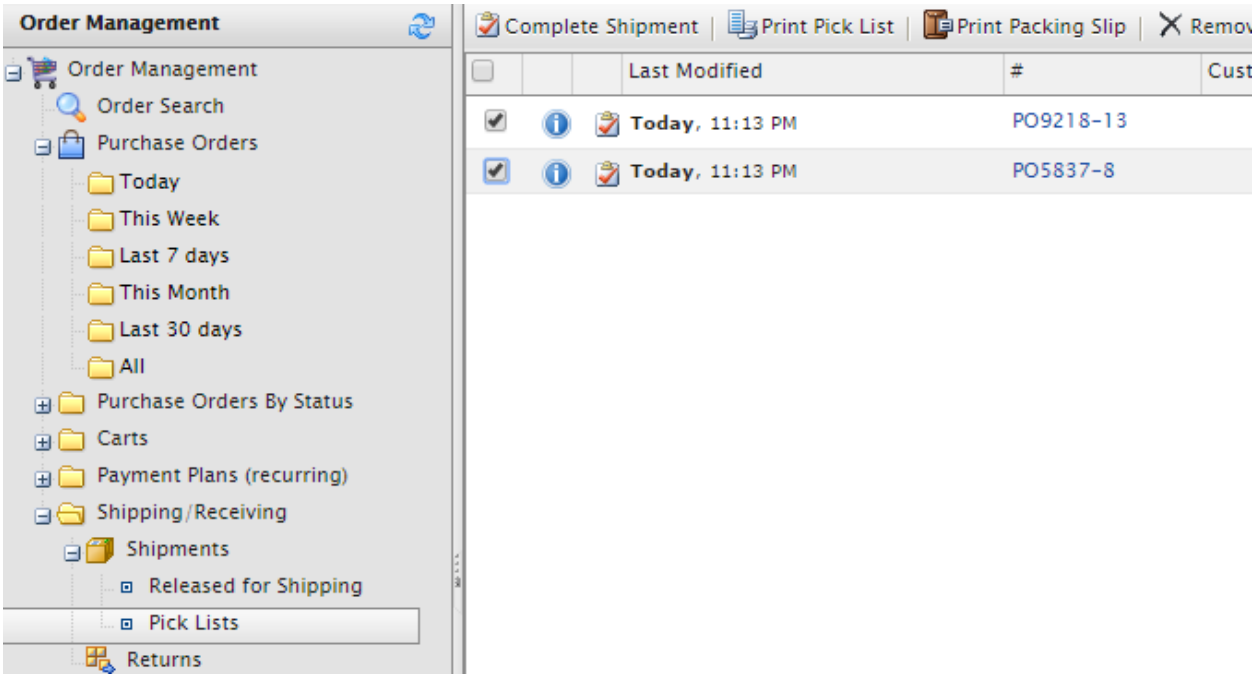
Released shipments, ready for picked up for delivery

Here we can add shipments to picklists. By the code, it’s equivalent to set the status of the shipment to `Packing`:

```
OrderStatusManager.PickForPackingOrderShipment(shipment);
```

If you are new to Commerce, picklists are the list of goods ready to be delivered - so the delivery guys can pick up the packages from warehouses and deliver it to awaiting customers. After a shipment is picked up, it can be completed (finally! [35])

[35] In fact, you are not required to set a shipment to `Packing` state to make it “complete-able”. After a shipment is relased, you can complete it directly.



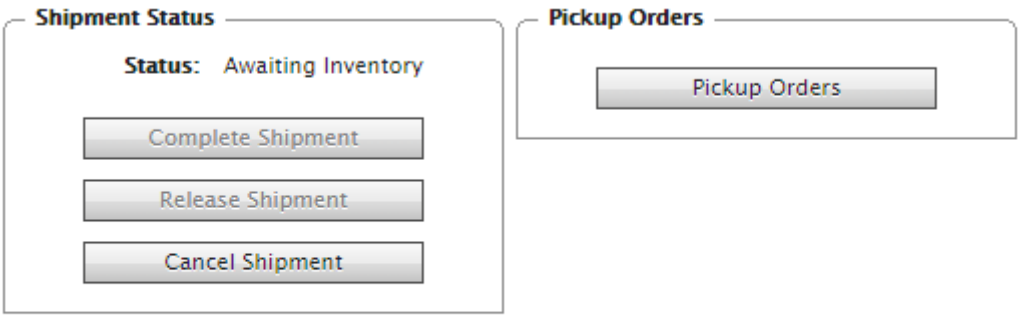
Complete shipments, the final step

```
OrderStatusManager.CompleteOrderShipment(shipment);
```

Completing a shipment is an important step. It does not only set the status of the shipment to `Shipped`, but also it finalizes the payments. For some payment gateways, such as PayPal, when an order is placed, the payment is holding - it is subtracted from buyer's account, but not yet transfered to your site's account. Only when you the customer acknowledges that he or she received the goods - the payment is "released" aka, completes.

If the order has only one shipment, or all other shipments have been shipped, completing that shipment will effectively completes the whole order.

We've been examining the normal case when the goods need to be delivered. However, it's not always the case. When the customer chooses to get the goods directly at the warehouse (in store pickup), it does not really make senses to "release shipment". When the customer picks the shipment up, it completes the shipment.



Pick up shipment and complete it directly

```
if(ShippingManager.IsHandoffShippingMethod(shipment.ShippingMethodName) &&
     (OrderStatusManager.IsShipmentReleaseable(shipment)
     || OrderStatusManager.IsShipmentPackable(shipment)
     || OrderStatusManager.IsShipmentCompletable(shipment))
&& OrderStatusManager.IsOrderPaid(purchaseOrder))
     {
         OrderStatusManager.CompleteOrderShipment(shipment);
     }
```

In this section we talked about purchase order - but all methods mentioned here also have overloads to work with `PaymentPlan`. Remember? They are all `OrderGroup`!

Of course, if you are selling digital contents, when the purchase order is placed, you are free to make it completed directly.

Chapter 7: New Order system - the new promise

Why a new system

You might ask - why a new system. The order system is not so bad.

But it is not really good, neither. And here are somethings it is not great at:

- Performance: The design of `PurchaseOrder` and `ShoppingCart` as two metaclasses of `OrderGroup`, so they're entitled to share the same base table (`OrderGroup`) in database: whenever you place an order, it's actually a cart being deleted and a purchase order being created. Two operations in same table, you know it's bad for performance, especially there is an index there. It's even worse with the bad practice of saving the cart it's changed. Whenever a customer add or remove an item from the cart, it's supposed to be saved. Imagine you have hundreds or even thousands of customers browsing your website, it'll kill the performance of placing orders.
- The API:s lack of abstraction. You can't just save the orders to somewhere else. (You can, of course, take the order and save it in another place, but it does mean that you are saving to two places, and end up loading from two places). And lacking of abstraction also mean it's hard to unit test. It's almost impossible to mock those classes. This is a very frequent complain about `CartHelper` - althought it's not in the core API:s. It's like a wrapper for `Cart` which is, again, hard (if not possible) to mock.
- The quirks of `Lineitem-Shipment` relation

The new order system is promised to solve those problems. Let's see how those promises are fulfilled.

Everything is an interface.

Well, almost. In previous chapter, you did know about `OrderGroup`, `OrderForm`, `Shipment`, and `LineItem`. Those are concrete `MetaClass`(es) - therefore they are tightly coupled with the database and it's almost impossible if you want to plug your own implementations.

To allow the required abstraction, new interfaces were introduced: `IOrderGroup`, `IOrderForm`, `IShipment`, `ILineItem`, `IOrderAddress`, `IPayment`, etc. Let's take a look at `IOrderGroup`:

```
public interface IOrderGroup : IExtendedProperties
{
    /// <summary>
    /// Gets an instance of the class <see cref="OrderReference"/>, which contai\
ns the unique identity for the order group, and the type of order group it repre\
sents (cart, purchase order, or payment plan)
    /// </summary>
    OrderReference OrderLink { get; }

    /// <summary>
    /// Gets the collection of order forms in the order group.
    /// </summary>
    IEnumerable<IOrderForm> Forms { get; }

    /// <summary>
    /// Gets the order status.
    /// </summary>
    OrderStatus OrderStatus { get; set; }

    /// <summary>
    /// Gets the market.
    /// </summary>
    IMarket Market { get; }

    /// <summary>
    /// Gets the currency used in the order.
    /// </summary>
    Currency Currency { get; }

    ...
}
```

Note `IExtendedProperties` - it's the core of all other abstractions, as it represents the concept of `MetaClass` for the abstraction. It's a rather simple interface:

```
public interface IExtendedProperties
{
    /// <summary>
    /// Gets the property bag for dealing with custom meta fields.
    /// </summary>
    Hashtable Properties { get; }
}
```

That's it. You can store the collection of `MetaFields` there.

Note the `IEnumerable<IOrderForms> Forms` property. You might have guessed - it's an abstraction over `OrderGroup.OrderForms`.

Like `IOrderGroup`, `IOrderForm`, `IShipment` and other interfaces are implemented in an explit way. So if you have an `OrderGroup`, which implements `IOrderGroup`, you can not access its `Forms` by simply calling

```
orderGroup.Forms
```

but instead:

```
((IOrderGroup)orderGroup).Forms
```

You might see it as inconvenient, wouldn't it easier to implemnt it as the property of the implemenation? If you are mixing old API:s and new interfaces, yes. However, the explicit interface implementation will enforce the way you work with API:s. You should only work with interface without having to care about implementations. By that, when you want to change your implementation, such as adding new storage option, you don't have to change the code you wrote. It also enforce the decoupling with the implemenation, and makes it easier for you to mock and write unit tests for your code.

IOrderRepository - Gooobye CartHelper

You might have noticed the `OrderReference` property. It's not related `ContentReference`, but it has something in common. As its name might suggest, it's a identifier to an `OrderGroup`. It has four properties, but only `OrderGroupId` (which is the id) and `OrderType` (which is the type) really matter. `OrderReference` is the way to work with orders in a repository pattern - using `IOrderRepository`. Let's see how `OrderReference` looks like:

```
public class OrderReference
{
    public OrderReference(int orderGroupId, string name, Guid customerId, Type o\
rderGroupType);

    public int OrderGroupId { get; private set; }

    public string Name { get; private set; }

    public Guid CustomerId { get; private set; }

    public Type OrderType { get; private set; }
}
```

This class is, in fact, redundant. `OrderGroupId` and `OrderType` are enough to identify an `OrderGroup`[36]. `Name`, `CustomerId` can be get from the `IOrderGroup`.

In a way, `IOrderRepository` can be seen to be inspired by `IContentRepository`. It's one unified API:s to deal with `IOrderGroup`, regardless of their true types (`Cart`, `PurchaseOrder` or `PaymentPlan`). It's the providers, which know how to work with a specific type, handle the leg work. (Remember `CatalogContentProvider`?) `IOrderRepository` also has some extension methods to make it more convenient to work with `OrderGroup`.

You still remember `CartHelper`? You can forget it now, here's the new way to create a cart:

```
var cart = _orderRepository.LoadCart<ICart>(CustomerContact.Current.CurrentConta\
ctId, name);
```

This will load the cart in the current market (there is another extension method to load with a specific market), or return null if there is no cart existing.

> It's a best practice to delay the creation of `Carts` as late as possible. In the old system, whenever you call to `new CartHelper()`, a `Cart` will always be called if it has not yet existed. This has a major drawback - every customer visiting your site will have at least one `Cart`, and you will, in the end, have a lot of empty carts in the system, resulting in a performance hit. In the new system, it is more flexible as you can try to load a `Cart` and if it is `null`, simply return an empty mini cart. Only when your customer actually adds something to the cart, a real `Cart` will be created in database.

When we actually need to have the cart in the database (customer is adding items to cart), it's time to call another overload, `LoadOrCreateCart`. It behaves like `new CartHelper()` - create a new one if there is none.

[36] With the default implementation, `OrderGroupId` is supposed be unique across the system, as all `Cart`, `PaymentPlan` and `PurchaseOrder` share the same `OrderGroup` table. However, this might change in the custom implementation where you save your carts and your purchase orders in different storages. This is one the main points of the new order system - allow you to persist the data wherever you want.

```
var cart = _orderRepository.LoadOrCreateCart<ICart>(CustomerContact.Current.Curr\
entContactId, name);
```

And you might have guessed how to save a cart:

IOrderFactory - everything in memory

New order system comes with a *Factory* - `IOrderFactory` allows you to create in-memory instances of order classes.

```
public interface IOrderFactory
    {
        ICart CreateCart(Guid customerId, string name);

        IPurchaseOrder CreatePurchaseOrder(Guid customerId);

        IPaymentPlan CreatePaymentPlan(Guid customerId, string name);

        IOrderForm CreateOrderForm();

        IShipment CreateShipment();

        ILineItem CreateLineItem(string code);

        IOrderAddress CreateOrderAddress();

        IOrderNote CreateOrderNote();

        IPayment CreatePayment();

        ICreditCardPayment CreateCardPayment();

        ITaxValue CreateTaxValue();
    }
```

`IOrderFactory` represents an important philosophy in new order system: everything should be in-memory until you need to persist them. This allows the abstraction to work better. In the old system - the instances are tightly coupled with database. When you create a `Cart` - you create it in database and every later operation requires a save to database. With new system, you create everything in memory, manipulate everything in memory and only saves when you absolutely need. Database

access, is, indeed, much slower than memory access. Cutting database saves is the reason new order system is expected to be *much* faster.

Now we need to add some items to cart. If the lineitem is already in the cart, if not, create a new one then add. In the end, increase the quantity of the lineitem by 1.

```
var allLineItems = cart.Forms.SelectMany(f => f.Shipments).SelectMany(s => s.Lin\
eItems);
var lineItem = allLineItems.FirstOrDefault(x => x.Code == code);

if (lineItem == null)
{
    lineItem = _orderFactory.CreateLineItem(code);
    cart.AddLineItem(lineItem, _orderFactory);
}

lineItem.Quantity++;
```

Wait a minute. Shouldn't we can get all the `LineItems` by getting them from `OrderForm` directly? No we can't (anymore). With the new abstractions, `IOrderForm` contains `IShipments`, and each `IShipment` contains `LineItems`. There is an existing overload to get all lineitems in the cart - `cart.GetAllLineItems()` which does exactly the samething.

Next, we create a lineitem by `IOrderFactory.CreateLineItem(code)` and that's it. No more `Entry` or heavy database operations. Then we simply add the newly created `ILineItem` to the cart. Wait (another) minute here. The `ICart` does not own the `ILineItems`, at least, not directly. We need to add the `ILineItem` to an `IShipment`, which itself belongs to an `IOrderForm`. `IOrderGroup.AddLineItem` is an extension method which does that for you, but if you want to know what is going on under the hood, then here it is:

```
var form = orderGroup.Forms.FirstOrDefault();
if (form == null)
{
    form = orderFactory.CreateOrderForm();
    form.Name = orderGroup.Name;
    orderGroup.Forms.Add(form);
}

var shipment = form.Shipments.FirstOrDefault();
if (shipment == null)
{
    shipment = orderFactory.CreateShipment();
    form.Shipments.Add(shipment);
```

```
}

shipment.LineItems.Add(lineItem);
```

Once the item was added to cart, it's time to save.

```
var orderLink = _orderRepository.Save(cart);
```

`IOrderRepository.Save(IOrderGroup)` is equivalent with `Cart.AcceptChanges()`, or `PaymentPlan.AcceptChanges`, or `PurchaseOrder.AcceptChanges`, depends on the actual type of `IOrderGroup`. The same rule still applies - we should only save when it's absolutely necessary. Save your "Save" for last!

IOrderAddress

`IOrderAddress` is the interface extracted from `OrderAddress`. If you recall from previous section, to set the address of a `Payment` or a `Shipment`, you first need to add the `OrderAddress` to `OrderGroup.OrderAddresses` collection, then set the `ShippingAddressId` or `BillingAddressId` to the name of one of the addresses. A little too many steps, right? Here's the way in new API:s

```
shipment.ShippingAddress = shippingAddress;
payment.BillingAddress = billingAddress;
```

where `shipment` is an `IShipment` and `payment` is an `IPayment`. `shippingAddress` and `billingAddress` are both `IOrderAddress`, which can be created by `IOrderFactory` by:

```
var shippingAddress = _orderFactory.CreateOrderAddress();
//Set values of the fields.
```

However, there is a limitation here: in default implementation, their underlying class must be `OrderAddress`, otherwise an `Exception` will be thrown. `shippingAddress` and `billingAddress` will be added to `OrderGroup.OrderAddress` it they are not already existed.

Simple, huh? I told you the new API:s are to be loved.

Workflows or non-workflows, that's the question.

In old versions of Commerce, workflows can be an convenient and straighforward way to handle carts and orders. However, not every one likes workflows. Some find it to be unintuitive and confusing. Others want to be explicit - calling workflows seems to hide underlying operations. And customizing workflows is not everyone's favorite.

Now with the new abstractions, it's possible to get rid of workflows. Instead of calling a workflow to do a bunch of stuffs, now you can call exactly what you want to. If comes with a caveat - you *must* be using new promotion system to be able to follow this way. The old promotion system is tightly coupled with workflows.

`IOrderGroup` comes with a plenty of extension methods to allow you to do stuffs which previously are responsibilities of workflows. Here we are talking about the most simple overloads - there are other overloads which take extra parameters. If you want to unit test your code - make sure to use those with dependencies as you can mock them in your tests.

- To apply discounts on the cart:

```
1     cart.ApplyDiscounts();
```

- To validate the cart to see if any items are no longer valid (out of stock, not available in current market, prices have changed etc.):

```
1     var validationIssues = new Dictionary<ILineItem, List<ValidationIssue>>();
2     cart.ValidateOrRemoveLineItems((item, issue) => validationIssues.AddValidati\
3 onIssues(item, issue));
4     cart.UpdatePlacedPriceOrRemoveLineItems((item, issue) => validationIssues.Ad\
5 dValidationIssues(item, issue));
6     cart.UpdateInventoryOrRemoveLineItems((item, issue) => validationIssues.AddV\
7 alidationIssues(item, issue));
```

- To send requests to inventory system to allocate inventory for items in the cart:

```
1     var validationIssues = new Dictionary<ILineItem, List<ValidationIssue>>();
2     cart.AdjustInventoryOrRemoveLineItems((item, issue) => validationIssues.AddV\
3 alidationIssues(item, issue), _inventoryProcessor);
```

- To process payments:

```
cart.ProcessPayments();
```

Pretty simple, right? If you hate workflows enough - Episerver has something else for you. And of course, it looks better.

The new cart system.

One of the main points of the new order system is to allow the extensibility - so you can plug your implementation into the order system. You don't want to save/load the orders to Commerce database, but to your ERP system? No problem. You want to have Redis as the backend for the carts? Easy peasy. But we're not there yet. Order system is big, and replacing it takes a lot of time, if you remember that every change must be carefully considered because the data is extremely valuable. So it'll be done in pieces, and the first piece is the cart system.

At this time of writing, a new cart system is under testing and will be released soon. What does it do? And what is the benefit? It's much faster! In some tests, it is 10x times faster than the old system. 10x. How?

To explain why the new system is faster - we need to understand why the old system is slow. As we learned in the previous section, the carts and the orders share same tables - OrderGroup, OrderForm, Shipment, LineItem, etc. There are two cases which the old cart system perform badly: When you have a B2B site where each order can be 100, or even more items. So when you save to a cart, it's actually (at least) 100 round trips to data base. And when a customer places an order, aka the cart, and everything belongs to it is deleted.

In later version of Commerce, the saving for meta objects was improved. If a meta object such as a lineitem was not changed, it'll not be saved when you save the cart. This drasmatically improves the performance of saving. However, if you save a new cart with 100+ lineitem, performance will still suffer.

The new cart system solves these two issues by the serialization approach. Except some mandatory information such as the name of the cart, customer id or the market id, all other information is serialized and stored in a big column. Now no matter how many items you have in your cart, it's only one round trip to database. And when a customer places an order, it'll still delete the cart, but again it's only one round trip, and it'll not be the same tables with the order.

You'll get the biggest benefits if you are a B2B site which big cart, or if your site has high concurrent when many customers continously adding/removing items from carts and placing orders (like the BlackFriday sale ;)). If you are in one of the those situations, it makes senses to evaluate the upgrade to new cart system.

The new cart system is not enabled by default when you upgrade your site. You will need to turn it on manually by this setting in ecf.app.config. (In the chapter about promotion system, we will have the same kind of setting for the promotion system)

<Application defaultApplicationName="ECApplication"> <Features> <add feature="SerializedCarts" state="Enabled" type="Mediachase.Commerce.Core.Features.SerializedCarts, Mediachase.Commerce" /> <Features> </Application>

This will enable one scheduled job allowing to migrate the old carts into new ones. Depending on your number of carts, it might takes minutes to hours for the migration.

But that's not everything. You'll need to update your code to make sure it works with the new system. How? Simply replace your usages of `IOrderFactory` with `IOrderGroupFactory`:

```
/// <summary>
/// Responsible for creating instances of order group components.
/// </summary>
public interface IOrderGroupFactory
{
    /// <summary>
    /// Retrieves the <see cref="IOrderGroupBuilder"/> implementation for the sp\
ecific order group.
    /// </summary>
    IOrderGroupBuilder BuilderFor(IOrderGroup orderGroup);

    /// <summary>
    /// Creates the order form.
    /// </summary>
    IOrderForm CreateOrderForm(IOrderGroup orderGroup);

    /// <summary>
    /// Creates the shipment.
    /// </summary>
    IShipment CreateShipment(IOrderGroup orderGroup);

    /// <summary>
    /// Creates the line item.
    /// </summary>
    ILineItem CreateLineItem(string code, IOrderGroup orderGroup);

    /// <summary>
    /// Creates the order address.
    /// </summary>
    IOrderAddress CreateOrderAddress(IOrderGroup orderGroup);

    /// <summary>
    /// Creates the order note.
    /// </summary>
```

```
    IOrderNote CreateOrderNote(IOrderGroup orderGroup);

    /// <summary>
    /// Creates the payment.
    /// </summary>
    IPayment CreatePayment(IOrderGroup orderGroup);

    /// <summary>
    /// Creates the card payment.
    /// </summary>
    ICreditCardPayment CreateCardPayment(IOrderGroup orderGroup);

    /// <summary>
    /// Creates the tax value.
    /// </summary>
    ITaxValue CreateTaxValue(IOrderGroup orderGroup);
}
```

`IOrderGroupFactory` seems to be simliar with `IOrderFactory` except each method has one `IOrderGroup` parameter. It's not just a replacement for `IOrderFactory`, it's a new factory-provider pattern that allows to plug your custom implementation.

If you want to have your own implemenation of how to create order or shipment. What you can do is to implement your own `IOrderGroupBuilder`. It's more or less `IOrderFactory`, except with two new methods:

```
    /// <summary>
    /// The sort order of the builder.
    /// </summary>
    int SortOrder { get; }

    /// <summary>
    /// The type(s) of order group that this builder will work with.
    /// </summary>
    Type[] ForType { get; }
```

By specifing `ForType` property, you'll register your class to be the handler of the implementations of `ICart`, `IPurchaseOrder` or `IPaymentPlan`. And in case there is already class registered for those implementations, a higher `SortOrder` will take effect.

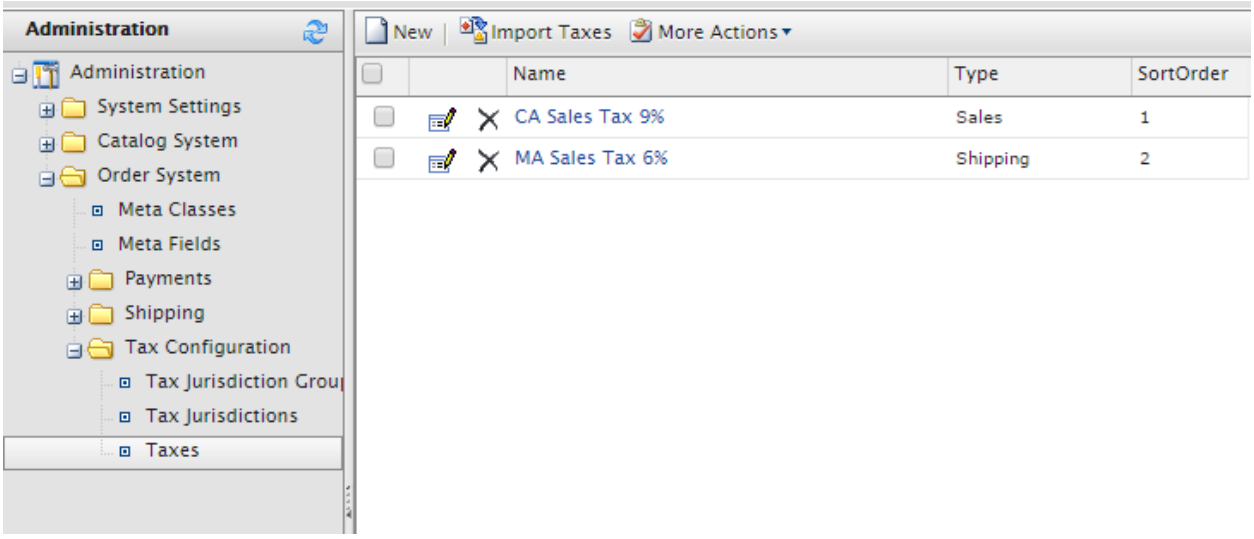
The new cart system will be in BETA state for a while (meaning the API:s can be changed without a major version), until it's matured enough to be out of BETA.

Chapter 8: Taxes

Taxes are essential for every Commerce websites - and you want to get it right. This chapter will be devoted to explain the concepts - how taxes work and how to calculate taxes (correctly).

There are two approach with taxes - some websites precalculated the taxes and set the prices directly. Other sites only calculate taxes at check out. I can't say which approach is better - but the later seems to be more common, and it's a good way to encourage customers to put more items to the cart. Technically, it's also more flexible and easier to implement. So in this book we'll go with it.

Taxes configuration



Tax configuration in Commerce Manager

Configuring taxes manually can be pretty tiring work - especially when you do it for the first time. It would be easier to just import a sample tax, as provided here[37]

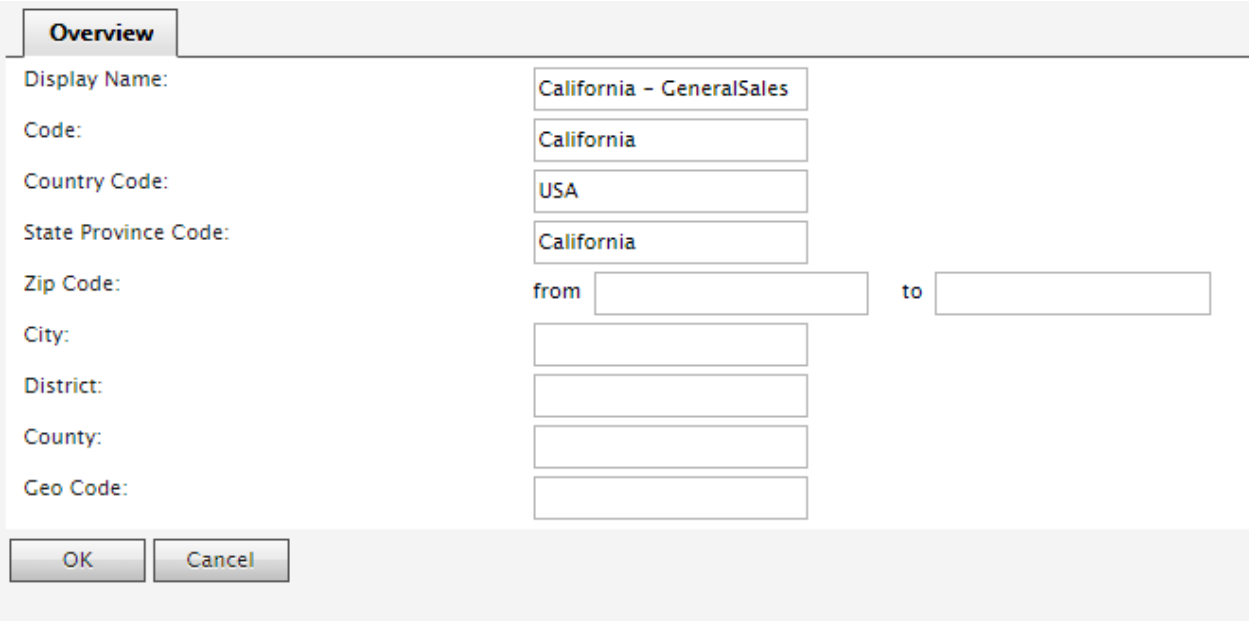
Here, after importing above sample taxes, we come with two taxes - Sales tax for California and Shipping tax for Massachusetts.

The structure of taxes can be confusing at first - so if you are confused, don't worry, it's meant to be complicated.

For each tax, you can configure a tax value for each *Tax Category* and each *Jurisdiction Group*. So if you want to create two taxes for California - one for general sales and one for luxury items, you will

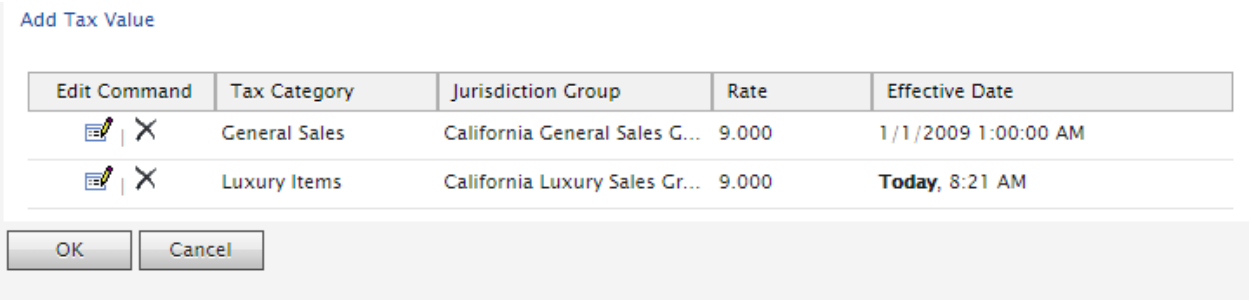
[37] http://docs.mediachase.com/download/attachments/4685836/TaxCSVSample1.csv

have to create two *Jurisdiction Group*. Each of these *Jurisdiction Group* will have one *Jurisdiction* which all point to California state.



Tax jurisdictions specify when a tax applies

Taxes can be set up to city or even zip code. However, the most common scheme is to states (US) or countries (Europe). When a shipment is shipped to an address matching a tax jurisdiction, the tax corresponding with that will be applied. It's important to note that the jurisdiction must match every value in the shipping address. In example above, we have *CountryCode* as "USA" and *State Province Code* as "California" - if the shipping address has *CountryCode* as "US" or *State Province Code* as "CA", those are not matched and of course the tax will not be applied.



Tax value for each Jurisdiction Group

Now with the two sample taxes in place - we need to configure the variations for the *Tax categories*. This can be set in Pricing tab in Commerce Manager or Catalog UI.

Taxes calculation

There are two ways to calculate taxes: using workflows (if you are using the old order API:s and using `ITaxCalculator` (If you are using the abstractions. However, we usually don't use this interface directly, but `IOrderGroupTotalsCalculator`)

To calculate taxes using workflow, `CartPrepare` workflow is used. By default, `CartPrepare` should be called customers reach checkout page. If you don't want to use workflows, you can simply call `IOrderGroupTotalsCalculator.GetTotals`:

```
var totals = _orderGroupTotalsCalculator.GetTotals(cart);
SubTotal = totals.SubTotal,
CartTotal = totals.Total,
ShippingTotal = totals.ShippingTotal,
ShippingSubtotal = cart.GetShippingSubTotal(_shippingCalculator),
OrderDiscountTotal = cart.GetOrderDiscountTotal(cart.Currency, _orderGroupCalcul\
ator),
ShippingDiscountTotal = cart.GetShippingDiscountTotal(_shippingCalculator),
ShippingTaxTotal = totals.ShippingTotal + totals.TaxTotal,
TaxTotal = totals.TaxTotal,
```

Those requirements are needed for the taxes to be calculated and shown correctly to customers:

- Shipping address for each `Shipment` has been set, with all required fields match the information in Jurisdiction.
- `LineItems` have been added to `Shipments` correctly.

Part 4: Events - Listen and react

One of the key features of Episerver Commerce is the ability to integrate into external systems. To do that, parts of Commerce need to communicate, to let other parts know that they are doing something/something have been done. The most common way of doing so is to use events. Thankfully, Episerver Framework already provides a strong mechanism for firing and catching events, so Commerce only has to care about when to raise events and what to raise from it.

Chapter 9: Content events.

As we all know from Part 1, `CatalogContentProvider` is the way Commerce uses to plug the catalog system into the content system, and then it receives (almost) full benefits from the content system - including the events fired when contents are changed. It work pretty well in `IContentRepository` level, as you can simply inject `IContentEvents` into your class and use it there. Let's assume you want to check everytime a content is being moved via Catalog UI, you want to make sure the destination is allowed for current user[38], if not, you will cancel the operation, you can have something like this:

```
using EPiServer;
using EPiServer.Commerce.Catalog.ContentTypes;
using EPiServer.Core;
using System;

public class CatalogContentEventProcess : IDisposable
{
    private readonly IContentEvents _contentEvents;
    public CatalogContentEventProcess(IContentEvents contentEvents)
    {
        _contentEvents = contentEvents;
        _contentEvents.MovingContent += Content_Moving;
    }

    private void Content_Moving(object sender, ContentEventArgs e)
    {
        if(e.Content is CatalogContentBase)
        {
            //Your code goes here
        }
    }

    public void Dispose()
    {
        //Unregister the event to avoid memory leaks.
        _contentEvents.MovingContent -= Content_Moving;
```

[38] Currently there is no access control list management in catalog system, it's all or nothing: a user might have access to all catalog contents, or nothing. Of course this is a limitation, but it will not be addressed anytime soon. If you want to do some extra check for permissions, you'll have to do it your own.

```
    }
}
```

Deleting and Deleted. Like common event naming convention, the content events comes with pair of events - such as `Updating/Updated`, or `Deleting/Deleted`. By convention, the `Deleting` event is supposed to fire *before* it happens, while the `Deleted` is fired *after*. Just keep in mind that at the point of `Deleted` event, you'll no longer be able to get back the content. So if you want to do something with catalog content being deleted, make sure to use `Deleting` event.

There is no "Waste Basket" concept in `CatalogContentProvider`. So when you delete a catalog content, it's gone for good, it is not moving to Waste Basket as normal CMS contents.

It's convenient to use `IContentEvents`, but there is a limitation. Those events only get fired if you works on `CatalogContentProvider` level (via `IContentRepository`). If you work with lower levels, such as working with `ICatalogSystem` directly without involve the content system, those events will not be fired. In a perfect world, you should only have to care about `IContentEvents`, but we are still far from it. While you are no longer required to work with `ICatalogSystem` anymore, ServiceAPI is still using it internally[39]. ServiceAPI is increasingly popular, and most sites I've seen are using it, or planned to use it. You'll need another way to get it works, that's when `CatalogEventListenerBase` comes into place, and here's what it looks like:

```
public abstract class CatalogEventListenerBase
    {
        public virtual void EntryUpdating(object source, EntryEventArgs args) { }
        public virtual void EntryUpdated(object source, EntryEventArgs args) { }
        public virtual void EntryDeleted(object source, DeletedEntryEventArgs ar\
gs) { }
        public virtual void NodeUpdating(object source, NodeEventArgs args) { }
        public virtual void NodeUpdated(object source, NodeEventArgs args) { }
        public virtual void NodeDeleted(object source, DeletedNodeEventArgs args\
) { }
        public virtual void CatalogUpdating(object source, CatalogEventArgs args\
) { }
        public virtual void CatalogUpdated(object source, CatalogEventArgs args)\
 { }
        public virtual void CatalogDeleted(object source, DeletedCatalogEventArg\
s args) { }
```

[39]There is a plan to switch ServiceAPI to use the `IContent` way, but there is no timeframe for that.

```
        public virtual void CatalogLanguageUpdated(object source, CatalogLanguag\
eEventArgs args) { }
        public virtual void AssociationUpdating(object source, AssociationEventA\
rgs args) { }
        public virtual void AssociationUpdated(object source, AssociationEventAr\
gs args) { }
        public virtual void AssociationDeleted(object source, DeletedAssociation\
EventArgs args) { }
        public virtual void RelationUpdating(object source, RelationEventArgs ar\
gs) { }
        public virtual void RelationUpdated(object source, RelationEventArgs arg\
s) { }
        public virtual void RelationDeleted(object source, DeletedRelationEventA\
rgs args) { }
    }
```

How does it work? At the time your site starts up, `EventContext` in `Mediachase.Commerce.Catalog` will look for all registered instances of `CatalogEventListenerBase`, and then register their methods to its events. When certain is made to catalog system, for example when a `CatalogEntryDto` is saved, a relevant method of `EventContext` will be called, and it will fire the event itself, and the methods registered with that event will be triggered.

Simple, right?

Now let's consider two cases of processing events with `CatalogEventListenerBase`:

The first one is when you listen to the events from the same context (aka local event). For example, you have ServiceAPI running in the context of their site, and every time your Windows service import something via ServiceAPI to your system, you want to listen to them and do some special indexing.

The good news is this is the simple case. You can just create your own implementation of `CatalogEventListenerBase` and override the method you want to listen, then register your class like this:

Note that you don't have to override all the methods, just the ones you want to listen to. Build it, and it should be ready to go.

The second one is a more advanced scenario, you want to have ServiceAPI to run in a separate site, may be because of performance issues or security concerns. But when you import entries and nodes into that ServiceAPI instance, you want your site to listen to those events and act accordingly, probably to do some extra processing (clearing cache, indexing, etc. etc.). You want it to be "just ServiceAPI", so it means simpler to configure and maintance.

In other words, you need to listen to remote events and act.

In such case, `CatalogEventListenerBase` can't help you, at least directly. It is supposed to listen to *local* events, and it has to be in same context when you fire the events. You don't have both.

But luckily, the solution comes from one existing implementation of `CatalogEventListenerBase` - `CatalogEventBroadcaster`. As its name might suggest, this class is used to broadcast events, even remotely.

The following section will be an exercise to work with `CatalogEventBroadcaster`

CatalogEventBroadcaster uses Episerver Framework Events internally, so it might be worth to learn one thing or two about Events. Events use UDP to broadcast the events, with two important data: a GUID to identify the event (So listeners on listen to the events they care about), and a byte array contains the information from the sender. By default, the remote event for listening and broadcasting is disabled, you might need to enable them by un-comment these lines in `web.config`:

```
<services>
  <service name="EPiServer.Events.Remote.EventReplication" behaviorConfiguration\
="DebugServiceBehaviour">
    <endpoint name="RemoteEventServiceEndPoint" contract="EPiServer.Events.Servi\
ceModel.IEventReplication"
        binding="customBinding" bindingConfiguration="RemoteEventsBinding" addre\
ss="soap.udp://239.255.255.19:5000/RemoteEventService" />
  </service>
  <!-- Enable remote service -->
</services>
<client>
  <endpoint name="RemoteEventServiceClientEndPoint" address="soap.udp://239.255.\
255.19:5000/RemoteEventService"
    binding="customBinding" bindingConfiguration="RemoteEventsBinding" contract=\
"EPiServer.Events.ServiceModel.IEventReplication" />
</client>
```

A simple way to check if your setting is correct and the remote events are working, is to edit an entry in Commerce Manager and check if the changes were reflected property in Catalog UI. If it works, then remote events are working and the problem might be in somewhere else. If it does not, then check above setting and try again.

Now get back to the problem we have. `CatalogEventBroadcaster` use `Events` to raise the event remotely, so we need to know its event "Id", which is `CatalogEventBroadcaster.CommerceProductUpdated`

The first part is registering a method to listen to this event in your class. Note that this class can be an ordinary class - it does not have to inherit from `CatalogEventListenerBase`:

```
public void AddEvent()
{
    var catalogEventUpdatedEvent = Event.Get(CatalogEventBroadcaster.CommercePro\
ductUpdated);
    catalogEventUpdatedEvent.Raised += CatalogEventUpdated;
}
```

You'll need to call this method during your site initialization (Strictly speaking, you don't have to, you can register when you really want to process the event. But in most of the cases, you should register when your site starts.), there are several ways to do that, but's here the pattern I usually follow:

```
//Inside an IInitializationModule
public void Initialize(InitializationEngine context)
{
    var listener = context.Locate.Advanced.GetInstance<CatalogContentEventListen\
er>();
    listener.AddEvent();
}
```

It's also important to de-register your method from the event to avoid possible memory leaks, so let's create another method to do that:

```
public void RemoveEvent()
{
    var catalogEventUpdatedEvent = Event.Get(CatalogEventBroadcaster.CommercePro\
ductUpdated);
    catalogEventUpdatedEvent.Raised -= CatalogEventUpdated;
}
```

And this method should be called in Uninitialize:

```
//Inside your IInitializationModule
public void Uninitialize(InitializationEngine context)
{
    _listener.RemoveEvent();
}
```

The remaining work is to implement `CatalogEventUpdated` method, which should have a signature like this:

```
private void CatalogEventUpdated(object sender, EventNotificationEventArgs e)
```

The information carried with the `CatalogEventBroadcaster` events (stored in `EventNotificationEventArgs.Param`) are `byte[]` contain serialized data, so let's deserialize it back to object:

```
private static EventArgs DeSerialize(byte[] buffer)
{
    var formatter = new BinaryFormatter();
    using (var stream = new MemoryStream(buffer))
    {
        return formatter.Deserialize(stream) as EventArgs;
    }
}
```

Now depends on the events you want to listen, you'll have to cast `EventArgs` returned by `DeSerialize` to `CatalogContentUpdateEventArgs`, and check for its `EventType` property, which is a string.

These are the events you can listen to:

```
CatalogUpdatingEventType = "CatalogUpdating";

CatalogUpdatedEventType = "CatalogUpdated";

CatalogDeletedEventType = "CatalogDeleted";

CatalogLanguageUpdatedEventType = "CatalogLanguageUpdated";

CatalogNodeUpdatingEventType = "CatalogNodeUpdating";

CatalogNodeUpdatedEventType = "CatalogNodeUpdated";

CatalogNodeDeletedEventType = "CatalogNodeDeleted";

CatalogEntryUpdatingEventType = "CatalogEntryUpdating";

CatalogEntryUpdatedEventType = "CatalogEntryUpdated";

CatalogEntryDeletedEventType = "CatalogEntryDeleted";

AssociationUpdatingEventType = "AssociationUpdating";

```

```
AssociationUpdatedEventType = "AssociationUpdated";

AssociationDeletedEventType = "AssociationDeleted";

RelationUpdatingEventType = "RelationUpdating";

RelationUpdatedEventType = "RelationUpdated";

RelationDeletedEventType = "RelationDeleted";

LanguageConfigurationChangedEventType = "LanguageConfigurationChanged";
```

There are "only" 17 events - just make sure you only process only events you need to. Listening on events which you won't actually process will have negative impact on the performance.

These properties contains the real data, depends on which event you're listening to, they might be empty or not.

```
/// <summary>
/// Gets or sets the list of catalog Id.
/// </summary>
public IEnumerable<int> CatalogIds { get; set; }

/// <summary>
/// Gets or sets the list of catalog node Id.
/// </summary>
public IEnumerable<int> CatalogNodeIds { get; set; }

/// <summary>
/// Gets or sets the list of catalog entry Id.
/// </summary>
public IEnumerable<int> CatalogEntryIds { get; set; }

/// <summary>
/// Gets or sets the list of catalog association id.
/// </summary>
public IEnumerable<int> CatalogAssociationIds { get; set; }
```

You might argue that the `BinaryFormatter` is slow to use in Serialization and Deserialization. You might want to use something faster - but that does mean you'll need to implement your own `CatalogEventBroadcaster`.

Chapter 10: Inventories, Prices and Orders events

Inventories and prices events

Inventories and prices events were added quite recently, in Commerce 8.15 and was slightly changed in Commerce 8.16. (It's not that far away at this time of writing, the current released version of Commerce is 9.19. It's just we're moving fast.) While inventories are changed frequently and are less likely to get benefit from these changes, price events can be really useful - you can index an entry when its prices are changed. That means when you update prices by other means than in Commerce Manager, your entries will be indexed and your facets will be upated. Inventories and Prices events share many common characteristics. They are both based on `CatalogKeyEventBroadcaster`. The idea is the same with `CatalogEventBroadcaster` - when a change is made to inventory or price system, the implementations of those systems will fire an event. Listeners will have to listen to that event and react.

Firing events

As we mentioned about, the default implementations of `IPriceService` and `IInventoryStorage` will fire events when changes are made. If you have custom implementations of those interfaces - you will have to trigger the events yourself. It's simply calling `CatalogKeyEventBroadcaster` to do the work:

```
public class CatalogKeyEventBroadcaster
{
    /// <summary>
    /// Event raised when one or many prices have been updated.
    /// </summary>
    public event EventHandler<PriceUpdateEventArgs> PriceUpdated;

    /// <summary>
    /// Event raised when an inventory has been updated.
    /// </summary>
    public event EventHandler<InventoryUpdateEventArgs> InventoryUpdated;

    /// <summary>
    /// Raise event when prices or inventories are updated.
```

```
    /// </summary>
    public void RaiseEvent(CatalogKeyEventArgs e)

    /// <summary>
    /// Trigger <see cref="PriceUpdated"/>.
    /// </summary>
    public virtual void OnPriceUpdated(object source, PriceUpdateEventArgs args)

    /// <summary>
    /// Trigger <see cref="InventoryUpdated"/>.
    /// </summary>
    public virtual void OnInventoryUpdated(object source, InventoryUpdateEventAr\
gs args)
}
```

Why `CatalogKey`? As you might know `CatalogKey` is a simple class (with `ApplicationId` and `CatalogEntryCode`) - to identify an entry. If there is anything shared between both prices and inventories - then it is they are indeed attached to entries, so `CatalogKeyEventArgs` will be used to let the listeners know which entries' prices and inventories were updated. `PriceUpdateEventArgs` and `InventoryUpdateEventArgs` extends this class by setting the `Name` of the event.

And then when you need to trigger an event, for example, when prices are updated (aka *after* you implement your logic in `IPriceService.SetCatalogEntryPrices`), simply call this:

```
    var priceUpdateEventArgs = new PriceUpdateEventArgs(catalogKeySet)
    {
        ApplicationHasContentModelTypes = _applicationContext.HasContentModelTyp\
es
    };
    broadcaster.OnPriceUpdated(priceValues, priceUpdateEventArgs);
```

`ApplicationHasContentModelTypes` is a special flag to let the listener know if the sending application has the content types or not. FindCommerce relies on this flag to decide if it should reindex the content. If the current application has the custom content types (the content types you inherit from builtin ones), then this should be true, otherwise it's false.

It's worth noting that a change made by `IPriceDetailService` does not guarantee that an event will be fired. The default implementation of `IPriceDetailService` will not trigger an event in case the default implementation of `IPriceService` is marked as `ReadOnly`. Only prices make their way to `IPriceService` will trigger an event.

Listening to events

Most of the cases - you will not have to fire events, but listen to them. We can simply follow the pattern in previous chapter, but now with new `EventId`:

```
var catalogKeyEventGuid = Event.Get(CatalogKeyEventBroadcaster.CatalogKeyEventGu\
id);
catalogKeyEventGuid.Raised += CatalogKeyEventUpdated;
```

> It's worth reminding that this is for registering the event - which you should call in the the `IInitializableModule.Initialize(InitializationEngine context)`. You always should de-register the event in `Uninitialize(InitializationEngine context)`.

Now it's time to implement `CatalogKeyEventUpdated` - we already know how its signature should look like:

```
private void CatalogKeyEventUpdated(object sender, EventNotificationEventArgs e)
{
    var eventArgs = (CatalogKeyEventArgs)DeSerialize((byte[])e.Param);
    var priceUpdatedEventArgs = eventArgs as PriceUpdateEventArgs;
    if (priceUpdatedEventArgs != null)
    {
        RemotePriceUpdated(sender, priceUpdatedEventArgs);
        return;
    }
    var inventoryUpdatedEventArgs = eventArgs as InventoryUpdateEventArgs;
    if (inventoryUpdatedEventArgs != null)
    {
        RemoteInventoryUpdated(sender, inventoryUpdatedEventArgs);
    }
}
```

Let's try to do something useful in this case: reindex the entries which have updated prices - this is important because once the prices are updated, the facets should be updated, too.

```
private void RemotePriceUpdated(object sender, PriceUpdateEventArgs priceUpdated\
EventArgs)
{
    var searchManager = new SearchManager(AppContext.Current.ApplicationName);
    var itemIds = _referenceConverter.GetContentLinks(priceUpdatedEventArgs.Cata\
logKeys.Select(c=>c.CatalogEntryCode), CatalogContentType.CatalogEntry);
    searchManager.UpdateIndex(itemIds);
}
```

Orders events

There are also certain events to listen and react when an order is changed - either updated or deleted. Those events are in `OrderContext` class and are defined like this:

```
/// <summary>
/// Occurs when an order group is updated.
/// </summary>
public event EventHandler<OrderGroupEventArgs> OrderGroupUpdated;

/// <summary>
/// Occurs when an order group is deleted.
/// </summary>
public event EventHandler<OrderGroupEventArgs> OrderGroupDeleted;
```

So if you want to do some extra processing when an order is updated, you can register your listener in an initialization module:

```
//Register to the event:
OrderContext.OrderGroupUpdated += OrderGroupUpdated;

//Listen to the event
private void OrderGroupUpdated(object sender, OrderGroupEventArgs e)
```

Those events are for `OrderGroup` - when an `OrderGroup` is changed - via `AcceptChanges()`, the corresponding event will be fired. If you recall from previous chapter, an `OrderGroup` can be a `PurchaseOrder`, a `PaymentPlan`, or a `Cart`. So if you want to listen to changes made to `PurchaseOrder`:s only, you will need a little casting:

```
//Listen to the event
private void OrderGroupUpdated(object sender, OrderGroupEventArgs e)
{
    var po = sender as PurchaseOrder;
    if (po != null)
    {
        // do your stuffs here.
    }
}
```

It's worth noting that the orders events, unlike the events we talked about, are local-only. So if you have multiple sites and an order is changed in one site, then the listeners will not be called in other sites. However, I don't think there is a need for you to fire events in one site and process in another one.

Part 5: Search

Search has become more and more important part of any software system. When the data is big enough, search is essential, you and your customers don't want to, and can't, go through lists after lists to find the object you want. In this chapter, we will explore the searching options Episerver Commerce provides, and how can you use it in your website implementation.

Chapter 11: Search

Searching for products.

SearchProviders

Many important features in Commerce Manager use index-based search provider to search for entries - it's simply faster and more flexible to work with. They might not be truly "real-time", but they're close (with the right configuration). The only drawbacks are they require extra processing (for indexing entries), external files/configurations, but the advantages easily overweight any of those drawbacks.

There are three search providers provided by Episerver Commerce: LuceneSearchProvider, Solr35SearchProvider, and FindSearchProvider (four if you count SolrSearchProvider as well). LuceneSearchProvider comes as the default option with EPiServer.Commerce.Core, and it's the only one does not require extra configurations.

LuceneSearchProvider works pretty well to some points, but after that, it'll struggle when the catalog becomes big enough, or when it has to process high number of queries at the same time. When that happens, it might be the time for you to consider more suitable search providers - Solr35SearchProvider, FindSearchProvider, or even better, Find Commerce [40]

Where SearchProvider is used.

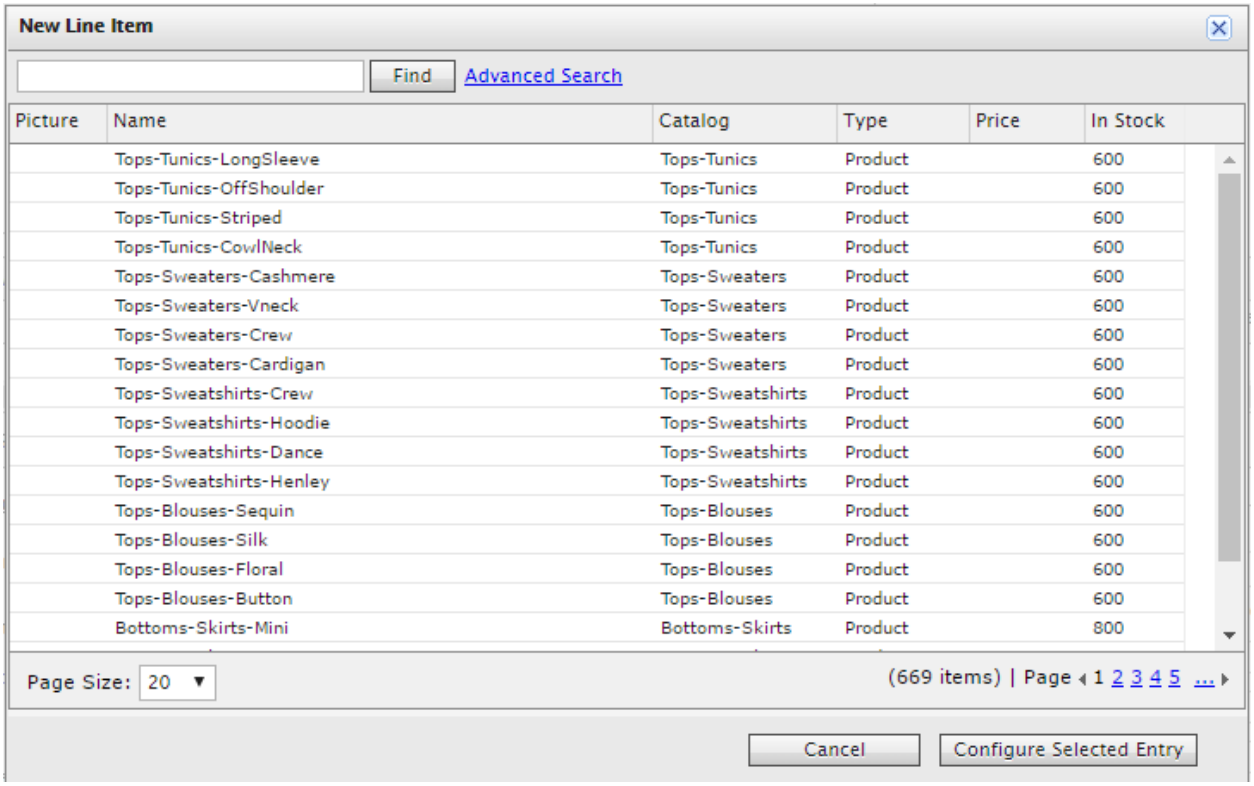
It can be confusing to understand where the SearchProvider is used or how does it distinct from EPiServer Search? It can be easier to grasp the idea if you remember that SearchProvider concept came from a Commerce (Mediachase eCF) background. Despite of a big effort Episerver have tried to integrate Commerce into CMS, they are still two sub-systems working together at this time of writing (and will be, at least in near future).

- EPiServer Search uses LuceneSearch internally and works with CMS pages. In case you use EPiServer Find, it uses Find internally.
- SearchProvider works only with catalog entries.

Then where SearchProvider is used? - Commerce Manager: As mentioned above, Commerce Manager uses `SearchProvider` in almost search-related functionalities, either when you search to

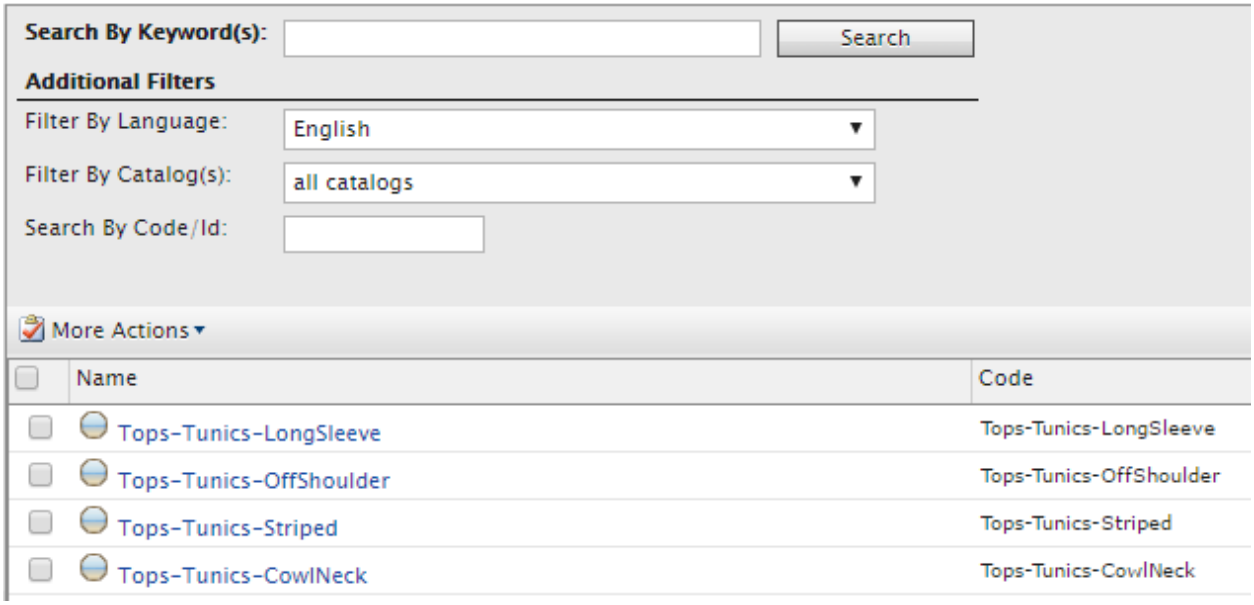
[40] Strictly speaking, Find Commerce does not fully replace a search provider, because it can not be used in Commerce Manager. However, it's usually the front end site which has to manage the high load of queries. You can still use LuceneSearchProvider in Commerce Manager.

edit an entry, or when you add a lineitem to an order, or when you set an entry to be a gift in a promotion.



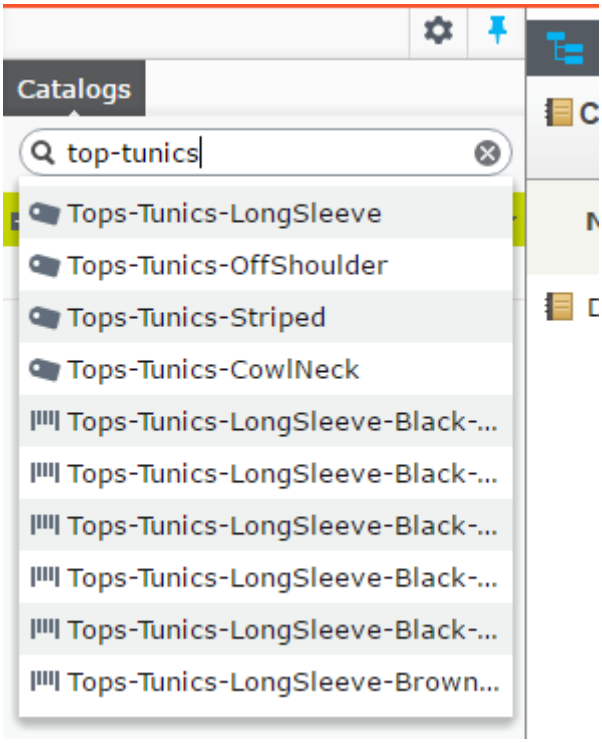
New Line Item dialog - powered by SearchProvider

There is still one exception: Search by code or id in Catalog Management use SQL search internally.



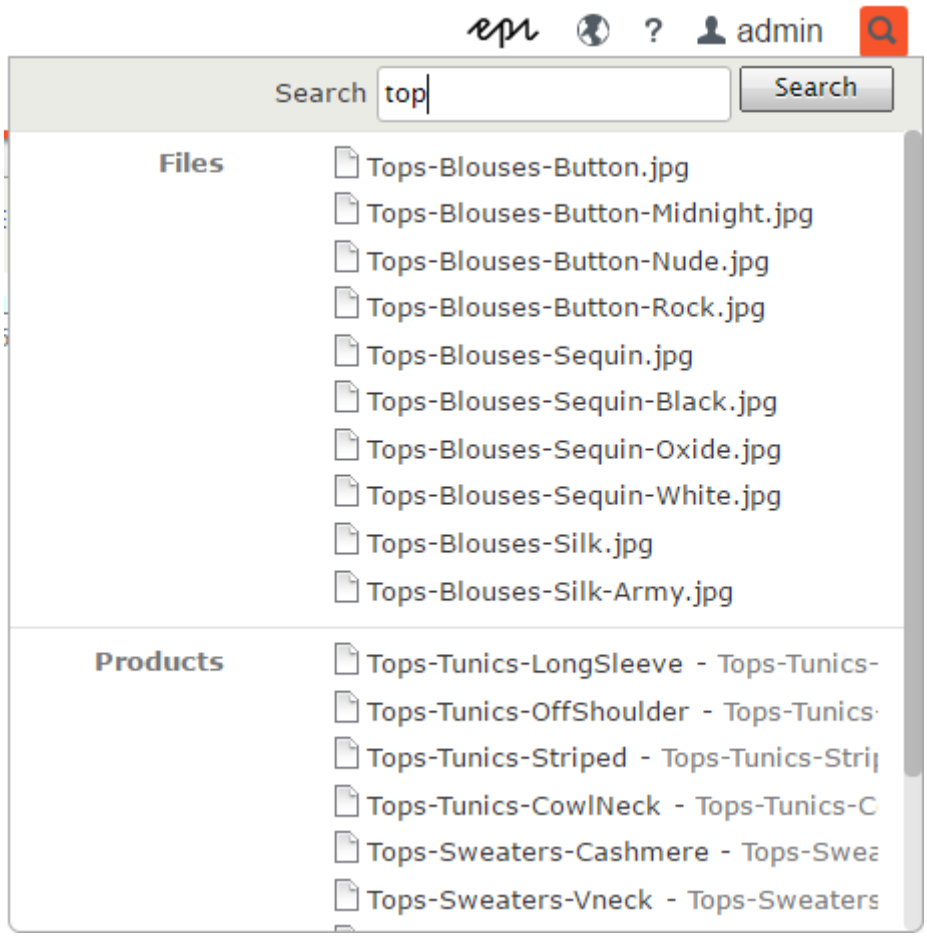
Search By Code/Id is the only place which *does not* use SearchProvider

- Front end: Catalog UI uses `SearchProvider` in the quick search box:



Search box use SearchProvider, but returns all entries including inactive

And Global search uses `SearchProvider` as well:



Global search, search results from SearchProvider are combined with results from other searches

The search in view-mode, is of course, depends on your choice. Most of the implementations will go with Search API:s (which use `SearchProvider` internally), sites with FindCommerce usually use the functionalities provided by Find.

What happens if you use Find? `SearchProvider` still works separately, even if you use `FindSearchProvider` - `FindSearchProvider` uses Find internally for storing and querying the indices, but in "its own way", not the "strongly typed contents" as FindCommerce does. The indices for Find and for `FindSearchProvider` will still be separated, even if you share the same "serviceIndex" from Episerver.

Use Search API:s

In its simplest form, searching using search API:s can be easy like this:

```
CatalogEntrySearchCriteria criteria = new CatalogEntrySearchCriteria();
criteria.SearchPhrase = "panasonic";
SearchManager manager = new SearchManager(AppContext.Current.ApplicationName);
ISearchResults results = manager.Search(criteria);
var ids = results.GetKeyFieldValues<int>()
```

We're searching for the entries with the keyword "panasonic", and getting a list of Id:s for matching entries. From this ids, you can get the list of contents. Still remember `ReferenceConverter` and `IContentRepository`?

Note that with the code above, any entry with a "panasonic" keyword in one of the indexed fields will match, no matter it's in the name, the code, the display name or the description (as long as the fields are marked as "indexable").

Let's explore more complex scenarios - how much can we search with `CatalogEntrySearchCriteria` - it is not a simple class, so we will go through most important properties, one by one.

- `CatalogNames`: Limits the catalogs you want to search in. If you have multiple catalogs, this can be used. Let's reuse the example above. You might have a catalog for consumer products and a catalog for professional products - and their names are respectively "Consumer" and "Professional", if you want to limit the search for entries in "Consumer" catalog only:

```
criteria.CatalogNames.Add("Consumer");
```

- `CatalogNodes`: Limits the catalog nodes you want to search in. If you want to limit the search for entries only in TV and Bluray nodes:

```
criteria.CatalogNodes.Add("TV");
criteria.CatalogNodes.Add("Bluray");
```

However, you can use Outlines as an alternative

- `Outlines`: The "paths" you want to search in, in form of CatalogNameCatalogNodes

```
criteria.Outlines.Add("Consumer\TV\OLED");
```

- `ClassTypes`: The types of entries you want to search for (Product, Variation, Package or Bundle). For example, to search for Product only:

```
criteria.ClassTypes.Add("Product");
```

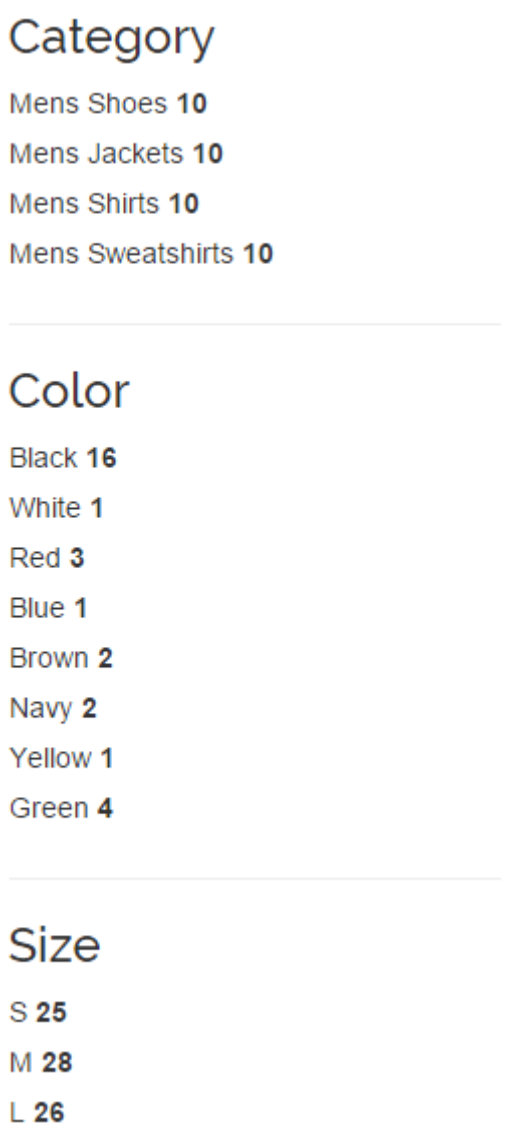
All properties about are of type `StringCollection` - and they mean “restrictions”. Leave them empty mean you don’t want to filter by that criteria. Note that you can add multiple criteria to one property. For example, you can filter by both “Product” *and* “Variation”:

```
criteria.ClassTypes.Add("Product");
criteria.ClassTypes.Add("Variation");
```

Those settings are useful, but nothing particular interesting - but the next one might be

SearchFilter / Facets

Facet or faceted search is an important feature of any eCommerce websites - it allow your customers to quickly filter the products based on some criteria when they browse your product listing. For example, here a very simple facet by Quicksilver



It might not be the best facet in the world, but it gives you the idea

There is more than 1 way to create faceted searching, but the most easy, flexible and powerfulway, is probably relying on the search functionalities. That means, your product listing page is simply a search-result display page (pretty common pattern, you know). The facets in Commerce Search API:s are done by `SearchFilter`. The idea is pretty simple - you define the filters you want your customers to see, and for every search, the search results will include the number of matching products for each filter. When the customer selects a "facet", you'll include the filter mapped with that in the search to limit the scope.

Here's how facets are done in Episerver Commerce:

- You define filters you would like to have in Mediachase.Search.Filters.Config, here's an

example of a filter for field = “size”:

```
<SearchConfig version="1.0">
    <SearchFilters>
    <Filter field="size">
        <Descriptions defaultLocale="en">
            <Description locale="en">Size</Description>
            <Description locale="sv">Storlek</Description>
        </Descriptions>
        <Values>
            <SimpleValue key="s" value="s">
            <Descriptions defaultLocale="en">
                <Description locale="en">S</Description>
            </Descriptions>
            </SimpleValue>
            <SimpleValue key="m" value="m">
            <Descriptions defaultLocale="en">
                <Description locale="en">M</Description>
            </Descriptions>
            </SimpleValue>
            <SimpleValue key="l" value="l">
            <Descriptions defaultLocale="en">
                <Description locale="en">L</Description>
            </Descriptions>
            </SimpleValue>
        </Values>
        </Filter>
    </SearchFilters>
</SearchConfig>
```

Pretty simple, right? You define a filter for the field “size” - which is a *searchable* metafield. You then define the value you want to create filters for, in this case, “s”, “m”, and “l” as three possible values for “size”. Of course, depends on your catalog, you might add the value for “xxs” or “XL”. There are three type of values you can configure for the filters:

- The simple value as we see above.
- The price range value:

```
<PriceRangeValue key="100-200" currency="USD" lowerbound="100" upperbound="200\
" lowerboundincluded="false" upperboundincluded="true">
    <Descriptions defaultLocale="en">
        <Description locale="en">$100-$200</Description>
        <Description locale="de">100â‚¬-200â‚¬</Description>
    </Descriptions>
</PriceRangeValue>
```

(How prices are indexed will be discussed later, when we talk about implementing customizing the search indexer.)

- The range value, basically everything else which is "numberic", butprice:

```
<RangeValue key="under-1.5" upperbound="1.5">
<Descriptions defaultLocale="en">
    <Description locale="en">1.5 in &amp; Under</Description>
    <Description locale="de">1.5 in &amp; Under</Description>
</Descriptions>
</RangeValue>
```

The `SearchFilters` configuration, in fact, is not required - you can define your `SearchFilter`(s) by code, however. `SearchFilterHelper` in Mediachase.Commerce.Website will automatically deserialize this information for you, and having these information in a separated config file should make the configuration much more easier, without having to recompile your code.

It's worth spending some times to consider the facets you will show to your customers. I've seen a site (Luckily, not using Episerver Commerce), to list almost every size of the smartphones they have: 1.2", 1.7", 2.3", 3.0", 3.5", 3.7", 4", etc etc. I consider those to be a bad facet configuration. Nowadays, you should only have something like "Less than 4"", "From 4"-5.1"", "From 5.2" to 6"", and "More than 6"" (more or less, you can argue about the exact numbers, but you get the idea). As a developer, you might not be the one to decide which facets will be, but if you see some bad facets like above, make sure to bring some good arguments and make your facets better.

All the `SearchFilter`(s) configured will be available in

```
SearchFilterHelper.Current.SearchConfig.SearchFilters
```

At first, we can try to add all the filters:

```
foreach (SearchFilter filter in SearchConfig.SearchFilters)
{
    criteria.Add(filter);
}
```

This signals the search provider to return the matching count of all facets, and you can display them in your fancy shiny website. When a customer select a facet (which is commonly store in the URI QueryString), we can try to include that filter into the search. First, we need to figure out which facets have been selected:

```
var filters = new List<SelectedFilter>();
foreach (string key in Request.QueryString.AllKeys)
{
    foreach (var filter in SearchFilterHelper.Current.SearchConfig.SearchFilters)
    {
        //Get the values from QueryString, here we are assuming | is the separat\
or
        var values = Request.QueryString[filter.field].Split('|');
        foreach (var value in values)
        {
            if (filter.Values.SimpleValue != null)
            {
                var filterValue = filter.Values.SimpleValue
                    .FirstOrDefault(x => x.value.Equals(value, StringComparison.\
OrdinalIgnoreCase));
                if (filterValue != null)
                {
                    filters.Add(new SelectedFilter(filter, filterValue));
                }
            }

            if (filter.Values.RangeValue != null)
            {
                var filterValue = filter.Values.RangeValue
                    .FirstOrDefault(x => x.key.Equals(value, StringComparison.Or\
dinalIgnoreCase));
                if (filterValue != null)
                {
                    filters.Add(new SelectedFilter(filter, filterValue));
                }
            }

```

```
            if (filter.Values.PriceRangeValue != null)
            {
                var filterValue = filter.Values.PriceRangeValue
                    .FirstOrDefault(x => x.key.Equals(value, StringComparison.Or\
dinalIgnoreCase)
                    && x.currency.Equals(SiteContext.Current.Currency.CurrencyCo\
de,
                    StringComparison.OrdinalIgnoreCase));
                if (filterValue != null)
                {
                    filters.Add(new SelectedFilter(filter, filterValue));
                }
            }
        }
    }
}
return filters;
```

The idea is simpler than the code looks (the nested `foreach`:es and `if`:s make it looks worse) we iterate over all the keys in `QueryString` (split by "|", you can of course choose another delimiter):

- If a key is found to be a `filter.field`, we get its value and split it into values(Assuming you want to allow multiple facets selection, for example, filter for both "red" *and* "blue").
- If a value matches one of the Values of the filter, we add a new `SelectedFilter` to the list. Depends on the type of `filter.Values`:
  - `SimpleValue`, the value must be matching a simple field.
  - `RangeValue`, the value must be within the selected range, such as size.
  - `PriceRangeValue`, the price (in the currently selected currency) must be within the selected range.
- After all the keys were tested, we return a list of selected filters.

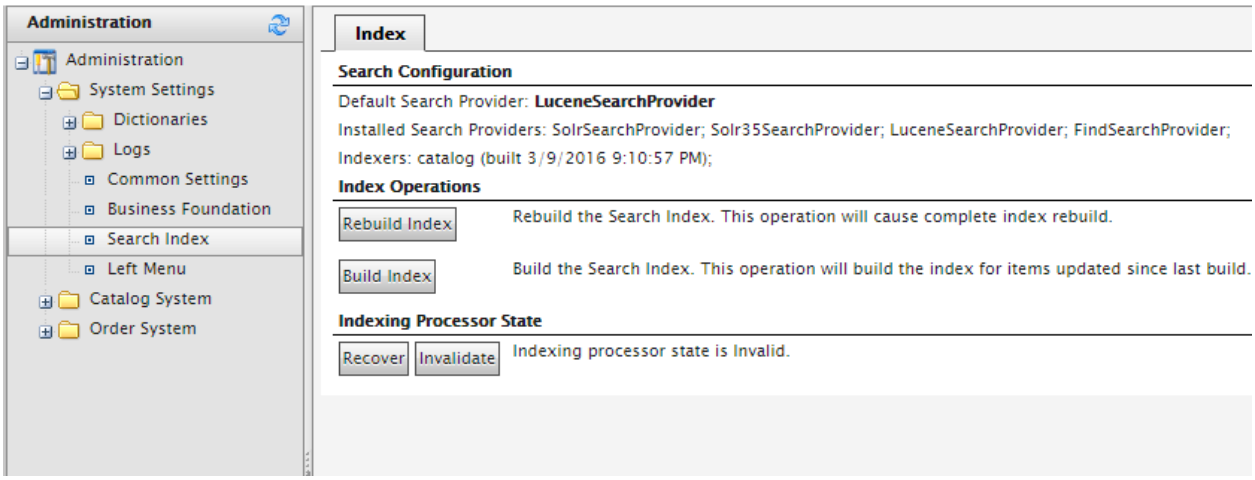
The last step is to include the selected filters in your search criteria, to filter the products which customer wanted to see:

```
foreach (var filter in filters)
{
    if (filter.PriceRangeValue != null)
    {
        criteria.Add(filter.Filter.field, filter.PriceRangeValue);
    }
    if (filter.RangeValue != null)
    {
        criteria.Add(filter.Filter.field, filter.RangeValue);
    }
    if (filter.SimpleValue != null)
    {
        criteria.Add(filter.Filter.field, filter.SimpleValue);
    }
}
```

It's easy after all, right?

Search provider internals.

Prior to Commerce R3a, catalog entries must be indexed manually. That means, everytime you edit an entry, you must trigger an index build, so the newly updated fields will be search-able. Commerce R3a came with a feature allow the indexing to be fully automated - you are no longer required to go to Search Index and click the Build Index button manually.



Search Index UI in Commerce Manager. However, the status of the processor is jokingly called "The most mysterous feature in Commerce Manager"

The indexer handle all three scenarios:

- Rebuild the index entirely: which does mean all entries will be re-indexed, regardless if they have been changed since the last indexing time. This is an expensive operation, of course, and

you should not use it for a running production site. However, it might make senses to do that when you deploy your site, when the content types (and the the metaclasses) of your entries might have changed (metafields have been added or removed).

- Build the index: `SearchProvider` system keeps track of the last time it indexed the entries, and it will try to index the entries which have been updated since. This is the prefered option as it's reasonable cheap, however, it should not be needed anymore, if you enable...
- Index by event: The indexer will try to index everytime an entry is updated [41]. This was added in Commerce R3, and it has been working pretty well, except when it's not. We'll dive deep to see how it works and how to make it works a bit later.

SearchProviders are configured by Mediachase.Search.config, in Configs folder of both front-end and Commerce Manager site, here's an example of how it looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<Mediachase.Search>
  <SearchProviders defaultProvider="LuceneSearchProvider">
    <providers>
      <add name="Solr35SearchProvider" type="Mediachase.Search.Providers.Solr35.\
SolrSearchProvider, Mediachase.Search.Solr35SearchProvider" queryBuilderType="Me\
diachase.Search.Providers.Solr35.SolrSearchQueryBuilder, Mediachase.Search.Solr3\
5SearchProvider" url="http://localhost:8080/solr" shareCores="true" facetLocaliz\
edFieldValuesOnly="true" commitWithin="10000" maximumBatchSize="50" />

      <add name="LuceneSearchProvider" type="Mediachase.Search.Providers.Lucene.\
LuceneSearchProvider, Mediachase.Search.LuceneSearchProvider" queryBuilderType="\
Mediachase.Search.Providers.Lucene.LuceneSearchQueryBuilder, Mediachase.Search.L\
uceneSearchProvider" storage="<path-to-index-folder>" simulateFaceting="true" />

      <add name="FindSearchProvider" type="EPiServer.Commerce.FindSearchProvider\
.FindSearchProvider, EPiServer.Commerce.FindSearchProvider" serviceUrl="http://<\
find-index-url>" defaultIndex="myindex" />
    </providers>
    </SearchProviders>
   <Indexers basePath="<path-to-index-folder>">
    <add name="catalog" type="Mediachase.Search.Extensions.Indexers.CatalogIndex\
Builder, Mediachase.Search.Extensions" />
  </Indexers>
</Mediachase.Search>
```

There are a couple of things you want to care about:

[41] Actually, the updated entries, the entries belong to updated nodes and the entries with updated node-entry relations are all re-indexed. Note that the index of one entry, once created, cannot be partly updated - it needs to be reindexed entirely.

- The `defaultProvider` - this is the search provider you're going to use. The required attribute is the type, because Mediachase.Search needs to create an instance of it, other attributes are used by the search provider. The type must inherit from Mediachase.Search.SearchProvider.
- The `basePath` - this is the path to store the indices, it's only used by file-based indexers, in most of the case cases, LuceneSearchProvider. HTTP-based indexers, such as Solr35SearchProvider or FindSearchProvider, do not care about this. There one thing to remember if you are using file-based indexers: make sure the paths in both Mediachase.Search.config point to same directory, and both AppPool for both sites have right access to this folder.
- The type of Indexer. This is where you register how your providers index the entries, it must implement `ISearchIndexBuilder` interface. For `LuceneSearchProvider` and `Solr35SearchProvider`, it's `CatalogIndexBuilder`, which inherits from `BaseCatalogIndexBuilder`. During the indexing, `BaseCatalogIndexBuilder` will call to:

```
protected virtual int IndexCatalogEntryDto(IndexBuilder indexer, CatalogEntryDto\
.CatalogEntryRow entryRow, string defaultCurrency, string[] languages)
```

This method creates an `ISearchDocument` from the `entryRow`, and then finally call to this method in the provider:

```
public override void Index(string applicationName, string scope, ISearchDocument\
 document)
```

`FindSearchProvider` works a bit different, it registers `FindSearchIndexBuilder`, which overrides `IndexCatalogEntryDto` directly to create `ISearchDocument` in its own way, but at the end it also calls to `SearchProvider.Index` to do the index.

Implements a search provider yourself:

The search system allows you to create your own search provider. There are three steps you can do:

- Implement your own search provider. This can be done by creating a class inherits from `Mediachase.Search.SearchProvider`:

```
    public abstract class SearchProvider : ProviderBase
    {
        /// <summary>
        /// Gets the class type of the query builder. This class will be used to\
 dynamically convert SearchCriteria to the query
        /// that Search Provider can understand.
        /// </summary>
        public abstract string QueryBuilderType { get; }

        /// <summary>
        /// Searches the datasource using the specified criteria.
        /// </summary>
        public abstract ISearchResults Search(string applicationName, ISearchCri\
teria criteria);

        /// <summary>
        /// Adds the document to the index. Depending on the provider, the docum\
ent will be commited only after commit is called.
        /// </summary>
        public abstract void Index(string applicationName, string scope, ISearch\
Document document);

        /// <summary>
        /// Removes the document by specifying scope (core in SOLR), key (a fiel\
d that can be used to lookup for a document) and
        /// value of the key.
        /// </summary>
        public abstract int Remove(string applicationName, string scope, string \
key, string value);

        /// <summary>
        /// Removes all documents in the specified scope.
        /// </summary>
        public abstract void RemoveAll(string applicationName, string scope);

        /// <summary>
        /// Indexing has completed, close the connections, release the resources.
        /// </summary>
        public abstract void Close(string applicationName, string scope);

        /// <summary>
        /// Commits changes. For example, save your indices to disk or send them\
```

```
 to server.
        /// </summary>
        public abstract void Commit(string applicationName);
    }
```

All applicationName parameter can be get by `AppContext.Current.ApplicationName` property.

You of course don't have to implement all these methods - you can inhertit from the search provider you want to extend, and only override the method you need to.

- Implement your own `ISearchIndexBuilder` to handle the indexing:

```
    public interface ISearchIndexBuilder
    {
        /// <summary>
        /// An event handler to send message about the progress back to the call\
er.
        /// </summary>
        event SearchIndexHandler SearchIndexMessage;

        /// <summary>
        /// Gets or sets the search manager. This will be set automatically by S\
earchManager itself.
        /// </summary>
        SearchManager Manager { get; set; }

        /// <summary>
        /// Gets or sets the build indexer. This will be set automatically by Se\
archManager itself.
        /// </summary>
        IndexBuilder Indexer { get; set; }

        /// <summary>
        /// Builds the index, if rebuild = true, rebuilds the entire index.
        /// </summary>
        void BuildIndex(bool rebuild);

        /// <summary>
        /// Updates indices of specified entries.
        /// </summary>
        bool UpdateIndex(IEnumerable<int> itemIds);
    }
```

Again, you don't have to override all of these. You can inherit from `BaseCatalogIndexBuilder` and override only `BuildIndex` and `UpdateIndex`. Or you can override only how each entry is indexed, by overriding `IndexCatalogEntryDto`.

- Register your implementations as the defaultSearchProvider and the indexer.

Demystifying the index processor

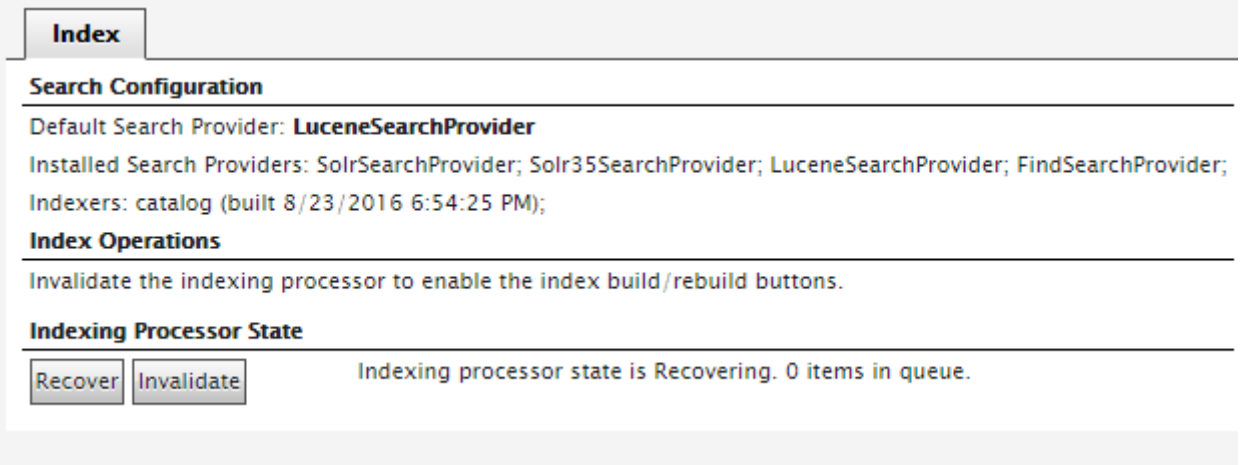
With proper configuration, Episerver Commerce will index the changed entries automatically, without your manual interactions. To allow this, Commerce relies on the `ChangeNotification` concept from Framework. It's a database-backed queue. When one or more entries are updated, Commerce will send those updates to the queue. That also happens when a node is updated, a node-entry relation or a catalog association is updated, all related entries will be sent to the queue.

If you wonder what is actually sent to the queue - it's the code of the entry. As we learned from Part 1, `Code` is unique across the system and can be used to identify the entry.

Meanwhile, Commerce has one class registered to process such change - `CatalogEntryChangeProcessor`. `ChangeNotificationManager` will call `CatalogEntryChangeProcessor` when it finds anything in the queue that `CatalogEntryChangeProcessor` registered to. By default, every 5 seconds (a "heartbeat"), `ChangeNotificationManager` will get remaining items from the queues (It will get the `Code`, then use `ICatalogSystem` to get the `CatalogEntryDto`) and call the processors to process those.

There are several statuses of the processor:

- Valid: the processor is working and it'll update the indices when there are changes. This is of course the status you want to see.
- Invalid: This is the status when you click on Invalidate button. That means you don't want the processor to continue processing the queue. You'll be able to build the index manually.
- Recovering: It's usually when the status of the processor when you click "Recover" when it was "Invalid". The processor is trying to recover consistency - and by that does mean it is rebuilding the index.

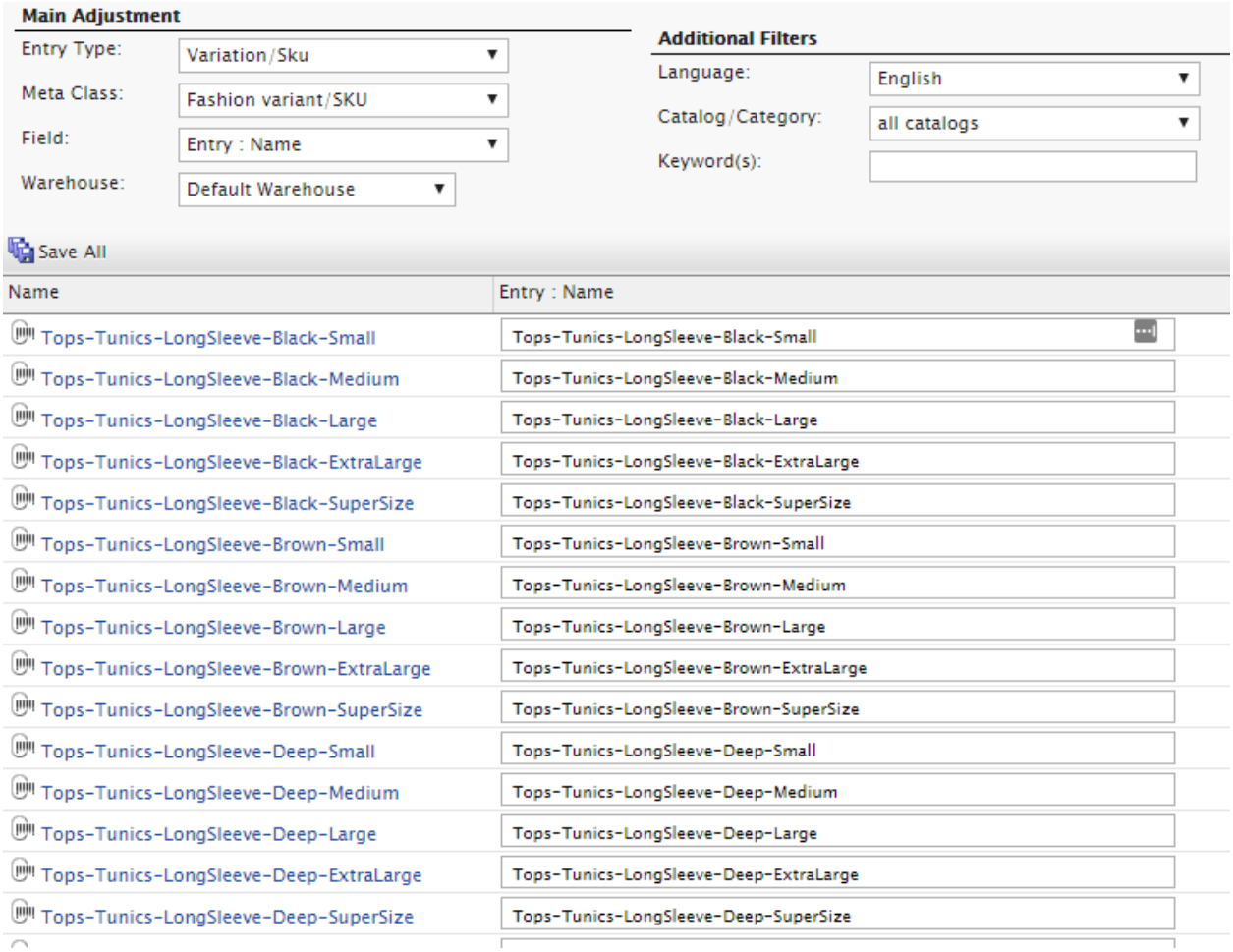


This can take quite long time, depending on your catalogs

- Closed: This is the status you don't want to see - normally it means something went totally wrong, and the processor is no longer to process the queue. Your only option is usually to restart your site.
- Opening: This usually happens at time the site starts, it's so fast that you hardly see the processor in this state.

Search by Search API:s

Episerver Commerce provides a SQL-based API:s for you to search for entries which match a certain criteria. They works in real-time (without any delay like the solution provided by Search Providers), but they's not that fast. Search API:s are mostly used in Commerce Manager, and you can use them for some batching updates, but they definitely should not be used in any customer-facing features. For such features, use Search providers instead.



Catalog Batch Update is one example of how Search API:s can be used

Search API:s are built around `CatalogSearchParameters` and `CatalogSearchOptions`. Here's the simplest example of using Search API:s

```
CatalogSearchOptions options = new CatalogSearchOptions();
CatalogSearchParameters searchParams = new CatalogSearchParameters();
int totalCount = 0;
CatalogEntryDto entryDto = catalogSystem.FindItemsDto(searchParams, options, ref\
 totalCount);
```

`entryDto` will contain every entry in the system - you won't want to use such code in your production system. A more realistic scenario is to iterate over variations, in 200 entries/batch

```
CatalogSearchOptions options = new CatalogSearchOptions();
CatalogSearchParameters searchParams = new CatalogSearchParameters();
int totalCount = 0;
options.Classes.Add("Variant");
//We don't want to get any entry, we only need the totalCount
options.RecordsToRetrieve = 0;
catalogSystem.FindItemsDto(searchParams, options, ref totalCount);
int recordsCount = 0;
while (totalCount > 0 && recordsCount < totalCount)
{
    options.RecordsToRetrieve = 200;
    options.StartingRecord = recordsCount;
    var catalogEntriesDto = CatalogContext.Current.FindItemsDto(searchParams, op\
tions, ref totalCount, new CatalogEntryResponseGroup(CatalogEntryResponseGroup.R\
esponseGroup.Info));
    //Do some processing here
    recordsCount += 200;
}
```

`CatalogSearchOptions` contains settings of how the search should behave - which classes to search for, how many rows should be returned, which is the start row to get, or should the cache result be cached. `CatalogSearchOptions` inherits from `SearchOptions` with the `Namespace` is set to "Mediachase.Commerce.Catalog":

```
public abstract class SearchOptions
    {
        /// <summary>
        /// Filter by the meta class's names.
        /// </summary>
        public StringCollection Classes { get; set; }

        /// <summary>
        /// The maximum number of records to return.
        /// </summary>
        public int RecordsToRetrieve { get; set; }

        /// <summary>
        /// The start record to return.
        /// </summary>
        public int StartingRecord { get; set; }

        /// <summary>
```

```
        /// Indicates if the search should return total count.
        /// </summary>
        public bool ReturnTotalCount { get; set; }

        /// <summary>
        /// Filter by the namespace.
        /// </summary>
        public string Namespace { get; set; }

        /// <summary>
        /// Indicates if the results should be cached.
        /// </summary>
        public bool CacheResults { get; set; }

        /// <summary>
        /// Timeout of the cache.
        /// </summary>
        public TimeSpan CacheTimeout { get; set; }

        ...
    }
```

`CatalogSearchParameters` contains some filter you can apply to your search:

```
public sealed class CatalogSearchParameters : SearchParameters
    {
        /// <summary>
        /// Filter by catalog names.
        /// </summary>
        public StringCollection CatalogNames { get; }

        /// <summary>
        /// Filter by catalog node codes.
        /// </summary>
        public StringCollection CatalogNodes { get; }

        /// <summary>
        /// Filter by Markets, only entries belong to those markets will be retu\
rned.
        /// </summary>
        public StringCollection Markets { get; set; }

```

```
        /// <summary>
        /// Filter by language, only entries available in this language (aka it'\
s one of languages its catalog supports) will be returned.
        /// </summary>
        public string Language { get; set; }

        ...
    }
```

If you recall about `CatalogEntrySearchCriteria`, you can see the similarity here - same properties for filtering results. Personally I think this is a place where Mediachase eCF got it right - the consistency in API:s is very important to make them easy to remember and use. However, the true powers of Search API:s are provided in the bas class - `SearchParameters`.

```
public abstract class SearchParameters
    {
        /// <summary>
        /// Gets or sets the type of the join used in the query.
        /// The example types are: inner join, outer join, left join and right j\
oin.
        /// </summary>
        public string JoinType { get; set; }

        /// <summary>
        /// Gets or sets the join source table. This value has to be a table nam\
e that exists in original
        /// query. For example it will be CatalogEntry for the Catalog Entry Sea\
rch or CatalogNode for the Catalog
        /// Node Search.
        /// </summary>
        public string JoinSourceTable { get; set; }

        /// <summary>
        /// Gets or sets the join target query. This can either be a table name \
or a complete query. For exammple
        /// the table name can be something like CatalogEntryEx for the Catalog \
Entry search. This particular join will allow
        /// sorting by meta fields. This is not possible without a join. In some\
 cases the target table can contain multiple
        /// records for the same record in source table. That will cause problem\
s with constraints that are inforces by the
        /// search DTOs. In order to overcome this, you can specify the query, s\
```

```
o if our CatalogEntryEx table had multiple
        /// records for one CatalogEntry, we would join it using the follow quer\
y:
        /// (select distinct ObjectId, DisplayName from CatalogEntryEx) CatalogE\
ntryEx
        /// </summary>
        /// <value>The join target table.</value>
        [DataMember]
        public string JoinTargetQuery { get; set; }

        /// <summary>
        /// Gets or sets the join source table key.
        /// </summary>
        public string JoinSourceTableKey { get; set; }

        /// <summary>
        /// Gets or sets the join target table key.
        /// </summary>
        public string JoinTargetTableKey { get; set; }

        /// <summary>
        /// Gets or sets the SQL where clause. You can specify filter conditions\
 against the tables returned by the query.
        /// </summary>
        public string SqlWhereClause { get; set; }

        /// <summary>
        /// Gets or sets the SQL meta where clause. You can specify filters for \
meta tables. Make sure to specifiy columns that
        /// exist for items searched.
        /// </summary>
        public string SqlMetaWhereClause { get; set; }

        /// <summary>
        /// Gets or sets the keyword text search phrase.
        /// </summary>
        public string KeywordSearchPhrase { get; set; }

        /// <summary>
        /// Gets or sets the order by clause.
        /// </summary>
        public string OrderByClause { get; set; }
```

```
    }
```

Don't be overwhelmed by these properties. We will start by examining the most commonly used ones for now - `SqlWhereClause` and `SqlMetaWhereClause`. Simply put, `SqlWhereClause` can be used when you want to filter on `CatalogEntry` or `CatalogNode` table, while `SqlMetaWhereClause` can be used when you you want to filter on the metafields.

Let's try to find entries which are going to expire today. We know that the expire date is represent by `EndDate` in `CatalogEntry` table, so we need to have a filter on SqlWhereClause:

```
CatalogSearchOptions options = new CatalogSearchOptions();
CatalogSearchParameters searchParams = new CatalogSearchParameters();
searchParams.SqlWhereClause = " [CatalogEntry].[EndDate] < " + DateTime.UtcNow.T\
oString("yyyy-MM-dd HH:mm:ss");
int totalCount = 0;
CatalogEntryDto entryDto = catalogSystem.FindItemsDto(searchParams, options, ref\
 totalCount);
```

Simple, right? You can certainly extend this to find entries which are going to expire next week or next month.

It's worth noting that when you work with database, you should always use UTC timezone. All `DateTime` values are supposed to be saved as UTC in Commerce database.

Let's take another example - let's say we want to find the entries belong to a specific metaclass. With the `CatalogSearchOptions.Classes` we can filter `Product`, `Variant`, `Package` or `Bundle`, but we might want to go further and filter for `FashionSKU` or `SLR`. Again, `SqlWhereClause` can solve this because `CatalogEntry` has a column named `MetaClassId` which indicates the metaclass of the entry.

```
//Assuming we don't know the Id, but only the name of the MetaClass
MetaClass metaClass = MetaClass.Load(CatalogContext.MetaDataContext, "SLR");
CatalogSearchOptions options = new CatalogSearchOptions();
CatalogSearchParameters searchParams = new CatalogSearchParameters();
searchParams.SqlWhereClause = " [CatalogEntry].[MetaClassId] = " + metaClass.Id.\
ToString();
int totalCount = 0;
CatalogEntryDto entryDto = catalogSystem.FindItemsDto(searchParams, options, ref\
 totalCount);
```

Of course, you can combine those condition in one query - you can find entries belong to "SLR" metaclass *AND* expire next week.

While `SqlWhereClause` is quite straighforward, `SqlMetaWhereClause` can be a bit tricky to work with. The idea is still the same - to filter by SQL. If you recall from how metaclasses are stored in the first chapter, you can see that the tables for metaclasses are dropped in Commerce 9, and replaced by the views. The views act pretty much the same way, but due to their structure, the query will be a bit different.

Let's go through another example: find entries which "Color" property, with type `LongString` has value of "Blue"

```
MetaClass metaClass = MetaClass.Load(CatalogContext.MetaDataContext, "SLR");
CatalogSearchOptions options = new CatalogSearchOptions();
CatalogSearchParameters searchParams = new CatalogSearchParameters();
searchParams.SqlMetaWhereClause = " META.MetaFieldName = 'Color' AND META.LongSt\
ring = 'Blue'";
int totalCount = 0;
CatalogEntryDto entryDto = catalogSystem.FindItemsDto(searchParams, options, ref\
 totalCount);
```

So you need to specify field name, by 'META.MetaFieldName', and field value, by 'META.<field type>'.

`SqlMetaWhereClause` and `SqlWhereClause` are not exclusive - they can be used in conjunction to create really powerful queries which combine from various condition.

It's important to remember that Search API:s do not know about, or work with versioning. They can only work with the published data (which are represented by `CatalogEntry`, `CatalogNode` and the views). Currently there are no API:s to find data from draft versions, and there probably will be no such features.

Find Commerce

Episerver Find, in term of product, is a .NET API:s for ElasticSearch, in term of service, it's an indexing service built on ElasticSearch. After acquiring Truffler in 2012, Episerver has been integrating Episerver Find to their core products, allowing contents to be indexed and searched in a very flexible and powerful way. However, Commerce was lagging behind in this matter, its integration package, Find Commerce, was not as fully featured as Find CMS. Luckily, Find Commerce has been giving a lot of attention recently, and with Find Commerce 9.5 [42], it's now more or less on par with what Find CMS has to offer.

[42] Find Commerce comes with a separated nuget package, EPiServer.Commerce.Find, and it is not released in Commerce "bundle". It has its own version and release schedule.

Find Commerve vs FindSearchProvider: FindSearchProvider is just a wrapper to allow entries to be indexed and searched by Episerver Find. It does not leverage the true power of Find, or content in a "proper way". The advantage of FindSearchProvider is it can be used for older versions of Commerce, including R1 SP2 and R3, and it can be used in Commerce Manager. In terms of features and flexibility, FindSearchProvider can't not compete to Find Commerce, it's not even close. If you are developing a new site, and intend to use Episerver Find, then Find Commerce should be your first choice. FindCommerce can only work on front-end site, however this still makes it a great solution because that's where you have to deal with the most complicated scenarios and demanding loads.

Find Commerce, as Find itself, relies heavily on extension methods and lambda expressions. Are you ready to go?

Find Commerce configurations

Find Commerce comes with many configurations, and most of them can be done by code. If we are to override some configurions, it's best to do that in an `IConfigurableModule`, which has dependency on `FindCommerceInitializationModule`. This way, we can be sure to override the default configuration and dependency registrations.

```
[InitializableModule]
[ModuleDependency(typeof(EPiServer.Find.Commerce.FindCommerceInitializationModul\
e))]
public class MyFindCommerceInitializationModule : IInitializableModule
{
}
```

By default, Find Commerce will reindex the content everytime its prices or it inventories are updated. Inventories can be updated often (especially if your goods are being sold like hot cakes), it might make senses to disable reindexing on inventory updated events. How?

Find Commerce uses `CatalogContentEventListener` to handle pricing and inventory events. This listener will do a check to see if it should index a content when an inventory event triggers - if the content type was configured to index the inventories. So this is how Find Commerce configures that:

```
protected virtual void ApplyIStockPlacementConventions()
{
    SearchClient.Instance.Conventions.ForInstancesOf<IStockPlacement>()
        .ExcludeField(c => c.InventoryReference)
        .IncludeField(x => x.Inventories());
}
```

The "conventions" are built around `SearchClient.Instance.Conventions`. Here we are saying: for contents implement `IStockPlacement`, exclude `InventoryReference`, but include `Inventories()` in the index. If we want to exclude `Inventories` from the index, we need to override it

```
public void Initialize(InitializationEngine context)
{
    SearchClient.Instance.Conventions.ForInstancesOf<IStockPlacement>().ExcludeF\
ield(c => c.Inventories());
}
```

because our `IInitializableModule` runs after `FindCommerceInitializationModule`, this will override the default conventions. Our site no longer indices the contents when an inventory event triggers. Basically, we can also override any default convention with this method.

Now consider another scenario. We want to index the content with the inventory information. But to reduce the overload, we only index if the the SKU has less than 10 items in stock (if it is more than 10, just display something like "10+ in stock"). First, we need to unhook the default event listener in `CatalogContentEventListener`:

```
public void Initialize(InitializationEngine context)
{
    _broadcaster = context.Locate.Advanced.GetInstance<CatalogKeyEventBroadcaste\
r>();
    _listener = context.Locate.Advanced.GetInstance<CatalogContentEventListener>\
();
    _broadcaster.InventoryUpdated -= _listener.InventoryUpdated;
}
```

Then we need to register our custom listener. As we talked in Part 4 - Events, the `sender` contains list of `InventoryRecords`, while `InventoryUpdateEventArgs` contains list of `CatalogKeys`, `Name` of the event and a flag for `ApplicationHasContentModelTypes`. Here we need to check if the remaining stock in all warehouses of a item is less than 10 or not.

```
private void InventoryUpdated(object sender, InventoryUpdateEventArgs args)
{
    var inventoryRecords = sender as IEnumerable<InventoryRecord>;
    if (inventoryRecords == null)
    {
        return;
    }

    var inventoryGroups = inventoryRecords.GroupBy(c => c.CatalogEntryCode);
    foreach (var inventoryGroup in inventoryGroups)
    {
        if (inventoryGroup.Sum(c=>c.PurchaseAvailableQuantity) < 10m)
        {
```

```
            _contentEventIndexer.SavingContent(_referenceConverter.GetContentLin\
k(inventoryGroup.Key, CatalogContentType.CatalogEntry));
        }
    }
}
```

_contentEventIndexer is an instance of ContentEventIndexer, the class used for indexing contents in Find, and that's the simple way to index a content - calling SavingContent with the ContentReference of the content to be indexed. Still remember our little buddy ReferenceConverter?

Search with Find Commerce

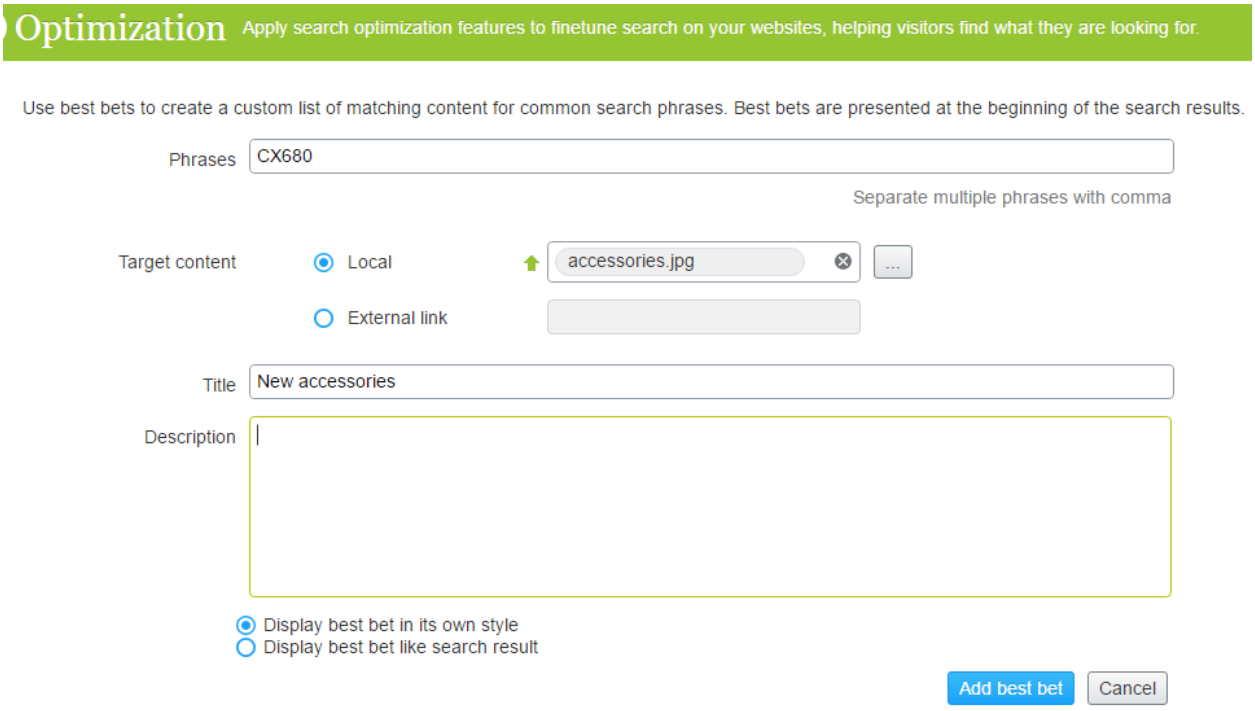
If you have experience with Find already, this should be easy. But if you don't, let's break down a normal search:

```
var variationContentSearch = SearchClient.Instance.Search<VariationContent>()
.For(searchText)
.FilterPriceAvailableForCurrentUser()
.FilterForVisitor()
.Select(x => new
{
    x.ContentLink,
    Inventory = ((IStockPlacement)x).Inventories(),
    Prices = ((IPricing)x).Prices(),

    ImageUrl = x.DefaultImageUrl(),
    x.DisplayName,
    Description = x.Description(),
    x.Code
})
.ApplyBestBets()
.Track()
.Skip(startIndex)
.Take(pageSize)
.GetResult();
```

"Woo woo, wait a minute" - I can hear you. I can understand, this can look scary at first. But not after we go through it one by one:

- We are searching for VariationContent, so no NodeContentBase, or even ProductContent. Find is all about strongly types, so if you want to search for "TV" - and you have a content type for that, simply change the type to "TVContent".

- A matching `searchText` is expected.
- We only search for entries which have prices available for current user.
- We only search for entries which are available for current user. So if an entry is only available in "US" Market, but the current user is in "Europe" Market, the entry will be filtered out.
- We don't want the entire content in the search result, we only want some important fields, the `ContentLink`, prices, inventories, the image, display name, etc. Again, it's about strongly typed types - so if your content has a property named "ScreenSize", and it's indexed, it can be used here.
- We are applying the "BestBets" here. It's a feature in Find that allows you to link a specific content to a keyword. For example, if you know customers often make a typo searching for "ASpecailKeyword", instead of "ASpecialKeyword", you can configure that keyword for a content which should be in the search result. Or if you know customers often search for a keyword that your products do not contain, but you want to promote one anyway.



An example of best bets in Find UI

- We want to track the search. For analytics, of course.
- We are using paging technique here. Note that `Skip` and `Take` here are not LINQ, althought they look the same. Note that Find query and LINQ expressions can't be used in conjunction.
- After building the query, we finally get the result. As most sites use Find as a cloud service (provided by Episerver itself), it's best to make the traffic as small as possile - so the way Find works is to only return what you need.

On your search page, you usually tweak the query above, but the core concept is more or less the same.

Advanced filters

The strongly typed types properties work a bit differently with "complex" (aka nested) types. `Inventory` and `Prices` are example of nested types, and you can have plenty of filtering scenarios for them in a Episerver Commerce site. For example, filtering entries which have prices in the current currency, or have prices less than a specific value. Find Commerce gets you covered - it provides a lot of *filters* for `IPricing` and `IStockPlacement`.

For example, if we want to filter only variations which have (default) prices less than 100, in current currency, the above query can be rewritten like this:

```
var variationContentSearch = SearchClient.Instance.Search<VariationContent>()
.For(searchText)
.FilterPriceAvailableForCurrentUser()
.FilterForVisitor()
.Filter(p=>p.PriceLessThan(100m))
...
```

It's good that Find Commerce has better documentation about its filters than Find itself. It would still take a while for you to discover all the filters, and when to use what, but believe me, it's worth your time and effort.

This section is not supposed to be your complete reference about Find Commerce, but instead a glimpse at it. I hope this crash course made you like Find Commerce - and Find in general. You now can explore it powers to make your coding days a joyful experience.

Searching for orders

Searching for orders shares many characteristics with searching for entries. They even share a large portion of API:s. Order searching API:s are built upon `OrderSearchOptions` and `OrderSearchParameters`, which inherit from `SearchOptions` and `SearchParameters`, respectively, so they have same base classes as `CatalogSearchOptions` and `CatalogSearchParameters`.

`OrderSearchOptions` is almost identical with `CatalogSearchOptions`. The only different is int the `Namespace` property - it limits the search to 'Mediachase.Commerce.Orders'. `OrderSearchParameters` is an even more simple inheritance of `SearchParameters`: it does not have any new properties on its own.

Searching for orders (`Cart`, `PaymentPlan` and `PurchaseOrder`) can be done with `OrderContext` class. This class provides many methods to find orders, and not limited to `OrderSearchOptions` and `OrderSearchParameters`. For example, to find for `Cart`, we can use these methods:

```
public Cart[] FindCarts(OrderSearchParameters parameters, OrderSearchOptions opt\
ions)
public Cart[] FindCarts(OrderSearchParameters parameters, OrderSearchOptions opt\
ions, out int totalRecords)
```

Let's write a simple example of searching for carts - we want to delete the carts which are abandoned, customers added something to the carts but then left and forgot about it. There is already a scheduled job for this, but we'll try to mimick it to see what happens under the hood:

```
int startIndex = 0;
int totalRecords = 0;
int daysOld = 30;
OrderSearchParameters parameters = new OrderSearchParameters();
parameters.SqlMetaWhereClause = String.Format("Meta.Modified <  GETDATE() - {0}"\
, daysOld.ToString());
OrderSearchOptions options = new OrderSearchOptions();
options.StartingRecord = startIndex;
OrderContext.Current.FindCarts(parameters, options, out totalRecords);
while (totalRecords > 0)
{
    Cart[] carts = OrderContext.Current.FindCarts(parameters, options, out total\
Records);
    //Delete the carts
}
```

You might have seen this pattern more than once - and it's something you should consider whenever you work with Search API:s. The reason is unless you know exactly the records returned, they can be quite many and cause some performance issues: out of memory or `SqlTimeoutException`. It's better to load and process entries and orders in small chunks.

It looks similiar, right? The only difference with the sample code we had with searching for entries is we use different class "ShoppingCart". You might wonder where the "Meta.Modified" field comes from. `Cart`, `PaymentPlan` and `PurchaseOrder` are metaclasses extend `OrderGroup` - if you still remember the concepts from Order system chapter. When you want to filter on a field specific to one of these class, you have to filter with "Meta.<Fieldname>" on `SqlMetaWhereClass` statement, otherwise you filter on "[OrderGroup].<FieldName>" on `SqlWhereClause`. Let's look at another example where we search for wishlists, where we do filter on the `[OrderGroup].Name`:

```
int startIndex = 0;
int totalRecords = 0;
int totalCount = 0;
OrderSearchParameters parameters = new OrderSearchParameters();
parameters.SqlWhereClause = "[OrderGroup].Name = 'WishList'";
OrderSearchOptions options = new OrderSearchOptions();
options.Classes.Add("ShoppingCart");
options.StartingRecord = startIndex;
OrderContext.Current.FindCarts(parameters, options, out totalRecords);
while (totalRecords > totalCount)
{
    Cart[] carts = OrderContext.Current.FindCarts(parameters, options, out total\
Records);
    totalCount += 20;
    //Do something with the wish lists
}
```

`SqlWhereClause` is just a normal `WHERE` clause, where you can combine conditions, such as `AND` and `OR` to form complex query. In above example, we only filter by `Name`, but with can combine with other conditions, such as `MarketId`. We also have methods to find `PaymentPlans`, which are almost identical to above methods:

```
public PaymentPlan[] FindPaymentPlans(OrderSearchParameters parameters, OrderSea\
rchOptions options)
public PaymentPlan[] FindPaymentPlans(OrderSearchParameters parameters, OrderSea\
rchOptions options, out int totalRecords)
```

For `PaymentPlan`, you can have some searches for the cylce, for example, this code will search for `PaymentPlans` with the cycle of 2 weeks:

```
int startIndex = 0;
int totalRecords = 0;
int totalCount = 0;
OrderSearchParameters parameters = new OrderSearchParameters();
parameters.SqlMetaWhereClause = "[META].PlanCycleMode = 2 AND [META].PlanCycleLe\
ngth = 2";
OrderSearchOptions options = new OrderSearchOptions();
options.Classes.Add("PaymentPlan");
options.StartingRecord = startIndex;
OrderContext.Current.FindPaymentPlans(parameters, options, out totalRecords);
while (totalRecords > totalCount)
{
```

```
    PaymentPlan[] paymentPlans = OrderContext.Current.FindPaymentPlans(parameter\
s, options, out totalRecords);
    totalCount += 20;
    //Do something with the payment plans
}
```

And finally, we have some methods to find the `PurchaseOrders`:

```
public PurchaseOrder[] FindPurchaseOrders(OrderSearchParameters parameters, Orde\
rSearchOptions options, out int totalRecords)
public PurchaseOrder[] FindPurchaseOrdersByStatus(params OrderStatus[] statuses)
```

Enough with the simple cases - let's try a more complicated case. How's about finding purchase orders with was made with lineitems which has a specific value of a custom metafield?

```
OrderSearchParameters parameters = new OrderSearchParameters();
parameters.SqlMetaWhereClause = "META.Color = 'Green'";
parameters.SqlWhereClause = "OrderGroupId IN (Select ObjectId FROM OrderGroup_Pu\
rchaseOrder)";
OrderSearchOptions options = new OrderSearchOptions();
options.Classes.Add("LineItemEx");
PurchaseOrder[] purchaseOrders = OrderContext.Current.FindPurchaseOrders(paramet\
ers, options);
```

This is where the complicated stuffs kick in - to search for a metafield value, you'll need to search in `LineItemEx` by adding it to the `Classes` of `OrderSearchOptions` - then you would be able to query by `SqlMetaWhereClause` as normal with `META.<FieldName>`. The only catch here is that you'll have to make sure you're finding `PurchaseOrder`, not anything else. That's why we need the `SqlWhereClause` to limit the search.

If you want to search for purchase orders which have a specific lineitem by code - unfortunately you can't apply the solution above. The problem is, the code belongs to `LineItem`, which is a system metaclass, while `OrderSearchOptions.Classes` can only work with user metaclasses (`OrderGroup` is a system metaclass, while `PurchaseOrder`, `ShoppingCart` and `PaymentPlan` are not). Here you will need to write a quite complex `SQL` query:

```
parameters.SqlWhereClause = "OrderGroupId IN (SELECT li.OrderGroupId From LineIt\
em li WHERE li.Code = 'Pro-Episerver-Commerce')";
```

Truth is, Order-search API:s are really powerful - you can do almost anything to find the orders you want - however it's less than pleasant to test if your code is correct. I would suggest you to try the simple tool here[43] to verify your query and test the results almost immediately.

[43] http://world.episerver.com/blogs/Quan-Mai/Dates/2014/10/Order-searchmade-easy/

Part 6: Promotions - Everybody loves discounts.

My wife, like almost every other women, becomes crazy when she sees a good discount on an item she likes - but might not really need - and she will buy it and use it only once. Businesses runs that way, marketing works that way and a good eCommerce site always comes with discounts.

It might not be as good as working with discounts, as redeeming them.

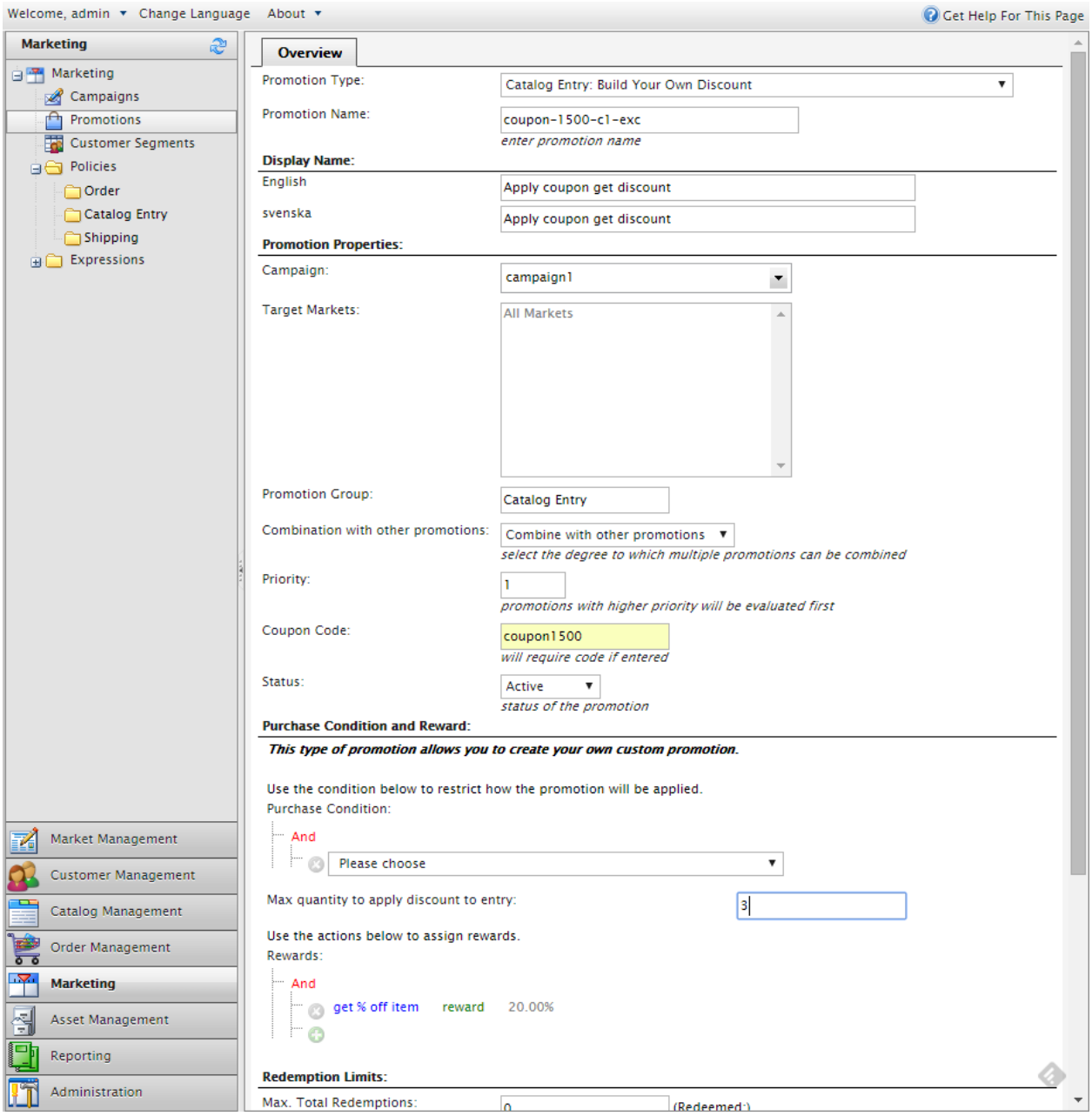
Chapter 12: The old promotion system.

At this time of writing, a new promotion system has been developing from scratch, and it will eventually replace the old one. However, I don't expect every Episerver Commerce site to switch to new system overnight, even when it comes out of beta. We still need to work with the old system and it's better if we understand about it.

For Episerver promotion system, there are two concepts you need to know: Campaigns and Promotions.

Campaign is a simple concept, it's more like of a group of promotions. You can set basic properties such as the valid time, or the markets it applies, and that's it. It's neither significantly sophisticated nor important, but you'll still have to create a campaign before creating any promotions.

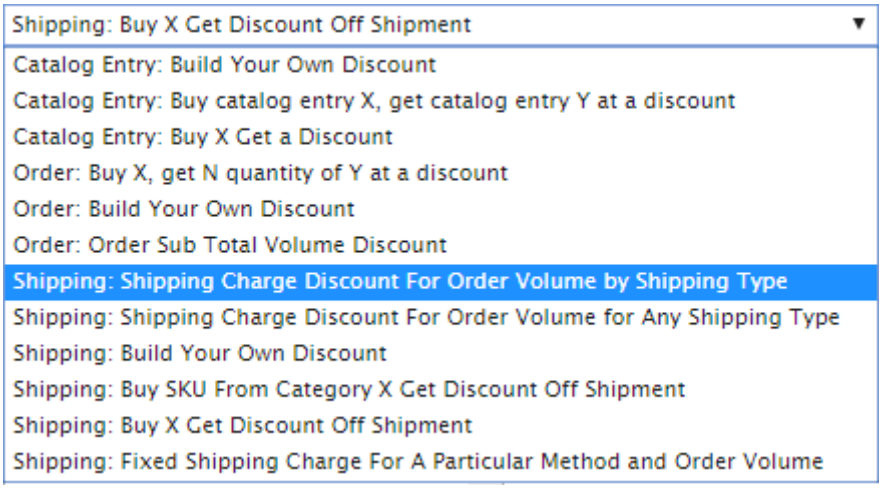
Promotion is a much more complicated concept. It has many settings you'll need to configure to get it work.



A "simple" promotion created in Commerce Manager

Let's walk through the important parts of a promotion:

- Type of promotion. There are 3 types available out of the box: `CatalogEntry`, `Order`, and `Shipping`. Just like the metafields, once you save the promotion, you can't change it type - your only option is to delete and create a new one. This is the most important setting, because it will decide how you can set the conditions and rewards of the promotion. Out of the box, there are 12 predefined promotion types - plenty for almost any requirements:



Shipping promotions seem to be the favourites

- Promotion Group - this is not editable and depends on the type of promotion. Promotion Group takes effect in the next setting, which is ...
- Combination with other promotions: You can choose to combine with other promotions, or exclude with other promotions in same group, or exclude all other promotions. The order of exclusion will depend on ...
- Priority: define the order in which promotions will be applied. Higher priority promotions will be applied first. In case of exclusion, if a promotion excludes another one, then only the higher one will be applied. [44]. For example, if you have a "Global exclusive" promotion with highest priority applied, then no other promotions will be applied. However, if it's not the highest priority and another promotion has been applied, it'll exclude itself.
- Coupon code: a code which customers must input to be able to get the promotion - it's either promotion code, or coupon code, depends on where you are.
- Active: it should explain itself. You can turn a promotion on or off by this flag.
- Purchase Condition and Reward: This is actually an complex combination of settings. Depends on the type of promotion, you can select the condition to be "If the order subtotal is equal or greater than 1000USD", and the reward to be "Get 10% discount off the whole order". There are a very huge amount of possible combinations, so make sure you test the condition before making it promotion public. This is the strength and also the weakness of the old promotion system.
- Redemption limits: This will limit the number of times your customers can redeem the promotion. There are three possible settings:
  - Limit on entire promotion,
  - Limit on one customer (so only logged in customers are entitled to this promotion),
  - limit per order group.

  You can combine all three, and the lowest possible setting will take effect.

[44]The combination, priority and exclusion is a complicated subject, which does not work perfectly until very recent version of Episerver Commerce. This, again, is a reason why you always should stay updated.

As a developer, you usually don't work with old promotion system, at least directly (Which is supposedly a good thing, it is not developer friendly to create promotions anyway). Most of the case, you'll call `CartValidate` or `CartPrepare` workflows, which has an activity named `CalculateDiscounts`, which itself will do bunches of stuffs before calling to `MarketingContext` to apply the promotions. There are only two cases you should care about it:

- When a customer apply a coupon code. It can be as simple as this:

```
    if (!String.IsNullOrEmpty(couponCode))
    {
        MarketingContext.Current.AddCouponToMarketingContext(couponCode);
    }
```

- When you want to get a discounted price for entries, such as when you display products on sale. It's not really an interesting or challenging task to be honest, but for the completeness of the book, we'll discuss it here: We'll have to rely on `PromotionHelper` (A lot of legacy classes from Mediachase named helper) to do the evaluatation of promotion, here is an example:

```
    public virtual PromotionContext Evaluate(PromotionFilter filter,
        PromotionEntriesSet sourceEntriesSet, PromotionEntriesSet targetEntriesS\
et, bool checkEntryLevelLimit)
    {
        var helper = new PromotionHelper();
        helper.PromotionContext.TargetGroup = PromotionGroup.GetPromotionGroup(P\
romotionGroup.PromotionGroupKey.Entry).Key;
        helper.PromotionContext.SourceEntriesSet = sourceEntriesSet;
        helper.PromotionContext.TargetEntriesSet = targetEntriesSet;
        helper.Eval(filter, checkEntryLevelLimit);
        return helper.PromotionContext;
    }
```

`PromotionFilter` is not really a filter of promotions. It's used by `MarketingContext` to see if it should check some additional condition (perhaps it could have been named `ValidationConditions`). For our purpose, we'll have it like this:

```
    var filter = new PromotionFilter
        {
            IgnoreConditions = false,
            IgnorePolicy = false,
            IgnoreSegments = false,
            IncludeCoupons = false
        };
```

- `IgnoreConditions` is false because we'll actually run the condition to verify if there are promotions can be applied.
- `IgnorePolicy` is false so any Policy expressions will be validated (however, this feature is rarely used).
- `IgnoreSegments` is false to validate segment and market. If the current cart does not belong to one of the markets the campaign is targeting, then it should not be entitled to get the promotions in that campaign.
- `IncludeCoupons` is false so `MarketingContext` will skip validating any coupons.

The next thing is `PromotionEntriesSet`. There are two things to fill in: the `sourceEntriesSet` and `targetEntriesSet` - they are collection of `PromotionEntry`. While `sourceEntriesSet` are the entries that might match the condition (required by the promotion), then `targetEntriesSet` are set of entries that might receive the promotion. In this case, we're looking for promotion that might apply for one entry, then both `sourceEntriesSet` and `targetEntriesSet` should be the same. But that's not always true. In other kinds of promotion, it can be different. For example, for Order level promotion, `sourceEntriesSet` will be null (because we'll examinate order's attribute, such as SubTotal or so), while `targetEntriesSet` are the entire lineitem in the order.

```
    private PromotionEntry CreatePromotionEntry(EntryContentBase entry, IPriceVa\
lue price)
    {
        var catalogNodes = string.Empty;
        var catalogs = string.Empty;
        foreach (var node in _catalogSystem.GetCatalogRelationDto(entry.ContentR\
eference.ID))
        {
            var entryCatalogName = _catalogSystem.GetCatalogDto(node.CatalogId).\
Catalog[0].Name;
            catalogs = string.IsNullOrEmpty(catalogs) ? entryCatalogName : catal\
ogs + ";" + entryCatalogName;
            catalogNodes = string.IsNullOrEmpty(catalogNodes) ? node.Code : cata\
logNodes + ";" + node.Code;
        }
        var promotionEntry = new PromotionEntry(catalogs, catalogNodes, entry.Co\
```

```
de, price.UnitPrice.Amount);
        return promotionEntry;
    }
```

I'll leave it here as an exercise - convert this method to use `IContentRepository` entirely.

Now we have all the information we need, let' try to get this discount:

```
        var sourceSet = new PromotionEntriesSet();
        sourceSet.Entries.Add(promotionEntry);
        var promotionContext = Evaluate(filter, sourceSet, sourceSet, false);
```

Evaluate will return a `PromotionContext`, and then we'll have to calculate the discount we get from that:

```
private decimal GetDiscountAmount(PromotionContext promotionContext)
{
    decimal discountAmount = 0;
    foreach (var record in promotionContext.PromotionResult.PromotionRecords)
    {
        var reward = record.PromotionReward;
        if (reward.RewardType != PromotionRewardType.EachAffectedEntry
            && reward.RewardType != PromotionRewardType.AllAffectedEntries)
        {
            continue;
        }
        if (reward.AmountType == PromotionRewardAmountType.Percentage)
        {
            discountAmount = record.AffectedEntriesSet.TotalCost * reward.Amount\
Off / 100;
        }
        else
        {
            discountAmount += reward.AmountOff;
        }
    }
    return discountAmount;
}
```

The `PromotionResult` we get from `PromotionContext` might contain multiple records - each record for one promotion which was successfully applied. We only care about the reward of type `PromotionRewardType.EachAffectedEntry` or `PromotionRewardType.AllAffectedEntries` for the discounted price of the entry. Sum all the reward and we'll have the `discountAmount` of the entry with applied promotions.

The code above is to display discounted prices for the product listing page. We don't have to do extra thing to apply promotions, because it's included in certain workflows such as `CartValidate` - so when a customer adds items into cart, we'll call the certain workflows and the promotions will be applied there.

The old promotion system is one of the reason you'll have to stick with workflows - even if you don't want other stuffs such as inventories checking and adjusting. From what I heard not many developers are fans of Workflows, but it's the way it is. Technically, you can extract the code from certain activities, such as `CalculateDiscountsActivity` and put them in a helper class which can be called directly from your code. However, it's a tedious work and you would want to avoid it anyway.

Coupon

The old promotion system also supports coupon. As we see above, each promotion can have only one coupon, but there is nothing prevents you from having combination of multiple promotions where multiple coupons are used (Except too much discount can be bad for your business). When a customer add a coupon code, you will have to add it to the `MarketingContext` before the promotion system is called:

```
MarketingContext.Current.AddCouponToMarketingContext(couponCode);
```

The coupon is now stored in `MarketingProfileContext`. When `CalculateDiscounts` activity is called, `MarketingContext` will try to get the coupon from `MarketingProfileContext` and apply the promotion. Simple, right, but it's not the entire story. `MarketingProfileContext` is stored in local thread storage if there is no `HttpContext` available, or in `HttpContext.Current.Items` if there is. So for most of the cases, it will be stored in `HttpContext.Current.Items`. The problem is `HttpContext.Current.Items` is per-request storage. If your customer wants to add a new item into the cart, so if he/she simply refreshs the cart page, or changes the address (which in most of the case will send new request to server) – then the storage – and therefore the coupon, is lost. The customer will have to input it again, or, you will have to store it somewhere.

There are two obvious options: Sessions and Cookies. Sessions are more convenient to use, as, in the end, it's only a `NameValueCollection` where you can simply add and get back anything you'd like. However, `Sessions` can't work in a load balancing environment. If you want to use load balancer, then Cookies are the only (simple) choice. Here's an example of what you can do in your Cart page:

```
private void ApplyCoupons()
{
    var cookie = _cookieService.Get(CouponKey);
    if (cookie != null)
    {
        var coupons = JsonConvert.DeserializeObject<HashSet<string>>(cookie);
        if (coupons != null && coupons.Any())
        {
            foreach (var coupon in coupons)
            {
                MarketingContext.Current.AddCouponToMarketingContext(coupon);
            }
        }
    }
}
```

`_cookieService` is only simple service for getting the cookie from HttpContext.Current.Request. The reason of using `HashSet<string>` is to allow customers to input multiple coupon codes, but they must be unique. And to avoid problem with any special characters in coupon codes, here we use `JsonConvert.DeserializeObject` to form a string from the coupons. If you are certain that your coupons will not contain some special characters, a `String.Join` might be enough . And then you need to add the coupon when customer adds it to the cart:

```
var cookie = _cookieService.Get(CouponKey);
var coupons = cookie != null ? JsonConvert.DeserializeObject<HashSet<string>>(co\
okie) : new HashSet<string>();
coupons.Add(couponCode);
_cookieService.Set(CouponKey, JsonConvert.SerializeObject(coupons));
```

It's a bit tricky here – normally you would only want to save the valid coupons, so you might need to check first – by adding the coupon to the `MarketingProfileContext` and run the certain workflows to see if it's applied:

```
MarketingContext.Current.AddCouponToMarketingContext(couponCode);
_cartService.RunWorkflow(OrderGroupWorkflowManager.CartValidateWorkflowName);
_cartService.SaveCart();

if (!GetAppliedDiscountsWithCode().Where(d => couponCode.Equals(d.DiscountCode))\
.Any())
{
    return new EmptyResult();
}
```

`GetAppliedDiscountsWithCode` is a method where you loop over `OrderForms`, `Shipments`, `LineItems` to see if their `Discounts` contain any code. And finally, remove it when the customer no longer wants it in the cart (either they actively delete it, or when they placed the order):

```
var cookie = _cookieService.Get(CouponKey);
if (cookie != null)
{
    var coupons = JsonConvert.DeserializeObject<HashSet<string>>(cookie);
    if (coupons != null && coupons.Contains(couponCode))
    {
        coupons.Remove(couponCode);
        _cookieService.Set(CouponKey, JsonConvert.SerializeObject(coupons));
    }
}
```

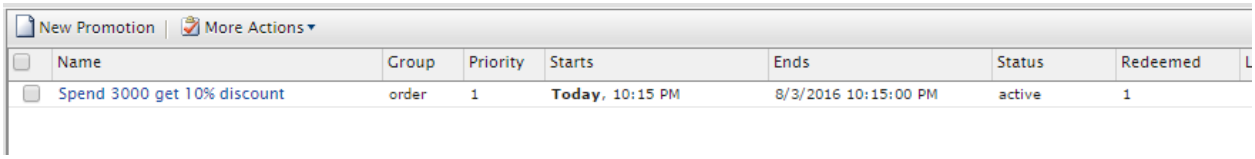
And finally remove it when customer placed the order. Remember, removing the cookie is not enough – you have to make it *expired*.

Redemption

As we mentioned earlier, you can configured the redemption limits for a promotion. A redemption is "counted" when it's actually used. When certain workflows are called, an activity (`RecordPromotionUsageActivity`) will be invoked to record one usage of the promotion. However, this does not necessary count toward the redemption limit, because a 'recorded' usage might still be "Expired" or "Cancelled". It takes a little more work to record the `PromotionUsage` when a customer checkouts an order:

```
Dictionary<string, object> dic = new Dictionary<string, object>();
dic.Add("UsageStatus", PromotionUsageStatus.Used);

OrderGroupWorkflowManager.RunWorkflow(Cart, OrderGroupWorkflowManager.CartCheckO\
utWorkflowName, true, dic);
```

This will run `CartCheckout` workflow with an additional context - to record any `PromotionUsage` for the order with `Used` status.



Only then the redemption is recorded

How promotion works?

The promotion you created, will be turned into a CodeDom Expression. This, along with other information, will be stored in Promotion table in database. If you want to see how the the Expression of your promotion looks like, you can go to Marketing/Expressions/Promotion and open it there. No, I don't like it and it does not really informative or interesting in anyway, but I can cheat two pages of writing, so here it is:

...

(In the end it looks terrible in the book, so sorry to disappoint you. It's "only" 188 lines so I can send it to you if you really want it. Mind you, it's only a "simple" condition.)

When you run certain workflows which contain the `CalculateDiscounts` activity, such as CartValidate, the cart will be pre-processed by `MarketingContext`. `MarketingContext` will do certain checks, such as if the redemption limit has reached or not. If all other conditions passed, `MarketingContext` will call the expression validator (by default implementation, is built around the `RuleEngine` class in `System.Workflow.Activities.Rules`) to validate the conditions to see if the promotion's conditions can be redeemed.

If the cart satisfies all the conditions, then the promotion will be applied - new discounts will be added to `OrderForm.Discounts`, `Shipment.Discounts`, or `LineItem.Discounts`, depends on the type of promotion, or new gift item will be added to cart. The promotion usages - which are used to calculate the redemptions, are also recorded.

Simple, right. Then what is so bad about the old promotion system? Well - plenty of things.

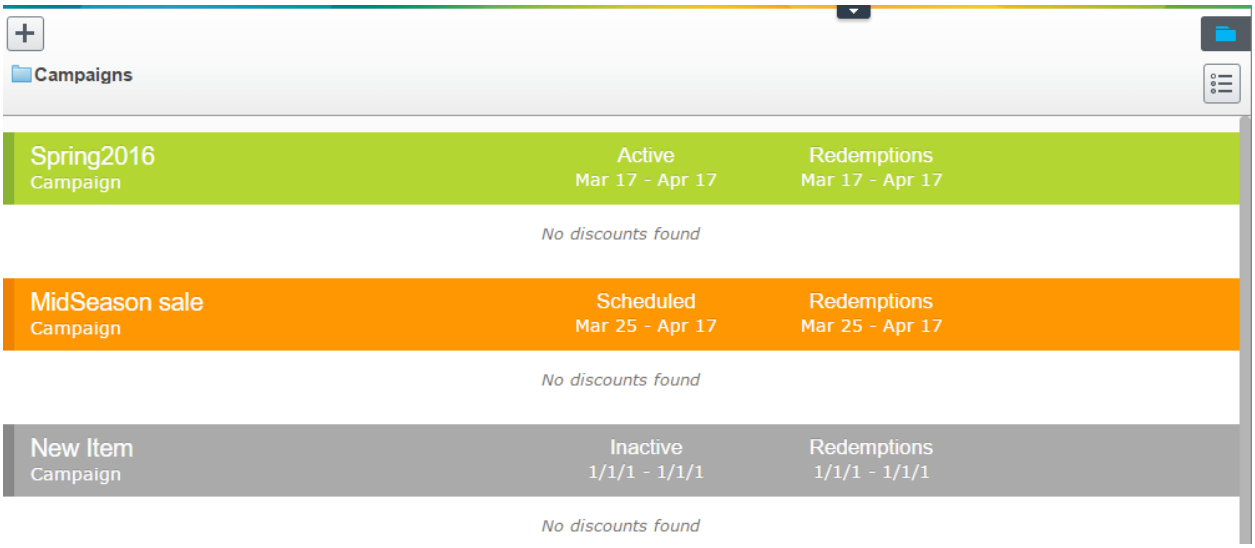
- It is slow. The strength of the rule-based validation is its flexibility, you can define almost any condition/reward you like (of course it's not easy to do so), but its weakness is it's slow

to validate. The validation of condition is based on RuleEngine will struggle to handle high volume of orders, when you have anything more than a few dozens of active promotions (I saw customers with 500 promotions or more, and they are being held back because of the performance).
- It's not developer friendly. The Expression is simply hard to read, write, debug, and impossible to test. You, in most case, can only rely on manual testing to verify the correctness of the conditions (and I believe you'll not like this - automated tests are better in our developers perspective.)
- It handles concurrence badly. You can try by adding a redemption limit to a promotion, and try to place two orders at the same time. It will fail no matter how simple your condition is.
- It lacks of supports for advanced scenarios. For example, there is no way to know if a promotion is almost fulfilled to encourage customer to spend more.

The old promotion system have been there for a long time, and it works to some extends. However, Episerver's partner developers never seem to like it, it has been in the high spot of pain-points lists. The architecture and design old the old promotion system allows very little improvement, both customization and performance. To be fair it might be the best solution, considered the human resources and deadline at the time it was written, but it's no longer the way a promotion system should be. That's why new one was born.

Chapter 13: The new promotion system

In Episerver Commerce 9.19 released in early July, 2016, the new promotion system is considered stable enough. The BETA tags are removed. The API:s will still be evolved, but mostly for adding new features. Changing or removing API:s will now requires at least a major release.



Marketing UI, the new editing UI for new promotion system

There are two terms being used by the new promotion system: discount is the term in user-facing texts, and promotion is for API:s level. In this chapter, we'll use promotion term.

The new promotion system is a completely rewrite of the old one [45]. And they are supposed to work side-by-side, exclusively. You can choose to run your promotions in new or old mode, but not at the same time [46]. The new promotion system has been developed with two things in mind:

- Performance, it has to be fast.
- Usability, it must be easy to work with, especially from developer perspective.

[45] The new system is written from scratch. Nothing from the old one was transferred, even code or data. If there are anything "re-used", they are concepts (campaign, promotion, redemption limit) and behaviors. It's unlike the "re-structure" of catalog system in Commerce 9: only the data which being restructure, while the API:s largely remain the same, or the new "Order API" where new API:s are built on top of existing data. It's entirely new.

[46] It was considered to support both systems at the same time, that will help to reduce the work of partners when they move to new promotion. However, the idea was scrapped as it can be to complicated to get right. I personally think it's a good decision to have one system at a time.

We'll see if Episerver can deliver their promises.

Before you start, there are a couple of things you need to prepare for your site. Note that if you are using Episerver Commerce 9.19 or greater, all of these actions are not required anymore - they are included by default. As always, you are highly recommended to use latest version of Commerce.

- To turn on the new promotion system, it's the same as the new order system. You can either use `IFeatureSwitch` to turn it on, or update the setting in `ecf.app.config`, as documented here[47] .[48]

From 9.19, new sites created by Episerver Commerce will have the new promotion system enabled by default. If you access the Marketing view in Commerce Manager, this warning will be displayed:

"Episerver Commerce is configured to use new promotion system. Use Marketing UI to manage campaigns and promotions instead."

The existing sites which use old promotion system already will not be affect - as the setting is not changed while upgrading. However,if you want to use the old promotion system for newly created sites (I don't see any reasons that you should, but of course you can), remove this configuration from `Configs\ecf.app.config`:

```
<Features>
<add feature="WorkflowsVNext" state="Enabled" type="Mediachase.Commerce.Core.Fea\
tures.WorkflowsVNext, Mediachase.Commerce" />
</Features>
```

Everything is content (again)

When Episerver designed the new promotion system, one of the first question need to be answered: How to work with data, including persisting and editing. Several options were considered, in the end, the content way won. There are several benefits of doing so: the unified API:s (read, `IContentRepository`) can be used, and the editing UI can be extended quite easily.

In new promotion system, everything is content.

Most of the concepts were carried from the old system, but adapted to the content system. The structure of promotion contents is more or less similiar with other content system system - a root, campaigns and promotions. A new `CampaignRoot` was introduced. It's not a virtual content like the RootContent in Commerce, but is saved to the database eventually. You can always access it by `SalesCampaignFolder.CampaignRoot`.

[47] http://world.episerver.com/documentation/Items/Developers-Guide/EPiServer-Commerce/8/Marketing/workflows-beta/

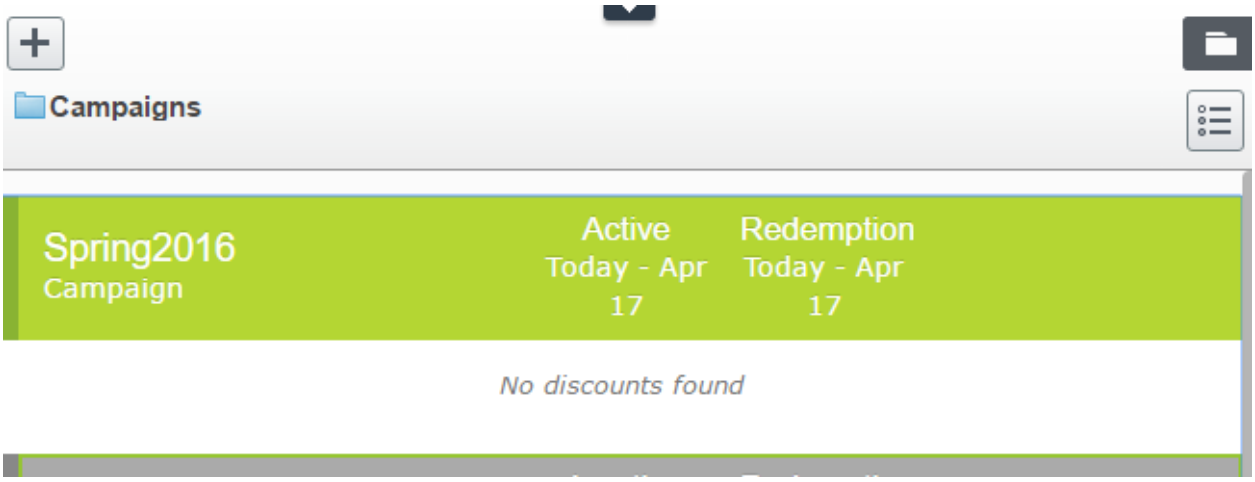
[48] Strictly speaking, the new promotion engine does not rely on the setting to be activated. You can call it directly. The setting is only used for so the `ActivityFlowRunner` engine will know it should call new `ActivityFlow`(s) which use the new promotion engine. Normally, for a quick promotion validation you can call the promotion engine, but in order to check out, you'll still need to run `ActivityFlow`(s) to do additional work (such as checking/requesting inventories).

Working with the new promotion means working with two strongly content types - `SalesCampaign` and `PromotionData`, they both reside in `EPiServer.Commerce.Marketing` namespace and inherit from `BasicContent`. The structure of promotion contents is much simpler than the structure of catalog contents: campaigns are direct children of the Campaign root, and promotions are direct children of campaigns. There is no other level, and there are no direct relations between campaign-campaign or promotion-promotion [49]

Let's try out the content way to see how it works:

```
var contentRepository = ServiceLocator.Current.GetInstance<IContentRepository>();
var campaign = contentRepository.GetDefault<SalesCampaign>(SalesCampaignFolder.C\
ampaignRoot);
campaign.Name = "Spring2016";
campaign.ValidFrom = DateTime.Now;
campaign.ValidUntil = DateTime.Now.AddMonths(1);
campaign.IsActive = true;
contentRepository.Save(campaign, DataAccess.SaveAction.Publish, EPiServer.Securi\
ty.AccessLevel.NoAccess);
```

I hope you can see the benefits of unified content API:s - you can read and understand the code without the explaination. As usual, we get an instance of `IContentRepository`, then create a default `SalesCampaign` instance as children of `SalesCampaignFolder.CampaignRoot` (this is always required, as we learned that `SalesCampaign` must be direct children of the `CampaignRoot`), set some properties, and save it again.



We created a campaign by code. Yay!

`SalesCampaign`, `PromotionData` and their derivatives are handled by the default `ContentProvider` (in other words, they are CMS contents and are saved to CMS database instead of Commerce),

[49] Strictly speaking, there are indirect relations between promotion and promotion, as we'll find out later.

so theoretically, they can have versions. However, the Marketing UI does not support managing versions (You can't add any gadget other than Catalog one). This might change in the future, but for now, you should always use `SaveAction.Publish` with promotion contents.

Let's take a deeper look at `SalesCampaign`:

```
public class SalesCampaign : CampaignData
{
    /// <summary>
    ///      Description.
    /// </summary>
    public virtual string Description { get; set; }

    /// <summary>
    ///     The time when the campaign starts.
    /// </summary>
    public virtual DateTime ValidFrom { get; set; }

    /// <summary>
    ///     The time when the campaign ends.
    /// </summary>
    public virtual DateTime ValidUntil { get; set; }

    /// <summary>
    ///     Status of the campaign: active or note.
    /// </summary>
    public virtual bool IsActive { get; set; }

    /// <summary>
    ///     Tracking code of the campaign.
    /// </summary>
    public virtual string Code { get; set; }

    /// <summary>
    ///     Target visitors group of the campaign.
    public virtual IList<Guid> VisitorGroups { get; set; }

    /// <summary>
    ///     Target markets of the campaign.
    /// </summary>
    public virtual string TargetMarket { get; set; }

    /// <summary>
```

```
    ///     Target revenue of the campaign.
    /// </summary>
    public virtual decimal RevenueGoal { get; set; }
}
```

Unlike the old system, while Campaign is only a place holder for promotions, the `SalesCampaign` has actually information that can be useful for campaign managers:

```
- `Code`: can be used for tracking, such as Google Analytics tracking code.
- `VisistorGroups`: can be used to define groups of users which are entitled to \
this campaign.
- `TargetMarket`: Only orders in this market are entitled to the promotions in t\
his campaign. Empty market means no orders can be redeemed, while set it to `EPi\
Server.Business.Commerce.Constants.AllMarkets` ("EPI_AllMarkets") means all mark\
ets.
- `RevenueGoal`: This is more of report feature than actually being used in the \
system.
```

Of course, you can extend `SalesCampaign` to add the properties you want to. However, in real world case scenarios, I don't think you, as a developer, will work directly with `SalesCampaign` everyday. It should be the work of editors/campaign managers to edit the campaign in the UI. It's the `PromotionData`, the processors, and the promotion engine we need to care about.

The promotion engine:

The promotion engine can be considered as the core of new promotion engine. It's a very simple interface, with only two methods:

```
public interface IPromotionEngine
{
    /// <summary>
    /// Checks the condition for all active promotions, and gives awards for the\
 promotions where the condition has been fulfilled.
    /// </summary>
    IEnumerable<RewardDescription> Run(IOrderGroup orderGroup);

    /// <summary>
    /// Checks the condition for all active promotions, and gives awards for the\
 promotions where the condition has been
    /// fulfilled. Promotions with higher priority (lower Priority value) will b\
e evaluated first and all excluded promotions  /// will not be evaluated.
```

```
    /// </summary>
    IEnumerable<RewardDescription> Run(IOrderGroup orderGroup, PromotionEngineSe\
ttings settings);
}
```

For your convenience, `PromotionEngine` - the default implementation will do those steps:

- Loads all promotions available, then filters inactive one and sort them by priority.
- Filters promotions which require coupon codes, but there was none supplied with the order group.
- For each promotion:
  - Checks for redemption limits of promotion to see if a promotion can be fulfilled.
  - Loads the processor which can process the promotion, `Evaluate` it.
  - Each promotion will return a list of Redemption. Applies the redemptions until the redemption limits are reached.
- Return list of `RewardDescription`. Each `RewardDescription` contains these information:

```
1     public class RewardDescription
2     {
3         /// <summary>
4         /// Gets or sets the reward typed, can be percentage, amount or free.
5         /// </summary>
6         public RewardType RewardType { get; private set; }
7
8         /// <summary>
9         /// Gets the discount amount when the discount type is money.
10        /// </summary>
11        public decimal UnitDiscount { get; private set; }
12
13        /// <summary>
14        /// Gets the discount amount when the discount type is percentage.
15        /// </summary>
16        public decimal Percentage { get; private set; }
17
18        /// <summary>
19        /// Gets the amount of money this reward has given the order.
20        /// </summary>
21        public decimal SavedAmount { get; internal set; }
22
23        /// <summary>
24        /// Gets the reward typed, can be percentage, amount or free.
```

```
25          /// </summary>
26          public RewardType RewardType { get; private set; }
27
28          /// <summary>
29          /// Gets the status.
30          /// </summary>
31          public FulfillmentStatus Status { get; private set; }
32
33          /// <summary>
34          /// Gets the redemptions.
35          /// </summary>
36          public IEnumerable<RedemptionDescription> Redemptions { get; private set\
37 ; }
38
39          /// <summary>
40          /// Gets the promotion.
41          /// </summary>
42          public PromotionData Promotion { get; private set; }
43
44          /// <summary>
45          /// Gets or sets a description of the promotion result, only for debuggi\
46 ng and logging.
47          /// </summary>
48          public string Description { get; set; }
49      }
```

Of course, you can implement the `IPromotionEngine` interface yourself, if you want to. But in most of the case, you can just use the default one. As developers, the things we will mostly deal with is the Promotion types and the processors itself.

Promotion type

New promotion system means strongly typed type all the way. That does mean we define our promotions by POCO classes, inherits from `PromotionData`:

```
public abstract class PromotionData : BasicContent
{
    /// <summary>
    /// Gets or sets the description of the promotion.
    /// </summary>
    public virtual string Description { get; set; }

    /// <summary>
    /// Gets or sets the promotion schedule.
    /// </summary>
    public virtual PromotionSchedule Schedule { get; set; }

    /// <summary>
    /// Gets or sets the status of the promotion.
    /// </summary>
    public virtual bool IsActive { get; set; }

    /// <summary>
    /// Gets or sets coupon information.
    /// </summary>
    public virtual CouponData Coupon { get; set; }

    /// <summary>
    /// Gets or sets redemption limits.
    /// </summary>
    public virtual RedemptionLimitsData RedemptionLimits { get; set; }

    /// <summary>
    /// Gets or sets the priority of promotion. The lower value, the higher prio\
rity.
    /// </summary>
    public virtual int Priority { get; set; }

    /// <summary>
    /// Gets or sets the list of excluded items.
    /// Excluded items won't be combined with this promotion.
    /// </summary>
    public virtual IList<ContentReference> ExcludedItems { get; set; }

    /// <summary>
    /// Gets whether the promotion is for an order or individual line items.
    /// </summary>
```

```
    public abstract DiscountType DiscountType { get; }
}
```

Note that we don't have to inherit from `PromotionData` directly, we can inherit from its subclasses - `EntryPromotion`, `OrderPromotion` and `ShippingPromotion`, which basically `PromotionData` with `DiscountType` set (`DiscountType.LineItem`, `DiscountType.Order` and `DiscountType.Shipping`, respectively). There are three thing interesting about `PromotionData` we might care about:

- `PromotionSchedule`: allow you to set if you want to stick with the campaign schedule, or to have its own schedule.`PromotionSchedule` is a rather simple `BlockData`:

```
    [ContentType(GUID = "C350562B-F36F-4A6E-BDEA-C30D4009280E", AvailableInEditM\
ode = false)]
    [AdministrationSettings(CodeOnly = true, GroupName = "marketing")]
    public class PromotionSchedule : BlockData
    {
        /// <summary>
        /// Gets or sets the track Campaign validation dates.
        /// </summary>
        public virtual bool UseCampaignDate { get; set; }

        /// <summary>
        /// Gets or sets the time when the promotion starts.
        /// </summary>
        public virtual DateTime ValidFrom { get; set; }

        /// <summary>
        /// Gets or sets the time when the promotion ends.
        /// </summary>
        public virtual DateTime ValidUntil { get; set; }

        /// <summary>
        /// Gets or sets the reference to the campaign.
        /// </summary>
        public virtual ContentReference CampaignLink { get; set; }
    }
```

- `CouponData`: By default, `Coupon` only contains one simple `string` field, where you can set the coupon code for the entire promotion - more or less the same with the old system. We'll discuss later about extending this to support more advanced scenarios.

- `RedemptionLimitsData`: It can be used to limit how many times a promotion can be redeemed. You can set it to be per-promotion, per-orderform, or per-customer (Requires customers to be logged in). If multiple limits are applied, then the lowest available value will be used. Like `PromotionSchedule`, it's just a `BlockData` with three nullable `int` for the limits.

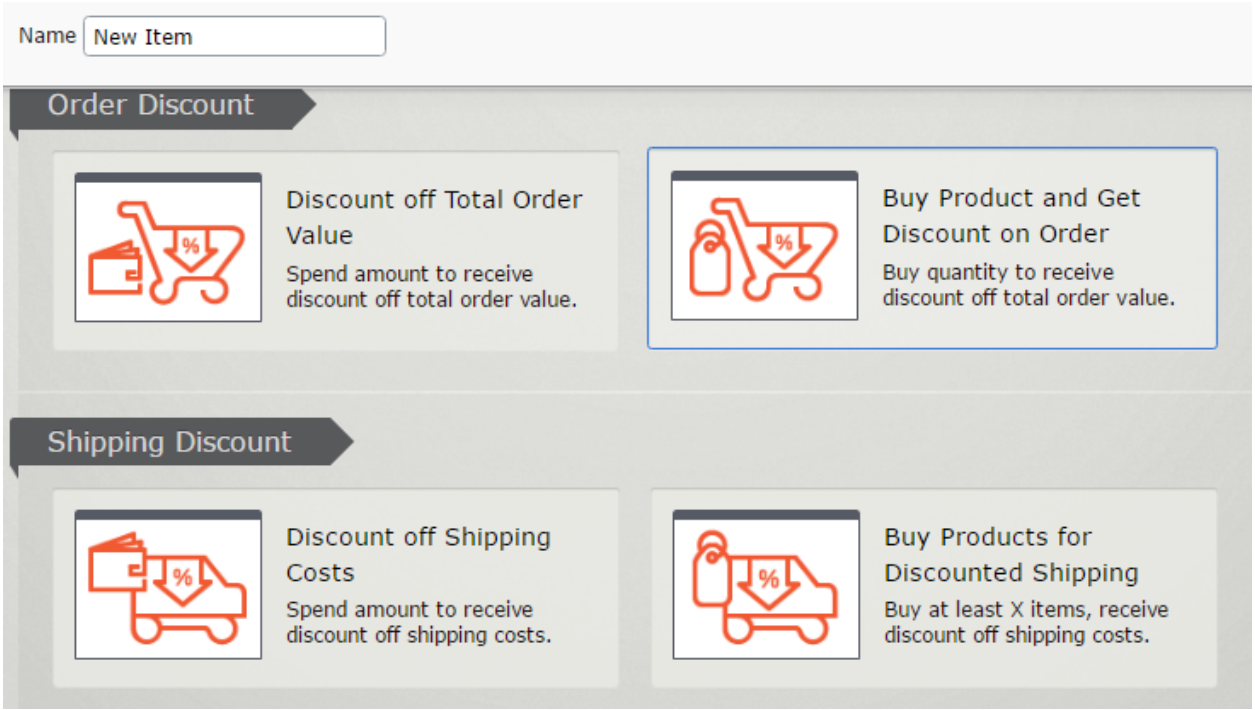
Promotion data and processor

As we learned above, there are three basic types of promotions: `EntryPromotion`, `OrderPromotion`, and `ShippingPromotion`.

Each has different characteristics, so we'll examine them one by one.

Order-level promotion

An order-level promotion is a promotion when the conditions, or the rewards, are based on the criteria of the promotion.

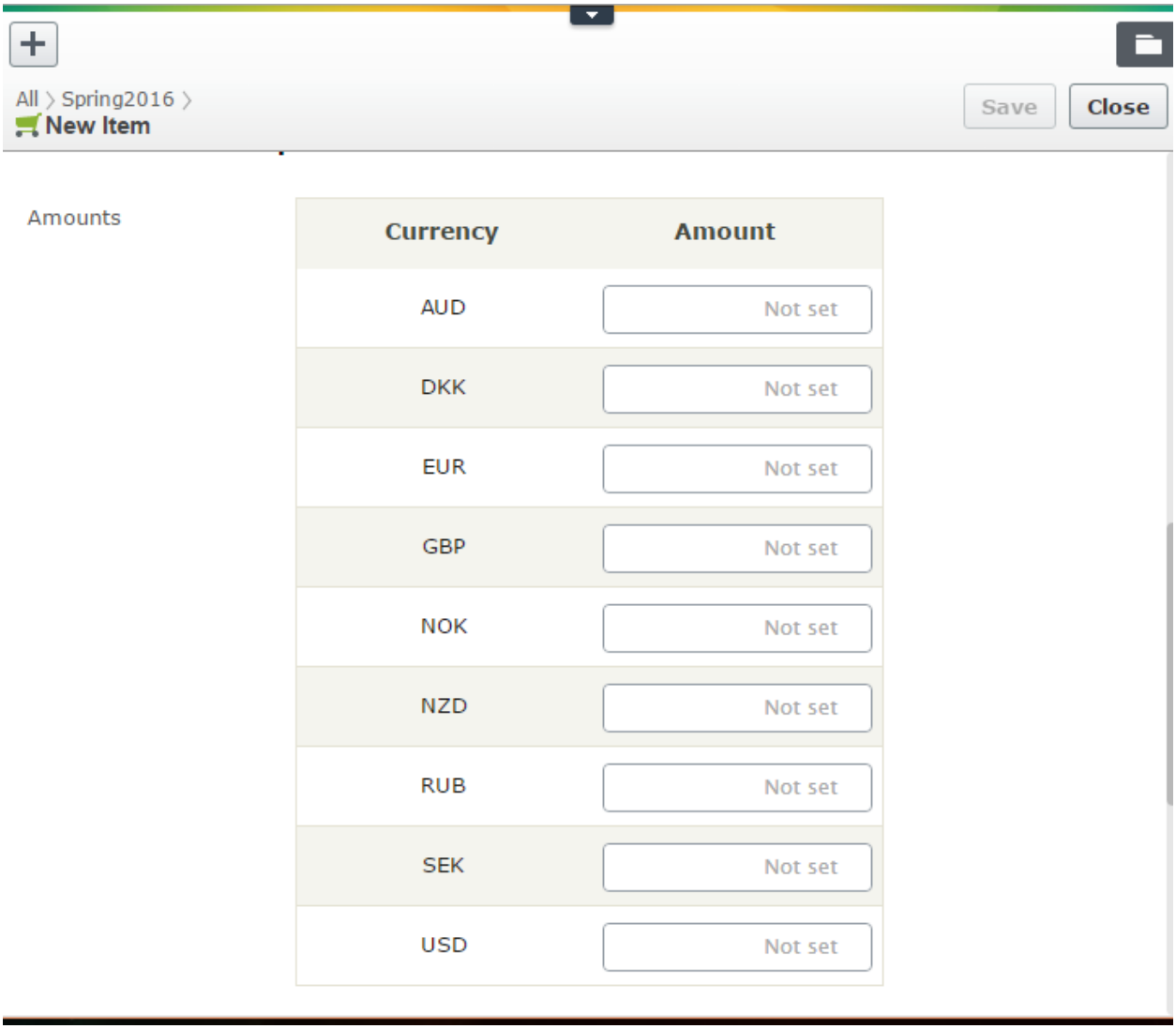


At this time of writing, Episerver Commerce comes with 2 order-level promotion out of the box

If a customer spend equal or more than a specific amount, he or she will get a 10% discount off the cart. We will one to support multiple currencies setting, for example, if the current cart is in USD, we want the threshold to be $100, while if it is SEK, then the threshold is SEK1000, etc…Because this cart affects the whole order (both the *conditions* and *rewards*) then it should be an OrderPromotion. First, let's create a content type for our promotion:

```
[ContentType(GUID = "7660BDC2-6F65-4D22-8FDB-812EE7F4074D")]
public class SpendAmountGetDiscount : OrderPromotion
{
    /// <summary>
    /// The amount of currencies required to get this promotion.
    /// </summary>
    public virtual IList<Money> Amounts { get; set; }

    /// <summary>
    /// The percent of amount off the order.
    /// </summary>
    public virtual decimal Discount { get; set; }
}
```

The content type, as usual, has to be an unique GUID. The interesting here is `Amounts` property - which is an `IList<Money>`. This is one of two specially treated `IList<T>` properties in the Marketing UI - the other one is `IList<ContentReference>`. Build your project and then go to marketing view, you now can see our new promotion type, and creating it will give us something like this:

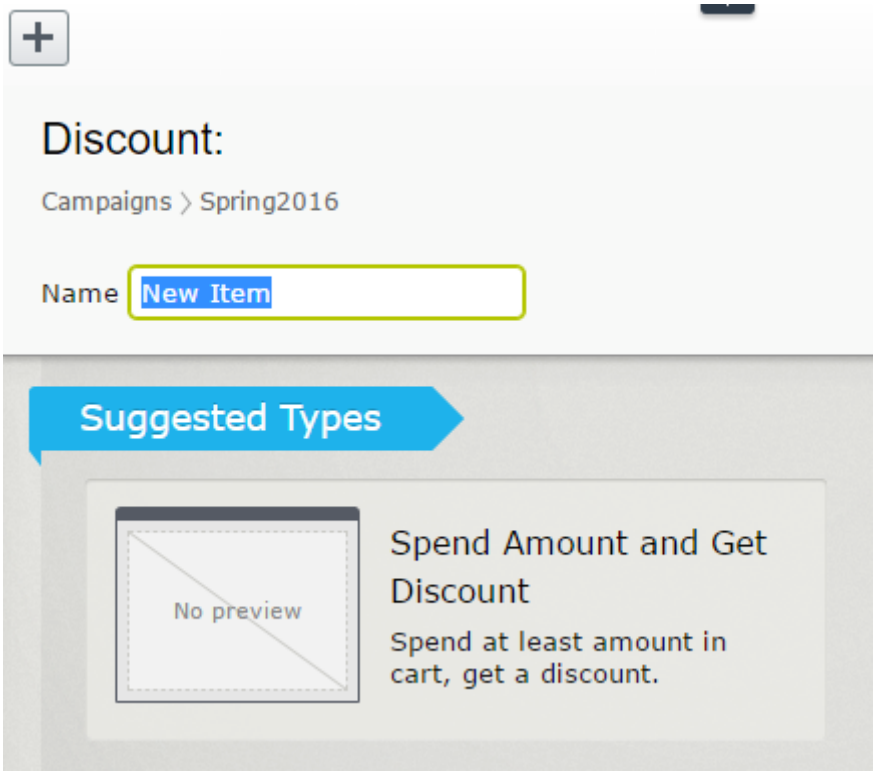


A list of money amount to set to

You can see that `IList<Money>` has been rendered to a very nice "control" (or widget, more precisely). You can edit the value for each currency separatedly. Easy, right? We did almost nothing and there are already results! However, there are two things to note: - The list of currencies available depends on the market you choose for your campaign. By default, "All" option is selected, which mean all currencies from all markets will be available. Selecting a specific market will limit the currencies you can set. - All the texts in new promotion is only the type name, which is not really nice. You can make it better by adding this localization file (for English, you can add files for other languages in same structure) - as I said, it's just another content type:

```
    <?xml version="1.0" encoding="utf-8" standalone="yes"?>
    <languages>
    <language name="English" id="en">
        <contenttypes>
        <spendamountgetdiscount>
            <name>Spend Amount and Get Discount</name>
            <create see="/contenttypes/spendamountgetdiscount/name" />
            <description>Spend at least amount in cart, get a discount.</descrip\
tion>
            <formdescription>Spend at least amount in certain currencies and get\
 a discount.</formdescription>
            <properties>
            <amount>
                <caption>Spend this amount</caption>
            </amount>
            <discount>
                <caption>Get a discount</caption>
                <help>The reward that should be used when the discount is applie\
d.</help>
            </discount>
            </properties>
        </spendamountgetdiscount>
        </contenttypes>
    </language>
    </languages>
```

Set it to be Embedded Resources and build your project, and it's better now:



Localized promotion

But not that's the only one visual improvement you can make to your promotion, you can also add an "icon/thumbnail" to it:

```
[ContentType(GUID = "7660BDC2-6F65-4D22-8FDB-812EE7F4074D")]
[ImageUrl("Images/SpendAmountGetDiscount.png")]
public class SpendAmountGetDiscount : OrderPromotion
```

But something is not ... right. The other builtin promotions are grouped nicely in three groups: Order Promotions, Shipping Promotions and Entry Promotions. Our new custom promotion is standing alone. Can we make it into Order Promotions group and place it on top of the list? Of course.

```
[ContentType(GUID = "7660BDC2-6F65-4D22-8FDB-812EE7F4074D", GroupName = "orderpr\
omotion", Order = 100)]
public class SpendAmountGetDiscount : OrderPromotion
```

Having `GroupName` set to "orderpromotion" will put our promotion into Order Promotion groups - while `Order` will determine how should it be positioned with other promotions - lower `Order` value means higher position in the list.

The only thing remaining is to create a processor, which will be called by the promotion engine to apply the promotion. We'll have to inherit from `PromotionProcessorBase<T>`, where T is

a class of type `PromotionData`. In this case, it's our `SpendAmountGetDiscount` promotion. At runtime, promotion engine will do a little magic (read, refection), to get the processor which can process a specific promotion - so it will look up for a class inherits from `PromotionProcessorBase<TSpecificPromotion>` to process a `TSpecificPromotion` promotion.

```
public class SpendAmountGetDiscountProcessor : OrderPromotionProcessorBase<Spend\
AmountGetDiscount>
{
    /// <summary>
    /// Evaluates a promotion against an order form.
    /// </summary>
    public override RewardDescription Evaluate(SpendAmountGetDiscount promotionD\
ata, PromotionProcessorContext context)
    {
        throw new NotImplementedException();
    }

    protected override bool CanBeFulfilled(SpendAmountGetDiscount promotionData,\
 PromotionProcessorContext context)
    {
        throw new NotImplementedException();
    }

    protected override PromotionItems GetPromotionItems(SpendAmountGetDiscount p\
romotionData)
    {
        throw new NotImplementedException();
    }
}
```

The processors can be reused. That means, if you have a class `SpendAmountGetDiscountEx` which inherits from `SpendAmountGetDiscount` and adds some extra properties for data (but they are not used in calculation), you don't have to create a processor for it - you can just reuse `SpendAmountGetDiscountProcessor` - `PromotionEngine` can handle it just fine.

There is a method we have to implement - `Evaluate` will take a `PromotionData` which contains the conditions and rewards of the promotion, and the a context (where we can get information about other data such as the `OrderGroup` itself). This `PromotionProcessorContext` contains almost any information from the "context" we need to evaluate the promotion:

```
public class PromotionProcessorContext
{
    /// <summary>
    /// Gets the <see cref="IOrderGroup"/> that is being evaluated.
    /// </summary>
    public IOrderGroup OrderGroup { get; }

    /// <summary>
    /// Gets the current order form that the promotion engine is processing at t\
he time.
    /// </summary>
    public IOrderForm OrderForm { get; }

    /// <summary>
    /// Gets the current order form sub total that the promotion engine is proce\
ssing at the time. This is calculated before every promotion has been evaluated.
    /// </summary>
    public decimal CurrentOrderFormSubTotal { get; }

    /// <summary>
    /// Gets the price matrix for entries.
    /// </summary>
    public OrderFormPriceMatrix EntryPrices { get; }

    /// <summary>
    /// Gets the price matrix for shipments.
    /// </summary>
    public ShipmentPriceMatrix ShipmentPrices { get; }
}
```

For our processor, we will create a helper method to calcuate the order subtotal - where we also need to care about the discounts applied by previous promotions. Lucky for us, we can use an existing extension method `IOrderForm.GetSubTotal(Currency)` to calculate the subtotal of the order form.

And then `Evaluate` is simple enough. `RewardDescription` provides a static method to create an instance with percentage discount:

```
protected override RewardDescription Evaluate(SpendAmountGetDiscount promotionDa\
ta, PromotionProcessorContext context)
{
    var orderForm = context.OrderForm;
    var threshold = promotionData.Amounts.FirstOrDefault(c => c.Currency == cont\
ext.OrderGroup.Currency);
    var subTotal = orderForm.GetSubTotal(context.OrderGroup.Currency).Amount;
    if (threshold.Amount <= subTotal)
    {
        var redemption = new List<RedemptionDescription> { CreateRedemptionDescr\
iption(orderForm) };
        return RewardDescription.CreatePercentageReward(FulfillmentStatus.Fulfil\
led, redemption, promotionData, promotionData.Discount, "The promotion has been \
fulfilled");
    }
    else
    {
        return RewardDescription.CreateNotFulfilledDescription(promotionData, Fu\
lfillmentStatus.NotFulfilled);
    }
}
```

The next thing to implement is `CanBeFulfilled`. This method serves as a quick check to see if we should further process the promotion or not. As `Evaluate` can be "slow" to execute, and in such cases we should avoid running it as much as possible. In this example, our check in `CanBeFulfilled` is simple - because we defined a list of `Money` to apply this promotion, so we will check for the currency of the order being evaluated:

```
protected override bool CanBeFulfilled(SpendAmountGetDiscount promotionData, Pro\
motionProcessorContext context)
{
    return promotionData.Amounts.Any(c => c.Currency == context.OrderGroup.Curre\
ncy);
}
```

The final thing is `GetPromotionItems`. It should return an `PromotionItems` instance, which indicates the items which should satisfy the conditions, and the items which should be in the rewards. `PromotionsItems` has a constructor which takes 3 parameters: a `PromotionData` instance, a `CatalogItemSelection` for conditions and a `CatalogItemSelection` as reward.

As we don't really care about the items in the cart, and the reward is for the entire order, so here's what it should look like:

```
protected override PromotionItems GetPromotionItems(SpendAmountGetShippingDiscou\
nt promotionData)
{
    return
        new PromotionItems(
            promotionData,
            new CatalogItemSelection(null, CatalogItemSelectionType.All, true),
            new CatalogItemSelection(null, CatalogItemSelectionType.All, true));
}
```

We will take more about `CatalogItemSelection` in upcoming section.

If the `SubTotal` of the `orderForm` is greater than or equal the required amount, then we give the discount. To let the promotion engine knows about that, we return a list of `RedemptionDescription`, which each contains the information of what has been given discount for (the entire order, specific line items, or the shipping cost). Here we are using a protected method (`CreateRedemptionDescription`) from the parent class `OrderPromotionProcessorBase`, to create a `RedemptionDescription` contains information about our `OrderForm`. In case the amount does not satisfy the requirement, we simply return a `NotFulfilled RewardDescription`.

Let's try to see if it works - Add a promotion with the required spent amount of USD1500 and the discount amount of 10%, then try to add something to the cart, of course, with the "WorkflowsVNext" flag enabled.

	Product Name	List Price	Your Price	Quantity	Total
	Tops-Tunics-CowlNeck-Black-Small	~~$1,000.00~~	$1,000.00	1 Default Warehouse	**$900.00**
	Tops-Tunics-LongSleeve-Black-Small	~~$1,000.00~~	$1,000.00	1 Default Warehouse	**$900.00**

Ta da! It works

I got a tweet from a reader saying that this picture is not correct - "Your price" should be 1000$. That is a reasonable question, however this is indeed correct. Because this is an order level discount, the price for each item is not changed, however the discount will be "distributed" for each item, which is 100$ in this case.

One interesting thing in above code is the `RedemptionDescription` - the processor is supposed to return that "description" to the promotion engine about the affected entities, the saved amount etc. In

this case, we have a promotion which affect the entire order, so we return an `AffectedOrder` object which points to the orderForm. In later examples, we will talk about other types of promotion which takes lineitem or shipment, and they will have different affected entities.

We kept it simple for demonstration, but let's consider a more advanced scenario: If you set the target to be USD1500, and the customer already has spent USD1200 in the cart, it's really nice to display some message like "Spend $300 more and get a 10% discount". In the end, we need to boost our sales while still make our customers happy, a little push like this can be more effective than you would think (I can speak for myself in this matter!)

To do that, let's add a property to `SpendAmountGetDiscount` promotion (and of course you might want to add a localized string for this new property):

```
public virtual decimal AlmostFulfilledThreshold { get; set; }
```

This property allows editors to set the threshold that customers start seeing the notification to "spend more". Now we can update `Evaluate` method to handle this number:

```
if (threshold.Amount * promotionData.AlmostFulfilledThreshold > subTotal)
{
    return RewardDescription.CreateNotFulfilledDescription(promotionData, Fulfil\
lmentStatus.NotFulfilled);
}
else
{
    var redemption = new List<RedemptionDescription> { CreateRedemptionDescripti\
on(orderForm) };
    if (threshold.Amount < subTotal )
    {
        return RewardDescription.CreatePercentageReward(FulfillmentStatus.Fulfil\
led, redemption, promotionData, promotionData.Discount, "The promotion has been \
fulfilled");
    }
    else
    {
        return RewardDescription.CreatePercentageReward(FulfillmentStatus.Partia\
llyFulfilled, redemption, promotionData, promotionData.Discount, "The promotion \
is almost fulfilled");
    }
}
```

You now might wonder about which statuses you can use for `RewardRedemption`, here's the full list of `FulfillmentStatus`:

```
[Flags]
public enum FulfillmentStatus
{
    NotFulfilled = 1 << 0,

    PartiallyFulfilled = 1 << 1,

    Fulfilled = 1 << 2,

    CouponCodeRequired = 1 << 8,
    Excluded = 1 << 9,
    VisitorGroupRequired = 1 << 10,
    RedemptionLimitReached = 1 << 11,

    InvalidCoupon = NotFulfilled | CouponCodeRequired,
    InvalidCombination = NotFulfilled | Excluded,
    MissingVisitorGroup = NotFulfilled | VisitorGroupRequired,
    NoRedemptionRemaining = NotFulfilled | RedemptionLimitReached,
}
```

Normally, you will only return one of first three statuses: `NotFulfilled`, `PartiallyFulfilled` or `Fulfilled` from your processor. Other statuses are expected to be returned by the implementation of `IPromotionEngine`. (Of course, nothing prevent you from implementing it your own and use those statuses as well).

Now we created a promotion which can return a `PartiallyFulfilled` `RewardDescription` in case the customer is close to their reward. Of course, as the seller, you want to let them know about that - to encourage them to spend some more money to get rewarded. You should be able to check it when a customer adds a new item to the cart - a quick evaluation for promotions, but nothing is applied. That's when the other overload of `IPromotionEngine.Run` comes in to play - you can pass a `PromotionEngineSettings` which specifies the statuses you are interested in, and if you want to apply the reward or not. Here's the code in the "Mini Cart" can look like:

```
var promotionEngine = ServiceLocation.ServiceLocator.Current.GetInstance<IPromot\
ionEngine>();
var rewards = promotionEngine.Run(cart, new PromotionEngineSettings() { Requeste\
dStatuses = RequestFulfillmentStatus.PartiallyFulfilled, ApplyReward = false });
var almostFulfilledPromotions = rewards.Where(r => r.Status == FulfillmentStatus\
.PartiallyFulfilled);
```

You now have a list of almost fulfilled promotions - and it's up to you to display some encouragement to the customers. They will be happy spending more money and you will be happy selling more items. It's a win-win situation!

In this section we talked about `IPromotionEngine.Run`, so you can get used to the API:s. However, in real code, you can use some of the extensions method, most notably `IOrderGroup.ApplyDiscounts()` and `IOrderGroup.ApplyDiscounts(PromotionEngineSettings)`. In above example, it can be rewritten to `cart.ApplyDiscounts(new PromotionEngineSettings() ...)`. There is another overload which take an `IPromotionEngine` as parameter, so you can mock it for unit-testing.

Unit testing

As I mentioned earlier, one of the reasons for the existing of new promotion system, is to make it easier to write tests. With the old system, the only possible way for automation test is to write integration tests: Set up a running environment mimicking a working site, then try to create a campaign, a promotion (which is, tedious) , then create a cart and add line items to it, and call the workflows to get the promotion applied, and verify the result. Trust me, you won't like it.

The approach in new promotion system allows you to easily write unit test for your processor. It *is* much more easier to mock your class up and verify your logic only. While you should write integration tests nonetheless, you should prefer to write unit tests - they are easier to understand, and fast to run. You can use any unit testing framework of choice, but in this example we will use xUnit, one of the most popular options (which is also used by Episerver Commerce team internally).

From the code of `Evaluate`, it looks like we should verify three cases: When the promotion is not fulfilled, when it is almost fulfilled and when it's fulfilled. We will need to construct an instance of `SpendAmountGetDiscountProcessor` and run `Evaluate` on an setup instance of `PromotionProcessorContext` which contains an `IOrderForm` which might or might not satisfies the condition. Then we assert the return value.

There was a small problem with our code - `Evaluate` use `IOrderForm.GetSubTotal(Currency)` - which does not really "mockable" (Extension methods are not really mockable in general, right?). We need to change it to the other overload `IOrderForm.GetSubTotal(Currency, IOrderFormCalculator)`, where we can mock `IOrderFormCalculator`, so our new processor will look like this:

```
public class SpendAmountGetDiscountProcessor : OrderPromotionProcessorBase<Spend\
AmountGetDiscount>
{
    private IOrderFormCalculator _orderFormCalculator;

    public SpendAmountGetDiscountProcessor(IOrderFormCalculator orderFormCalcula\
tor)
    {
        _orderFormCalculator = orderFormCalculator;
    }
```

```
11
12      protected override RewardDescription Evaluate(SpendAmountGetDiscount promoti\
13  onData, PromotionProcessorContext context)
14      {
15          var orderForm = context.OrderForm;
16          var threshold = promotionData.Amounts.FirstOrDefault(c => c.Currency == \
17  context.OrderGroup.Currency);
18          var subTotal = orderForm.GetSubTotal(context.OrderGroup.Currency, _order\
19  FormCalculator).Amount;
20          ...
21
22      }
23  }
```

This is a perfect example of dependency injection - we inject the "dependency" into the constructor. Now let's create some tests. Creating an instance of `SpendAmountGetDiscount` is easy enough - it's just another POCO object. We need to create an instance of `IOrderGroup` to mock the `Currency`, and then an instance of `IOrderFormCalculator` to mock for `GetSubTotal`. We test for `SpendAmountGetDiscountProcessor` and the result it would return when the subtotal of the `OrderForm` fulfills or not fulfills the requirement, so we mock the value returned by `GetSubTotal`.

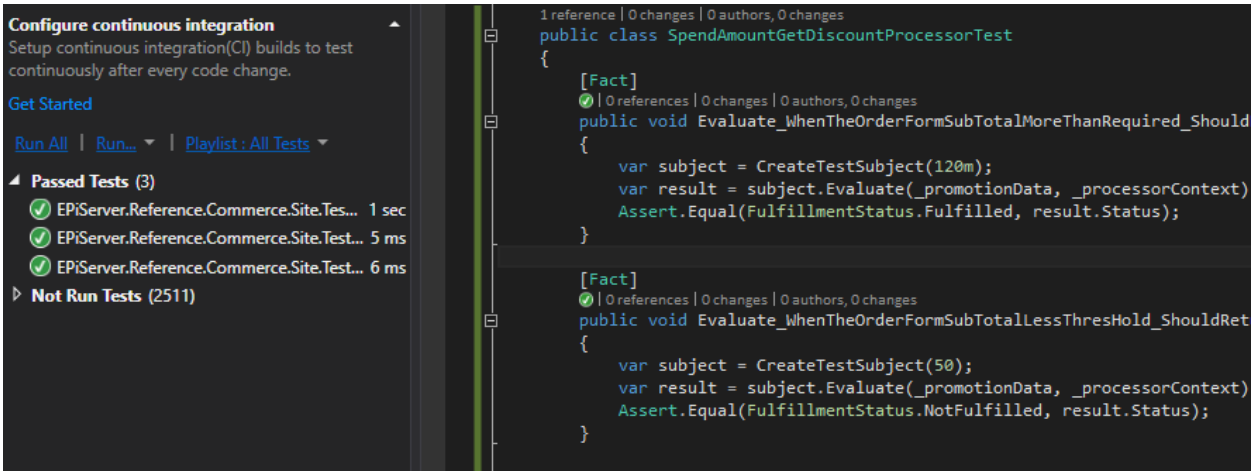
We are using Moq here, but of course you can use any mock framework of your choice - the syntax can be slightly different, but you get the idea:

```
1   public class SpendAmountGetDiscountProcessorTest
2   {
3       SpendAmountGetDiscount _promotionData;
4       PromotionProcessorContext _processorContext;
5
6       public SpendAmountGetDiscountProcessorTest()
7       {
8           _promotionData = new SpendAmountGetDiscount()
9           {
10              Amounts = new List<Money>() { new Money(100m, Currency.USD) },
11              Discount = 10m,
12              AlmostFulfilledThreshold = 0.9m
13          };
14          var orderGroupMock = new Mock<IOrderGroup>();
15          orderGroupMock.Setup(o => o.Currency).Returns(Currency.USD);
16          _processorContext = new PromotionProcessorContext(orderGroupMock.Object,\
17   new FakeOrderForm(), null, null, null);
18      }
19
```

```
    public SpendAmountGetDiscountProcessor CreateTestSubject(decimal subTotal)
    {
        var orderFormCalculatorMock = new Mock<IOrderFormCalculator>();
        orderFormCalculatorMock.Setup(c => c.GetSubTotal(It.IsAny<IOrderForm>(),\
 It.IsAny<Currency>())).Returns(new Money(subTotal, Currency.USD));

        return new SpendAmountGetDiscountProcessor(orderFormCalculatorMock.Objec\
t);
    }
}
```

And there are the tests we wanted to write

```
[Fact]
public void Evaluate_WhenTheOrderFormSubTotalMoreThanRequired_ShouldReturnFulfil\
led()
{
    var subject = CreateTestSubject(120m);
    var result = subject.Evaluate(_promotionData, _processorContext);
    Assert.Equal(FulfillmentStatus.Fulfilled, result.Status);
}

[Fact]
public void Evaluate_WhenTheOrderFormSubTotalLessThresHold_ShouldReturnNotFulfil\
led()
{
    var subject = CreateTestSubject(50);
    var result = subject.Evaluate(_promotionData, _processorContext);
    Assert.Equal(FulfillmentStatus.NotFulfilled, result.Status);
}

[Fact]
public void Evaluate_WhenTheOrderFormSubTotalLessRequiredButMoreThanThreshold_Sh\
ouldReturnPartiallyFulfilled()
{
    var subject = CreateTestSubject(95);
    var result = subject.Evaluate(_promotionData, _processorContext);
    Assert.Equal(FulfillmentStatus.PartiallyFulfilled, result.Status);
}
```



All tests are green, Yay!

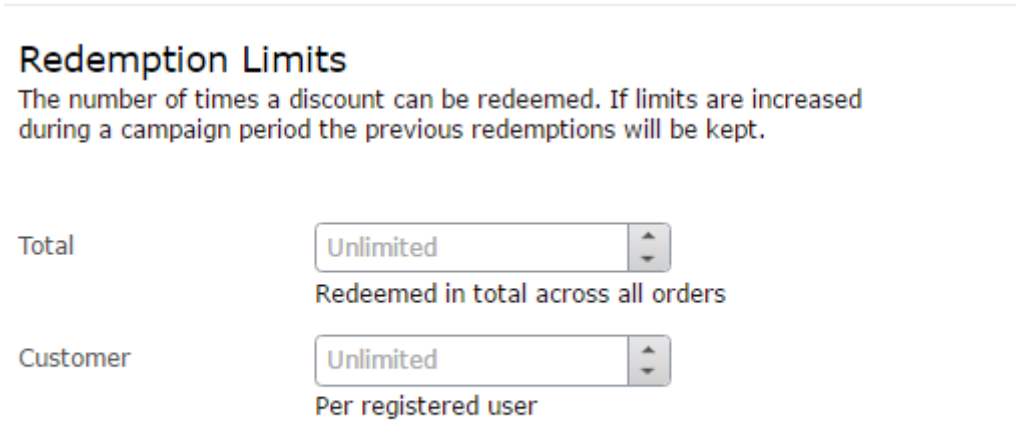
These tests have a pretty good test coverage. You can still test for the `CanBeFulfilled` method where the currency of the `IOrderGroup` does not match one of the currencies in `Amounts`. I'll leave that for you!

Redemption limit

We learned about the `RedemptionLimits` property of `PromotionData` to limit how many times can a promotion be applied. They are usually set manually by the editors - it's an easy job to do - but there are cases when you don't want it to be set - but fixed. For example, in case of an order-level promotion, it does make senses to allow only 1 per order. And we don't want editors to change that. How?

Promotion system has an atrribute `PromotionSettings` which has a property `FixedRedemptionsPerOrder`:

```
[PromotionSettings(FixedRedemptionsPerOrder = 1)]
[ContentType(GUID = "7660BDC2-6F65-4D22-8FDB-812EE7F4074D")]
[ImageUrl("Images/SpendAmountGetDiscount.png")]
public class SpendAmountGetDiscount : OrderPromotion
```



And the per-order redemption limit is now disabled

Many builtin promotions have this setting already, including any types derived from `OrderPromotion`.

Item Promotion

Now we create our first promotion with the new promotion system, I hope that you liked it - or at least found it easy enough. But we should not stop there. As an expression which is common in Sweden: "Practices make perfect" (I'm sure it's said everywhere, but a little more here), we should try to create another one to make sure we fully understand the system. Let's say we want to give a 10% discount on any items in a specific category. This is a builtin promotion in Episerver Commerce ("Buy From Category Get Discount") - but we'll implement ourselves to see how it works.

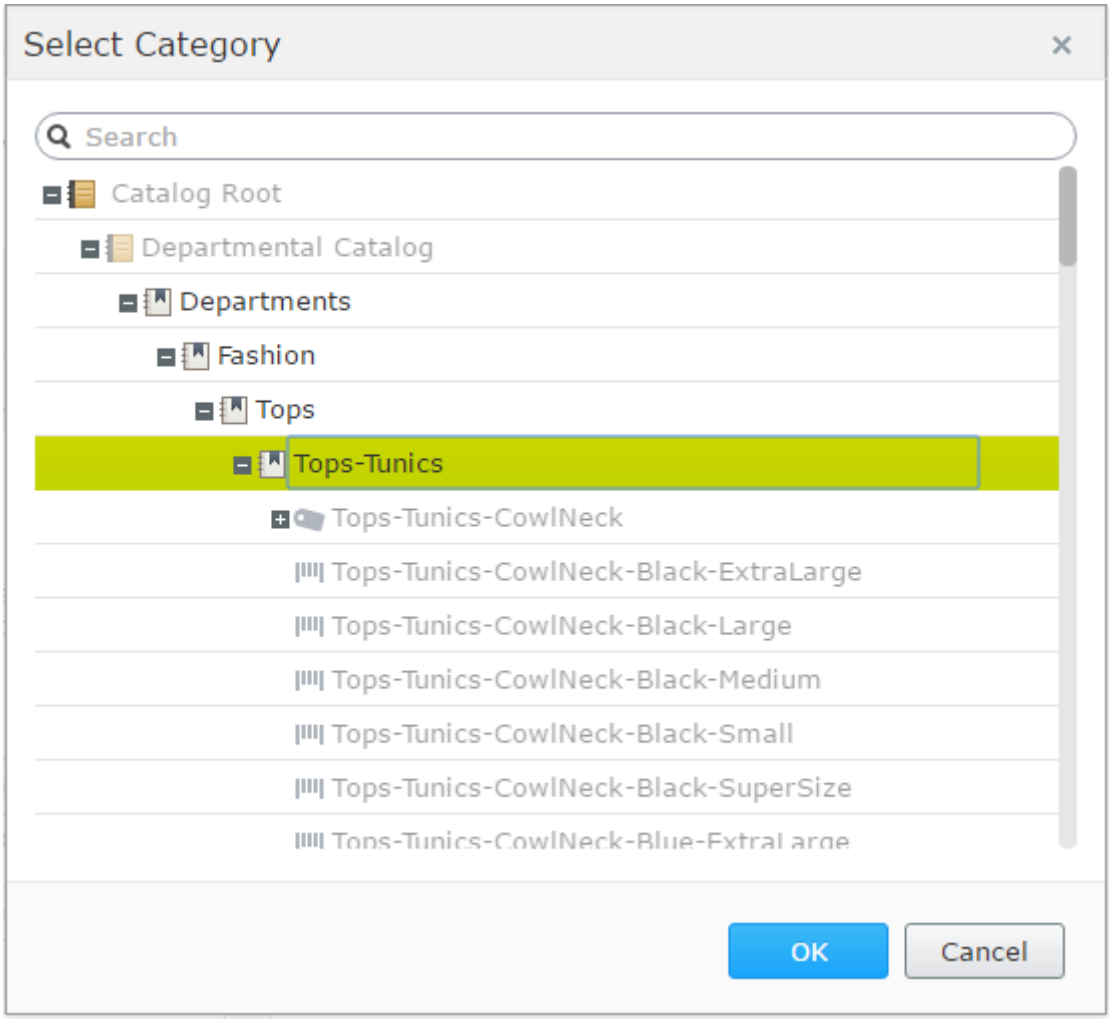
I don't want this. I want to have some new, interesting stuffs. However Episerver Commerce already contains all most common promotions. And I want to avoid too complicated promotions. As long as you understand the principles, you can create any promotions you like.

As usual, we need to define a promotion type:

```
[ContentType(GUID = "95F9301D-10D6-4C7F-AA3D-517FBDD560C4", GroupName = "entrypr\
omotion", Order = 12345)]
[ImageUrl("Images/ItemDiscountPromotion.png")]
public class ItemDiscountPromotion : EntryPromotion
{
    /// <summary>
    /// Any SKUs that belong to this category will get a discount.
    /// </summary>
    [AllowedTypes(typeof(NodeContent))]
    [Display(Order = 10)]
    [PromotionRegion(PromotionRegionName.Condition)]
    [UIHint(UIHint.CatalogNode)]
    public virtual ContentReference Category { get; set; }
}
```

To specify the category we want to give this discount, we need a `ContentReference` property. This has a lot of attributes decorated:

- `AllowedTypes` means this can only accept `NodeContent`, (and it derived classes). You won't be able to drag and drop other catalog content types to this property in On Page Editing mode.
- This property belongs to the "Condition" region of the promotion.
- You can only select a catalog node/category for this property.



Only category can be selected!

And then a processor:

```
public class ItemDiscountPromotionProcessor : EntryPromotionProcessorBase<ItemDi\
scountPromotion>
{
    protected override PromotionItems GetPromotionItems(ItemDiscountPromotion pr\
omotionData)
    {
        throw new NotImplementedException();
    }

    protected override RewardDescription Evaluate(ItemDiscountPromotion promotio\
nData, PromotionProcessorContext context)
    {
```

```
        throw new NotImplementedException();
    }

    protected override bool CanBeFulfilled(ItemDiscountPromotion promotionData, \
PromotionProcessorContext context)
    {
        throw new NotImplementedException();
    }
}
```

In Order promotion we talked about `GetPromotionItems` - however for order promotion it does not really matter. For this entry promotion, `GetPromotionItems` is important because it limits the number of entries we need to check.

```
protected override PromotionItems GetPromotionItems(BuyFromCategoryGetItemDiscou\
nt promotionData)
{
    var targetItems = new CatalogItemSelection(new[] { promotionData.Category },\
 CatalogItemSelectionType.Specific, true);
    return new PromotionItems(promotionData, targetItems, targetItems);
}
```

Here the condition and the reward are pointing to same `Category`. The final boolean parameter of `CatalogItemSelection` constructor is `MatchRecursive`, and it being true means if the `Category` contains sub-categories, then their entries will be included as well.

In this example we only set on category. However it's totally possible to have multiple categories, if we change the property to `IEnumerable<ContentReference>` instead. Try to make that change and update `GetPromotionItems` as well.

Next is the `CanBeFulfilled`. As we learned in previous promotion type - `CanBeFulfilled` is a shortcut for the promotion engine to know if it should proceed the promotion further, or just skip it. In this case, we should check if the `Category` is set and if the `IOrderForm` contains any items. I'll leave that to you as exercise.

The final piece is `Evaluate`. We will have to get all the descedents of the specified `Category`, get their codes, and then compare them to the lineitems in the cart. If one ore more of them match those codes, then we will give the discount. It does not sound difficult, but not easy either. Luckily for us, Episerver already provides `CollectionTargetEvaluator` to do exact the same thing. Its only public method, `GetApplicableCodes`, takes the collection of lineitems in the order, the collection of `ContentReference` defined by the promotion, and a flag (`boolean`) to indicate if it should check that collection recursively. (If the collection contains a category, and the flag is set to true, then it will look into sub-categories of that as well)

```
protected override RewardDescription Evaluate(TEntryPromotion promotionData, Pro\
motionProcessorContext context)
{
    var lineItems = GetLineItems(context.OrderForm); //GetLineItems is a protect\
ed method of PromotionProcessorBase
    var target = _targetGetter(promotionData);
    var affectedCodes = _targetEvaluator.GetApplicableCodes(lineItems, new [] { \
promotionData.Category }, true);

    if (!affectedCodes.Any())
    {
        return NotFulfilledRewardDescription(promotionData, context, Fulfillment\
Status.NotFulfilled);
    }

    ...
}
```

Now it's tricky part. If a customer buys two items from `Category`, is it counted as one, or two redemptions? What if he buys one item with quantity two? We will assume that both cases are counted as 1 redemption, for simplicity.

```
var affectedLineItems = lineItems.Where(c=> affectedCodes.Contains(c.Code));
//CreateRedemptionDescription is a protected method of EntryPromotionProcessorBa\
se
var redemption = CreateRedemptionDescription(new AffectedEntries(affectedLineIte\
ms.Select(c => new PriceEntry(c))));
return RewardDescription.CreatePercentageReward(FulfillmentStatus.Fulfilled, new\
 [] { redemption}, promotionData, 10.0m, "The promotion is fulfilled");
```

And we are done!

This is of course a simplified version of the builtin promotion. Handling the redemption count can be a complicated scenario when you need to support the decimal quantity. What if there are only 2 redemptions left, and there are 3 items qualified in the cart? That's done by a complex mechanism in the new promotion - involving `OrderFormPriceMatrix`. We might get to it later ;)

Shipping promotion

Now let's work with the final type of promotion - shipping promotion. One of the shipping promotion I see the most here in Sweden is "Buy items with at least value of <something>, get

free shipping" - which we are going to implement now. This is actually an builtin promotion in Episerver Commerce, but for educational purpose - let's try to implement that promotion in new system.

```
[ContentType(GUID = "87771CF4-2ABE-4DB2-835A-7467575B772B", GroupName = "shippin\
gpromotion", Order = 12345)]
[PromotionSettings(FixedRedemptionsPerOrder = 1)]
[ImageUrl("Images/SpendAmountGetFreeShipping.png")]
public class SpendAmountGetFreeShipping : ShippingPromotion
{
    /// <summary>
    /// The purchase amount required for the promotion to apply.
    /// </summary>
    [Display(Order = 10)]
    [PromotionRegion(PromotionRegionName.Condition)]
    public virtual PurchaseAmount Condition { get; set; }
}
```

You're now familiar with all of these attributes. The only thing new here is the `ShippingPromotion` - the base type for shipping promotions. Unlike `OrderPromotion` or `EntryPromotion`, which only contain the `DiscountType` property, `ShippingPromotion` also contain the list of `Guid` of shipping methods:

```
public abstract class ShippingPromotion : PromotionData
{
    /// <inheritdoc />
    public override DiscountType DiscountType
    {
        get { return DiscountType.Shipping; }
    }

    /// <summary>
    /// A collection of shipping method ids determining which shipping methods c\
an be applied.
    /// </summary>
    [Display(Order = 20)]
    [PromotionRegion(PromotionRegionName.Reward)]
    [UIHint(SystemUiHint.ShippingMethodList)]
    public virtual IList<Guid> ShippingMethods { get; set; }
}
```

`ShippingMethods` are selectable via Marketing UI, so an editor can specify if one shipping method (for example, standard shipping) can be free, while customers still have to pay for other ones (for example, express shipping), even if the order has fulfilled the condition.

`PurchaseAmount` is an out-of-box block which contains the list of money amounts and the `PartiallyFulfilledThreshold` (which serves same purpose as `AlmostFulfilledThreshold` we had).

```
public class PurchaseAmount : ConditionBlock
{
    public virtual IList<Money> Amounts { get; set; }
    //From ConditionBlock
    public virtual decimal PartiallyFulfilledThreshold { get; set; }
}
```

By default, `PartiallyFulfilledThreshold` is set to 0.95m, but you can easily set it to another value. (At this point of writing, it can be only set by code, and not available to editors in the UI. That's why we had our `AlmostFulfilledThreshold` to see how it works, and editors can easily set that value).

Now it's the time to implement the processor. Episerver Commerce provides to base processors for shipping promotion, `ShippingPromotionProcessorBase<T>` which inherits from `PromotionProcessorBase` which provides some methods to work with `Shipments`, and `FreeShippingPromotionProcessorBase` which extends it even further. For educational purposes, we should implement our processor by inheriting from `PromotionProcessorBase`, however due to some limitations (some constructors can't be accessed from APIs) - we will need to stick with at least `ShippingPromotionProcessorBase`.

```
public class SpendAmountGetFreeShippingProcessor : ShippingPromotionProcessorBas\
e<SpendAmountGetFreeShipping>
{
    protected override FulfillmentStatus GetFulfillmentStatus(SpendAmountGetFree\
Shipping promotionData, PromotionProcessorContext context)
    {
        throw new NotImplementedException();
    }

    protected override RewardDescription CreateRewardDescription(SpendAmountGetF\
reeShipping promotionData, FulfillmentStatus fulfillmentStatus, IEnumerable<Rede\
mptionDescription> redemptions, PromotionProcessorContext context)
    {
        throw new NotImplementedException();
    }

    protected override PromotionItems GetPromotionItems(SpendAmountGetFreeShippi\
ng promotionData)
    {
        throw new NotImplementedException();
    }
```

```

    protected override bool CanBeFulfilled(SpendAmountGetFreeShipping promotionD\
ata, PromotionProcessorContext context)
    {
        throw new NotImplementedException();
    }
}
```

Now it's time to implement the required methods - as they are abstract in the base class. First, we can try the two simpler methods. `GetPromotionItems` should return all items, because we don't really care about the items inside the cart:

```
protected override PromotionItems GetPromotionItems(SpendAmountGetShippingDiscou\
nt promotionData)
{
    return
        new PromotionItems(
            promotionData,
            new CatalogItemSelection(null, CatalogItemSelectionType.All, true),
            new CatalogItemSelection(null, CatalogItemSelectionType.All, true));
}
```

And `CanBeFulfilled` - which checks if the currency of the cart is one of the currencies specified in the `PurchaseAmount`, and if the cart has any shipping method matched with the ones specified in `ShippingMethods`

```
protected override bool CanBeFulfilled(SpendAmountGetFreeShipping promotionData,\
 PromotionProcessorContext context)
{
    var purchaseAmount = promotionData.Condition.PurchaseAmount;
    var validCurrency = purchaseAmount != null && purchaseAmount.Amounts != null\
 && purchaseAmount.Amounts.Any(c => c.Amount > 0 && c.Currency == orderCurrency)\
;
    var orderForm = context.OrderForm;
    var validShippingMethod = orderForm.Shipments != null && orderForm.Shipments\
.Any() &&
                shippingPromotion.ShippingMethods != null && shippingPromotion.Sh\
ippingMethods.Any() &&
                orderForm.Shipments.Any(shipment => shippingPromotion.ShippingMet\
hods.Contains(shipment.ShippingMethodId));
    return validCurrency && validShippingMethod;
}
```

The code above shows how it's done internally, but we can of course use the extension method provided by Episerver Commerce. It can be rewritten as following:

```
protected override bool CanBeFulfilled(SpendAmountGetFreeShipping promotionData,\
 PromotionProcessorContext context)
{
    return promotionData.Condition.CanBeFulfilled(context.OrderGroup.Currency) &&
            promotionData.CanBeFulfilled(context.OrderForm);
}
```

Now it's time for more complex methods. As we learned previously, `GetFulfillmentStatus` should return one of three value: `NotFulfilled` if the promotion is not fulfilled at all. `PartiallyFulfilled` for the case when the `AlmostFulfilledThreshold` is reached, and `Fulfilled` when the conditions are satisfied. As we only care about the `SubTotal` of the cart, and we already have `CanBeFulfilled` to filter out the cases when it can't be fulfilled, the method should be easy enough:

```
    protected override FulfillmentStatus GetFulfillmentStatus(SpendAmountGetFree\
Shipping promotionData, PromotionProcessorContext context)
    {
        var subTotal = context.CurrentOrderFormSubTotal;
        if (subTotal >= conditionAmount.Amount)
        {
            return FulfillmentStatus.Fulfilled;
        }

        if (subTotal >= conditionAmount.Amount * promotionData.PurchaseAmount.Pa\
rtiallyFulfilledThreshold)
        {
            return FulfillmentStatus.PartiallyFulfilled;
        }

        return FulfillmentStatus.NotFulfilled;
    }
```

Now it's time for the heart of the processor `Evaluate` method.

```
    /// <summary>
    /// Evaluates the shipping promotion.
    /// </summary>
    /// <param name="promotionData">The promotion model.</param>
    /// <param name="context">The context for the promotion processor evaluation\
.</param>
    protected override RewardDescription Evaluate(TShippingPromotion promotionDa\
ta, PromotionProcessorContext context)
    {
        var fulfillmentStatus = GetFulfillmentStatus(promotionData, context);

        if (!fulfillmentStatus.HasFlag(FulfillmentStatus.Fulfilled))
        {
            return RewardDescription.CreateFreeItemReward(
                    fulfillmentStatus,
                    Enumerable.Empty<RedemptionDescription>(),
                    promotionData,
                    fulfillmentStatus.GetRewardDescriptionText(_localizationServ\
ice));
        }

        var redemptions = GetRedemptions(promotionData, context.OrderForm);

        return RewardDescription.CreateFreeItemReward(
         fulfillmentStatus,
         redemptions,
         promotionData,
         fulfillmentStatus.GetRewardDescriptionText(_localizationService);
    }
```

`GetRedemptions` is an protected method from `ShippingPromotionProcessorBase`, and it's the reason we had to inherit from it. It returns a list of `RedemptionDescription` which each of them defines the objects which should be affected by the promotion. In our case, it's the shipments being free.

Coupon codes

Coupon codes (or promotion codes) are an essential part of promotions. In the previous chapter, we learned on how to work with coupons in old promotion system - adding the coupons to `MarketingContext` and how to "persist" them between requests. In the new system, it's much simple to add the coupons to the cart:

```
cart.GetFirstForm().CouponCodes.Add(coupon);
```

CouponCodes is an ICollection<string> property of IOrderForm. And that's it, this will be carried between requests without you have to deal with Session or cookie. Once you save the cart, the coupons will be there - even when you converted the cart to a purchase order. It's easily accessible via IOrderForm.CouponCodes until you explicitly remove it.

Before being "applied", the promotions will be filtered by ICouponFilter. The default implementation will make sure that if no coupon was supplied to the cart, the promotions which require coupon codes will be filtered out - so the promotion engine won't have to process them - saving time and resources.

ICouponFilter is a simple interface with only one method:

```
public interface ICouponFilter
{
    /// <summary>
    /// Filters promotions by supplied coupon codes.
    /// </summary>
    /// <param name="filterContext">The filter context.</param>
    /// <param name="couponCodes">The coupon codes added to the current order fo\
rm.</param>
    /// <returns>
    PromotionFilterContext Filter(PromotionFilterContext filterContext, IEnumera\
ble<string> couponCodes);
}
```

The default implementation of ICouponFilter makes sure that: If no coupon was supplied, then all promotions which require coupons will be filtered. If one or more coupons were supplied, then all promotions which coupons does not match the supplied coupons will be filtered. This will limit the number of promotions which needs to be applied - and help improving performance.

The default implementation works with case when each promotion can only have one coupon. In case you have multiple coupon codes per promotion, you will have to implement ICouponFilter yourself.

Promotions exclusion

As we learned in previous section, in old promotion system, you only have three options for exclusion: A promotion might exclude nothing, exclude all promotions in same group, or exclude everything. This works to some extend, but it's not flexible enough. The new promotion system provides much more powerful exclusion mechanism, to allow you quickly set the exclusion combination between promotions.

Programmatically, the exclusion settings for a promotion can be set by `ExcludedItems`, which is a collection of ContentReferences you want to be exclusive with the promotion. - If it is `CampaignRoot`, then the promotion will exclude all other promotions. - If it is a `SalesCampaign`, then the promotion will exclude all promotions in that campaign.

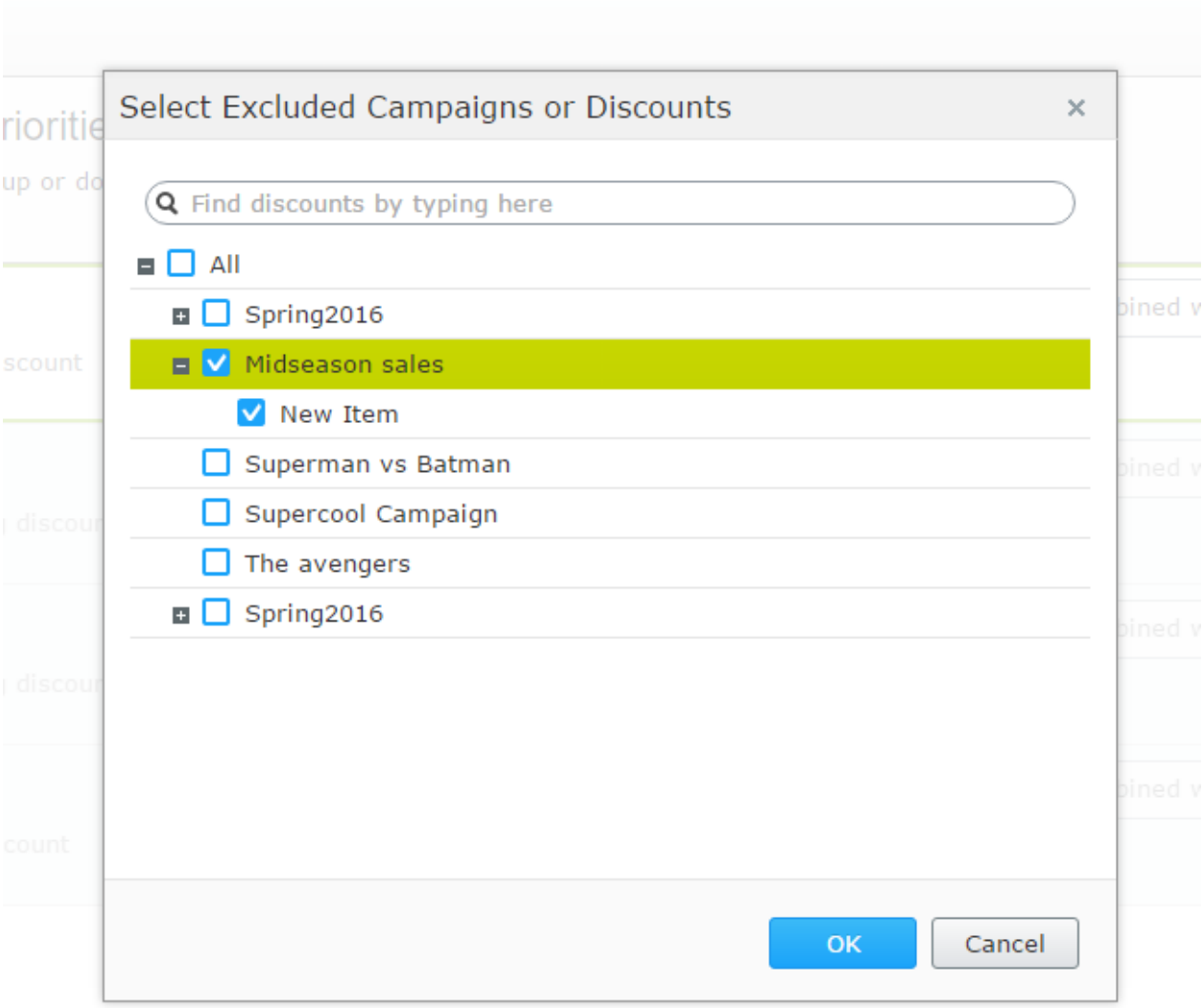
It's worth noting that for now, this is the way you should do it:

```
promotion.ExcludedItems = new List<ContentReference>() { otherPromotion.ContentL\
ink, summerCampaign.ContentLink };
```

When CMS 10 is out, you can write like this:

```
//Exclude other promotion.
promotion.ExcludedItems.Add(otherPromotion.ContentLink);
//Excluded promotions in summer campaign.
promotion.ExcludedItems.Add(summerCampaign.ContentLink);
```

The exclusion setting comes in pair with priority, it's basically the same rule as the old promotion system: a promotion with higher priority will exclude the promotions with lower priorities, not the way around. As I said, the exclusion setting in new promotion system is very flexible:



You can exclude anything and everything

Items exclusion

In above section we talked about excluding specific promotions from be ing applied. How's about excluding all promotions for a specific item. Some items, such as gift cards, can be considered as money and should not be discounted in any situations (otherwise some sneaky customers might find a loophole to eventually get something for free, you don't want that, do you?). In the old promotion system, this is simply impossible - you don't have anyway to do that. That's why the new promotion system is superior, at this time of writing, a new feature is just available - allows you to decide which items should be hidden from the promotion system, entirely.

This can be done via `IEntryFilter`, a simple interface which filters out entry codes which should not be checked by promotion engine:

```
public interface IEntryFilter
{
    /// <summary>
    /// Filters out entry codes which should be excluded by the promotion engine.
    /// </summary>
    /// <param name="entryCodes">The entry codes to be filtered.</param>
    IEnumerable<string> Filter(IEnumerable<string> entryCodes);
}
```

You can of course implement your own implementation, however, the default implementation is already very extensible and easy to use - with `EntryFilterSettings` you can practically exlude everything you want, both programmatically and manually. `EntryFilterSettings` has several methods, but the most important one is `AddFilter<T>(Predicate<T> filter)` - which allow you to specify which criterias to filter out entries. Here is what you can do in an `IInitializationModule`:

```
var filterSettings = context.Locate.Advanced.GetInstance<EntryFilterSettings>();
//Clear all filter
filterSettings.ClearFilters();

//Add filter predicates for a whole content type.
filterSettings.AddFilter<TypeThatShouldNeverBeIncluded>(x => false);

//Add filter predicates for property of strongly typed content types.
filterSettings.AddFilter<IInterfaceWithShouldBeExcludedProperty>(x => !x.ShouldB\
eExcluded);

//Add filter predicates for property of non strongly typed content types (proper\
ties which are created by metafields)
filterSettings.AddFilter<EntryContentBase>(x => !(bool)(x["ShouldBeExcludedMetaF\
ield"] ?? false));

//Add filter predicates for specific entries
var ExcludingCodes = new string[] { "SKU-36127195", "SKU-39850363", "SKU-3910125\
3" };
filterSettings.AddFilter<EntryContentBase>(x => !ExcludingCodes.Contains(x.Code)\
);
```

Those are pretty “programmatically” - so how about “manually”? In the above example, you can have a predicate so if an entry has a property (“Excluded by promotion engine”, for example) set to true, then it should be excluded by the promotion engine. When editors edit the entries, they can manually set that value in specific entries to be true. And voilà!

Getting discounted prices

So far, the promotion engine only works on orders. But a common case is to display the discounted prices of products when a customer is browsing the site - aka visiting the product listing page or checking the search result. This is of course important - a product at 100$ is surely not as attractive as itself, with 20$ discount so it only costs 80$. How?

In the old promotion system, you have `StoreHelper.GetDiscountPrice`, which take an `Entry` and some other parameters, to do the (almost) same thing. (`StoreHelper` is a class in `Mediachase.Commerce.Website`, which is an open source project. You can download it and have fun!). So the most common way to get discounts for product listing page is to load the `Entry` instance for each entry, and call to that method to get the discounted price. It's undoubtedly slow. What's about the new promotion system?

It comes with some extension methods for `IPromotionEngine`:

```
    public static IEnumerable<DiscountedEntry> GetDiscountPrices(this IPromotion\
Engine promotionEngine,
        ContentReference entryLink,
        IMarket market);

    public static IEnumerable<DiscountedEntry> GetDiscountPrices(this IPromotion\
Engine promotionEngine,
        ContentReference entryLink,
        IMarket market,
        Currency currency);

    public static IEnumerable<DiscountedEntry> GetDiscountPrices(this IPromotion\
Engine promotionEngine,
        IEnumerable<ContentReference> entryLinks,
        IMarket market,
        Currency marketCurrency);
```

These methods will return a list of `DiscountedEntry`, which has the information about the entry (the `entryLink`, actually), the default price, the discounted price and the promotion associated with it. It can be called with one or multiple `ContentReference` - a big plus compared to the old promotion. `ContentReference` is also faster, easier to get and use less memory than `Entry`.

How is it done internally? The promotion system will create an in-memory order, which contains that entry (with quantity = 1, in case of multiple entries, each will be evaluated individually). It then try to evaluate the promotions on that order. Of course, any promotions which require coupon, or at least 2 items to be fulfilled, will not be applied.

Disable built-in promotions

At this time of writing, Episerver Commerce already has more than 10 built-in promotion types - they are quite enough to cover almost every case of promotions. However, sometimes you don't want to use those built-in types, but your custom types, and you want to hide the built-in ones so to make it's easier for editors to work with promotions. How?

If we recall from the beginning of this chapter, promotion types are just content types - so if you want to disable a specific promotion, you can just set the availability of the corresponding content type to false - it'll be hidden.

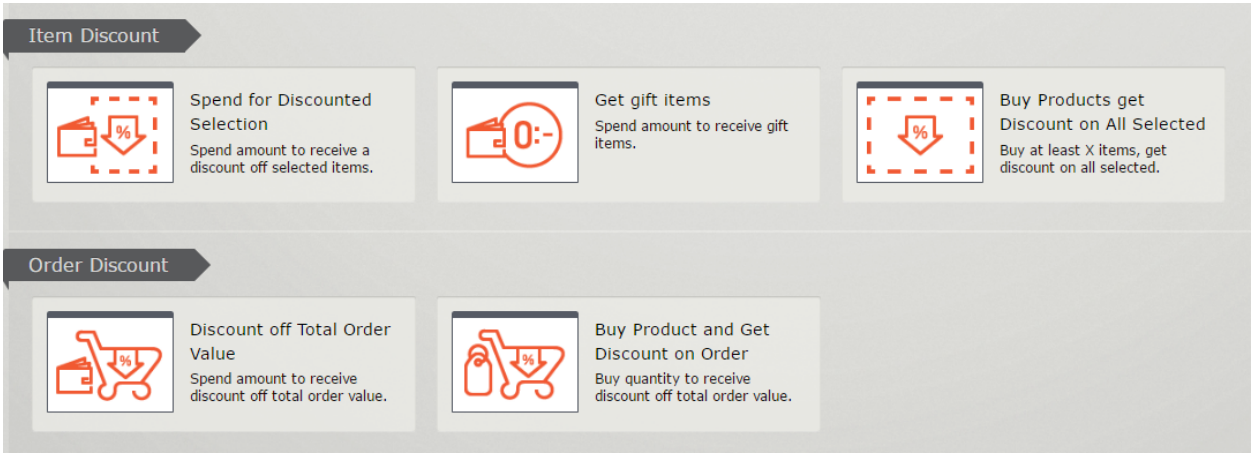
```
var type = typeof(SpendAmountGetDiscount);
var contentType = ServiceLocator.Current.GetInstance<IContentTypeRepository>.Loa\
d(type).CreateWritableClone() as ContentType;
if (contentType.IsAvailable)
{
    contentType.IsAvailable = available;
    _contentTypeRepository.Save(contentType);
}
```

Luckiky, you won't have to write such code yourself. Commerce 9.20 comes with new `PromotionTypeHandler` which allows you to disable/enable promotion types with ease:

```
public class PromotionTypeHandler
{
    public void DisablePromotions(IEnumerable<Type> promotionTypes);
    public void EnablePromotions(IEnumerable<Type> promotionTypes);
    public void DisableBuiltinPromotions();
    public IEnumerable<Type> GetAllPromotionTypes();
}
```

So let's say we want to disable all of the shipping promotion types - because you are selling digital content and it does not really make senses to have such promotions - for example. So we should add this to an initialization module:

```
private void DisablePromotionTypes(InitializationEngine context)
{
    var promotionTypeHandler = context.Locate.Advanced.GetInstance<PromotionType\
Handler>();
    var promotionTypes = promotionTypeHandler.GetAllPromotionTypes();
    var shippingPromotionTypes = promotionTypes.Where(t => t.IsSubclassOf(typeof\
(ShippingPromotion)));
    promotionTypeHandler.DisablePromotions(shippingPromotionTypes);
}
```



And then the shipping promotion types won't bother you again.

New promotion system in Commerce Manager

The new promotion system basically works in Commerce Manager, what's the point of having promotions that cannot be applied when you manage the orders? However, due to nature of Commerce Manager and strategy from Episerver, there are extra steps needed to make it work:

- The `VNextFeature` flag needs to be enabled as Commerce Manager relies on the workflows for discounts calculation, either by code of by `ecf.app.config`. (If your site was created newly from 9.19 or later, this is no longer needed. However if you use an older version, or you upgrade from an older version, this must be enabled manually.)
- Your promotion content types must be present in Commerce Manager bin. For that reason, you should place all of your promotion content types in one project (along with your shipment gateways and payment providers), to make it easier to manage.

In 9.19, all of known issues have been addressed and some new small UI tunings have been added to Commerce Manager. The Marketing section is also disabled if the `VNextFeature` flag is enabled:

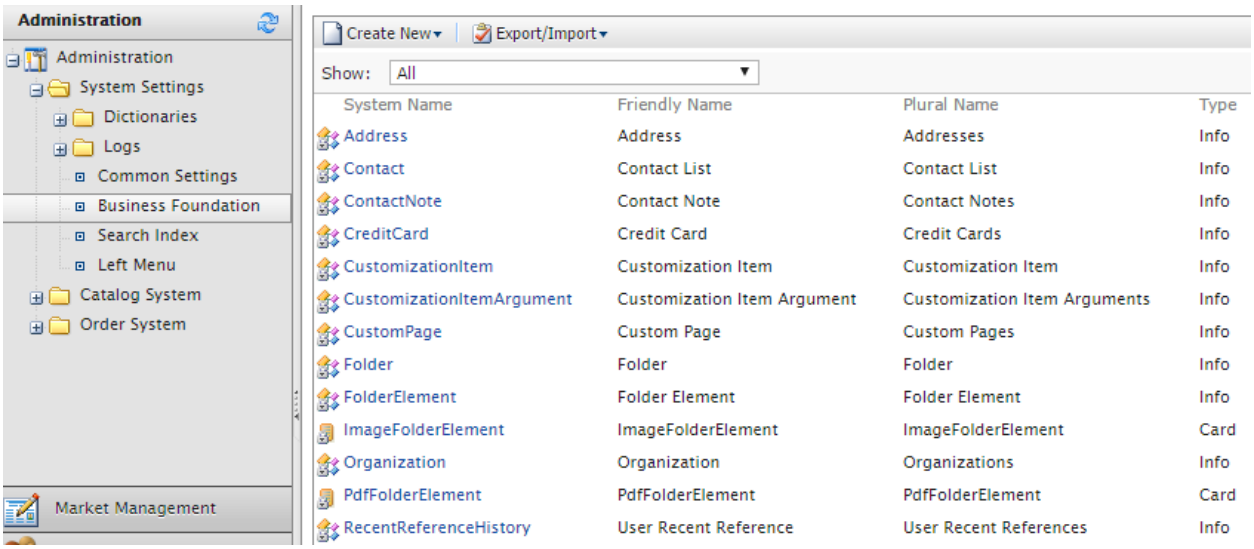
Goodbye, old promotion system!

Part 7: Business Foundation

This chapter is dedicated to one of the most advance and in some ways, most mysterious features in Episerver Commerce - Business Foundation. What is it, how does it work, how can it be extended.

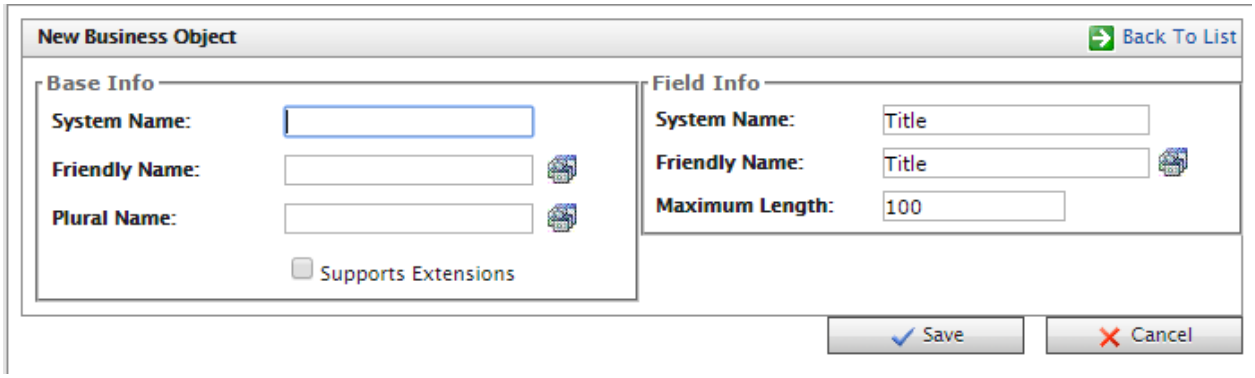
Chapter 14: Business Foundation overview

If you're new to Episerver Commerce, Business Foundation is a quite hidden feature. In a standard Commerce installation, the only place where you can find BF is in Administration/System Settings of Commerce Manager.



Configure business foundation objects in Commerce Manager

Here you can see the list of predefined business objects. Some of them are obsolete and are almost unused anywhere (`FolderEntity`, `FolderElementEntity` for example). But you'll likely to work with `Address`, `Contact`, `ContactNote` and `CreditCard`. Like the MetaClass system for catalog and order, you can define your own business object.



Create new Business Object in Commerce Manager

And you can import and export the settings. (Unlike the catalog system, where you must export the classes with the data, here you can export and import only the definition of classes.)

Business Objects are built around `EntityObject`, which itself is a simple class:

```
public class EntityObject
{
        public EntityObject(string metaClassName);

        public EntityObject(string metaClassName, PrimaryKeyId primaryKeyId);

        public string MetaClassName { get; set;}

        public PrimaryKeyId? PrimaryKeyId { get; set;}

        public EntityObjectPropertyCollection Properties { get; set;}

        public object this[string name] { get; set;}
}
```

Each `EntityObject` is identified by its `PrimaryKeyId` and `MetaClassName`. `PrimaryKeyId` is yet another `struct` to identify an object - it can take an `int` or a `Guid`.

If you recall from Part 1 - there are two classes named `MetaClass`. The one in `Mediachase.BusinessFoundation.Data.Business` is the one we need right now.

`EntityObject` is just the base class. Every business object extends it to add necessary fields and relations. However, all of are still managed by `BusinessManager` - the heart of Business Foundation. `BusinessManager` provides plenty of CRUD - Create, Read (both single object and list), Update and Delete - operations to work with `EntityObject`.

Here's an example to create an `Organization`, which of course, inherits from `EntityObject`. `Organization` is one of Business Objects available out-of-box.

```
var organization = new Organization();
organization.Name = "Umbrella Corp";
organization.OrganizationType = "Evil Corporation";
var organizationId = BusinessManager.Create(organization);
```

In this example, we are using `BusinessManager.Create(EntityObject object)` to create a new `EntityObject`, and return the `PrimaryKeyId` of the newly created object.

And then we can load it back, and edit a field, then save:

```
var organization = BusinessManager.Load("Organization", organizationId);
organization.OrganizationType = "Defunded Evil Corporation";
BusinessManager.Update(organization);
```

"Organization" is the name of the `MetaClass`. We can also use `List` which returns the list of objects:

```
public static EntityObject[] List(string metaClassName, FilterElement[] filters)
public static EntityObject[] List(string metaClassName, FilterElement[] filters,\
 SortingElement[] sorting)
public static EntityObject[] List(string metaClassName, SortingElement[] sorting)
public static EntityObject[] List(string metaClassName, FilterElement[] filters,\
 SortingElement[] sorting, int? start, int? count)
```

`FilterElement` is a simple class which helps you to filter the list based on some conditions:

```
public class FilterElement
{
        public virtual string Source {get; set;}
        public virtual FilterElementType Type {get; set;}
        public virtual object Value {get; set;}
        public bool ValueIsTemplate { get; set;}
}
```

Basically, one `FilterElement` means we can filter business objects which has `Source` field which match (or not match) `Value`. Value can be a template string. `Type` can be one of those values: `Equal`, `NotEqual`, `Less`, `LessOrEqual`,...

And finally, we can delete it:

```
BusinessManager.Update("Organization", organizationId);
```

Internally, all operations from `BusinessManager` are built around this single, simple method:

```
Response Execute(Request request)
```

`Request` is a base class of all others "request" classes, it contains the `EntityObject` to be worked on, and the method of that work

```
public class Request
{
        public EntityObject Target {get; set;}
        public string Method {get; set;}
}
```

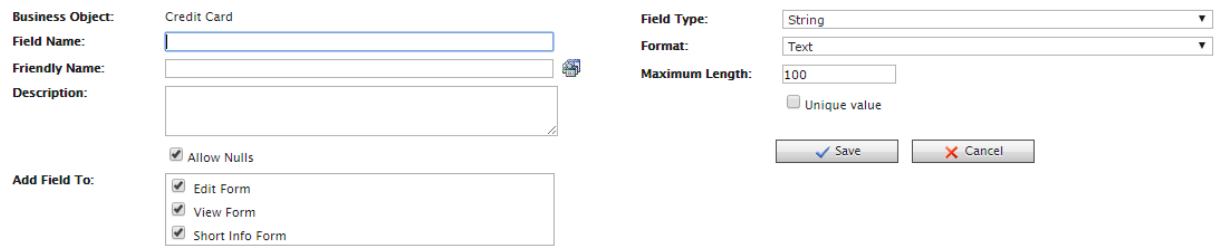
The request classes are only different in `RequestMethod` property:

```
public static class RequestMethod
{
        public const string InitializeEntity = "InitializeEntity";

        public const string Create = "Create";
        public const string Update = "Update";
        public const string Delete = "Delete";
        public const string Load = "Load";
        public const string List = "List";

        public const string Import = "Import";
        public const string InitializeMappingDocument = "InitializeMappingDocument";
}
```

So you can guess, a `DeleteRequest` is a `Request` with `RequestMethod.Delete` - and so on and so forth. Simple, right?

Extending Business Objects

If you recall about `MetaDataPlus` metadata system, then you know about the `MetaClass`(es) and `MetaField`(s), and the relations between them. With Business Foundation, every thing is almost the same, we also have `MetaClass` and `MetaField`, but instead of `Mediachase.MetaDataPlus`, we use `Mediachase.BusinessFoundation.Data` here. And instead of `MetaObject` for each instance, we have `EntityObject` as we talked in previous section.



Adding new `MetaField` to a `MetaClass`

Business Object can be extended by Business Foundation section in Commerce Manager, but of course it can be extended by code. The legacy way to extend Business Object is to use `IPlugin` interface. it's a quite simple one with only one method:

```
void Execute(BusinessContext context);
```

Whenever `BusinessManager.Execute(Request request)` is called, it will try to load all classes which implement `IPlugin` and run them (!). So we will have to use some mechanism to make sure that the module was not executed, and that check should be really fast. But that was pre-Episerver era, and for now we should use `IInitializationModule` (or even better, Migration steps) to do such task. We then only need to check for that module once on startup.

This is the check to see if we should continue creating the metaclass (named "CreditCard". It's simple as that, every `MetaClass` in Business Foundation is available in `DataContext.Current.MetaModel.MetaClasses`. {lang=C#} if (DataContext.Current.MetaModel.MetaClasses["CreditCard"] == null) { return; }

```
MetaClassManager manager = DataContext.Current.MetaModel;
```

using (MetaClassManagerEditScope scope = manager.BeginEdit(MetaClassManagerEditScope.SystemOwner, AccessLevel.System))

```
{
        ModuleManifest changeTrackingManifest = ChangeTrackingManager.CreateModuleManif\
est();
        MetaClass mc = manager.CreateMetaClass("CreditCard", "Credit Card", "CreditCard\
s", "cls_CreditCard", PrimaryKeyIdValueType.Guid);

        ModuleManager.Activate(mc, changeTrackingManifest);
        using (MetaFieldBuilder builder = new MetaFieldBuilder(mc))
        {
```

```
                mc.TitleFieldName = "CreditCardNumber";
                builder.CreateText("CreditCardNumber", "Credit Card Number", false, 512, false\
);
                builder.CreateEnumField("CardType", "Credit Card Type", "CreditCardType", fals\
e, string.Empty, true);
                builder.CreateText("LastFourDigits", "Last Four Digits", false, 4, false);
                builder.CreateText("SecurityCode", "Security Code", true, 255, false);
                builder.CreateInteger("ExpirationYear", "Expiration Year", true, null);
                builder.CreateInteger("ExpirationMonth", "Expiration Month", true, null);
                mc.Fields[MetaClassManager.GetPrimaryKeyName(mc.Name)].FriendlyName = "Primary\
 Key id";
                builder.SaveChanges();
        }
        scope.SaveChanges();
}
```

We are creating `CreditCard` class using `DataContext.Current.MetaModel.CreateMetaClass`. (Note that this class is available by default in Commerce installation, we are using it here for example). Objects in this class will be saved to `cls_CreditCard` table in your database, with a primary key of type `Guid`.

Of course you can choose the name of table as you like, but it's conventionally set to "cls_" prefix - as a shorthand of "class".

Next step is to add fields to the class - `MetaFieldBuilder` supports all the types available to `MetaField`:

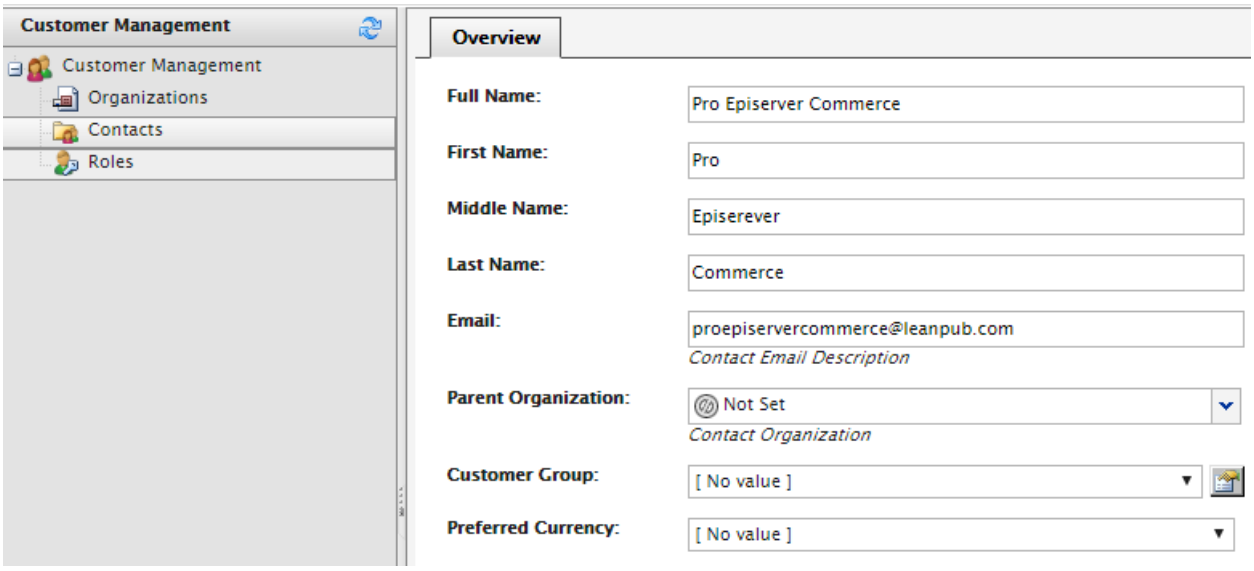
```
public MetaField CreateGuid(string name, string friendlyName, bool isNullable)
public MetaField CreateDateTime(string name, string friendlyName, bool isNullabl\
e, bool useUserTimeZone)
public MetaField CreateDate(string name, string friendlyName, bool isNullable)
public MetaField CreateInteger(string name, string friendlyName, bool isNullable\
, int? defaultValue)
public MetaField CreateFloat(string name, string friendlyName, bool isNullable, \
double? defaultValue)
public MetaField CreateCurrency(string name, string friendlyName, bool isNullabl\
e, decimal? defaultValue, bool allowNegative)
public MetaField CreateCheckBoxBoolean(string name, string friendlyName, bool is\
Nullable, bool? isChecked, string label)
public MetaField CreateDropDownBoolean(string name, string friendlyName, bool is\
Nullable, bool? isChecked, string trueLabelText, string falseLabelText)
```

```
public MetaField CreateText(string name, string friendlyName, bool isNullable, i\
nt maxLength, bool isUnique)
public MetaField CreateEmail(string name, string friendlyName, bool isNullable, \
int maxLength, bool isUnique)
...
```

The final step is to save the builder and then save the entire `scope`.

Customer Contact

Probably the most used Business Object of all - customer contact receives special treatments - it has it own predefined class - `CustomerContact`. At its core, `CustomerContact` is still `EntityObject`, and it's still managed by `BusinessManager`, but it's important enough for Episerver to provider some convenient methods to make `CustomerContact` handling easier.



Creating a new contact in Commerce Manager

`CustomerContact` extends `EntityObject` by adding a bunch of properties, such as `FirstName`, `LastName`, or `Email`, etc. It also has two collections, `ContactAddresses` and `ContactCreditCards`, which contains the contact's addresses and credit cards, respectively.

`CustomerContact` is handled by `CustomerContext` - which can be used to get `CustomerContact`(s) or `Organization`(s). For example:

```
var customerContact = CustomerContext.Current.GetContactById(contactId);
customerContact.PreferredLanguage = "en-US";
customerContact.SaveChanges();
```

The full list of methods and properties of `CustomerContext` can be found here[50]

Customer Address

Almost always tied to Customer Contact, Customer Address is something that worth your attention.

There are two addresses that special enough - `PreferredShippingAddress` and `PreferredBillingAddress`

Episerver Commerce has multiple methods to convert a customer address to an order address, and vice versa. This is very convenient when a registered user wants to check out an order:

```
var customerAddresses = CustomerContext.Current.CurrentContact.CustomerAddresses;
foreach(var customerAddress in customerAddresses)
{
        var orderAddress = new OrderAddress();
        CustomerAddress.CopyCustomerAddressToOrderAddress(customerAddress, orderAddress\
);
        order.OrderAddresses.Add(ordereAddress);
}
```

And when a customer decides to register for an account after placing an order:

```
var customerAddresses = CustomerContext.Current.CurrentContact.CustomerAddresses;
foreach(var orderAddress in order.OrderAddresses)
{
        var customerAddress = new AddressEntity();
        OrderAddress.CopyCustomerAddressToOrderAddress(orderAddress, customerAddress);
        customerAddresses.Add(customerAddress);
}
```

And of course we can set `PreferredBillingAddress` and `PreferredShippingAddress` for the `CustomerContact`.

[50] http://world.episerver.com/documentation/Class-library/?documentId=commerce/9/21F44EB1

Part 8: Extending the system.

Episerver Commerce is not just a software. It IS a framework. Out of the box, it should work with basic scenarios, but you will need something specialized for your implementation. You might want to talk with a CRM system, or to use a PIM to update catalog, or to get your ERP to handle the orders for you. Good news: Episerver Commerce allows that. Most of the important parts of the system can be replaced by your implementations. Then how? Let's discover it.

Plug your implementations - Episerver Framework way.

There are many ways to plug your implementations into Episerver Commerce. Mediachase eCF has its ways, but we'll be talking about the method provided by Episerver Framework, first.

There are two initialization modules from Commerce you might want to care about - the `CommerceInitialization` from `Mediachase.Commerce.Initialization`, and `InitializationModule`, from EPiServer.Commerce.Initialization. The latter depends on the former (so it's guaranteed to run *after*). Why should you care? `CommerceInitialization` will handle almost every dependencies with core stuffs, such as `ICatalogSystem`, `IInventoryService`, `IMarketService`, etc. `InitializationModule` will handle the stuffs related to catalog content and routing. You will know which module to depend on (of course if you depend on `InitializationModule`, you will automatically depend on `CommerceInitialization` as well).

However, their names do not tell the entire truth, in facts, they inherit from `IConfigurableModule`, which itself inherits from `IInitializationModule`

Here's how you can implement your `IConfigurableModule`:

```
[ModuleDependency(typeof(EPiServer.Commerce.Initialization.InitializationModule)\
)]
public class InitializationModule : IConfigurableModule
{
    public void Initialize(InitializationEngine context)
    {
        //Do your stuffs here, such as register for events.
    }

    public void Uninitialize(InitializationEngine context)
    {
        //Undo your stuffs, such as unregister the events.
    }

    public void ConfigureContainer(ServiceConfigurationContext context)
    {
        //Configure the dependencies
        context.Services.AddSingleton<ICurrentMarket, MarketStorage>();
    }
}
```

In the example above, we add `MarketStorage` as the singleton instance for `ICurrentMarket`, that means if you call this in your code:

```
var currentMarketService = ServiceLocator.Current.GetInstance<ICurrentMarket>();
```

You'll get the same instance of `MarketStorage` every time. `AddSingleton` is just one of 5 ways to register the life-cycle of your dependencies - `ServiceConfigurationProviderExtensions` provides plenty extensions methods for `IServiceConfigurationProvider`, but in short you can choose between:

- `AddSingleton<TService, TImplementation>`, as above
- `AddTransient<TService, TImplementation>`: new instance of `TImplementation` will be created every time `TService` is requested.
- `AddScoped<TService, TImplementation>`: registers the dependency to be "hybrid" - If HttpContext is available, then the life-cycle of `TImplementation` is per-HttpContext, otherwise it's per-thread when `TService` is requested.
- `AddHttpContextScoped<TService, TImplementation>`: new instance of `TImplementation` will be created for every `HttpRequest`.
- `AddAddHttpContextOrThreadScoped`: this should be equivalent with AddScoped.

Remember to choose the life-cycle widely, especially when you class has states stored. Having wrong life-cycle can messed up with states and can cause nasty problems, especially in concurrent cases.

The second way to register dependencies is to use `ServiceConfigurationAttribute`, for example:

```
[ServiceConfiguration(typeof(ApplicationContext), Lifecycle = ServiceInstanceSco\
pe.Singleton)]
[ServiceConfiguration(typeof(IApplicationContext), FactoryMember = "Instance")] \
// instance (otherwise the container will keep one singleton for each interface/\
type)
public class ApplicationContext : IApplicationContext
{
}
```

Here the list of scope you can choose from (There are other scopes, such as `HybridHttpSession`, however they will be obsolete soon):

```
public enum ServiceInstanceScope
{
    /// <summary>
    /// New instance per usage. Equivalent to AddTransient.
    /// </summary>
    Transient,

    /// <summary>
    /// One instance per container. Equivalent to AddSingleton.
    /// </summary>
    Singleton,

    /// <summary>
    /// New instance per request. Equivalent to Add
    /// </summary>
    HttpContext,

    /// <summary>
    /// New instance per thread or request. Equivalent to AddScoped.
    /// </summary>
    Hybrid,
}
```

Normally, you should use `ServiceConfigurationAttribute` to register your dependencies, it's more convenient way. However, there are cases when you need to use `IConfigurableModule` instead: to control the order of dependencies registration. If you want to override a certain implementation of a system interface, and you know it is registered in `CommerceInitialization`, for example - you can make your `IConfigurableModule` depends on that module, therefore your dependency registration will be guaranteed to run *after* the system one. In the world of DI, the later registration wins.

There are two notable `IConfigurableModule` you should know about Episerver Commerce:

- `CommerceInitialization` in Mediachase.Commerce namespace. This can be considered as the "main" module where almost important stuffs from Commerce are registered.
- `InitializationModule` in EPiServer.Commerce.Initialization. This is where content and routing stuffs are registered.

Unless you have strong reasons to have dependency on `CommerceInitialization`, it's usually safer to make your `IConfigurableModule` depends on `InitializationModule` to make sure every Commerce registrations have been done. It's is important to note that there is no guarantee about the order of execution for two modules which have dependency on same other module. So if your module depends on `CommerceInitialization`, it's undetermined that it will run before or after `InitializationModule`.

OK that's enough of theoretical information. Let's get our hands dirty with some code.

`ICurrentMarket` is a small interface with only two methods: to get and to set the current market. The class `MarketStorage` in Enoteca and Commerce Sample site simply allows you set current market (selected from a dropdown) to store in cookie, and load it back. In real world, you might not want customers to do that, the market must be automatically assigned from customers GeoIP (customers might be able to select the market afterward, but the market is assigned to them for the first time they visit)

The unusual ways.

As mentioned in previous chapters, Mediachase eCF has its own ways to extend the framework. It's not as unified or easy to understand as the way Episerver Framework provides, but it still does its job when you needs it. We will discuss case by case to see how you can extend the framework in each way.

Customize order processing

Until the new order system is in place (And you are willing to make the move), it might seem impossible to customize how orders (including cart, payment plan and purchase order) are handled. Not quite. In facts, it's quite easy to do so, with the support from the order configuration system.

Let's assume you want to do extra thing whenever you save a purchase order. You might want to call an external API:s to update some information, for example. How?

Let's first create your custom class:

```
public class MyPurchaseOrder : PurchaseOrder
{
    public override void AcceptChanges()
    {
        //Your fancy code goes here.
        base.AcceptChanges();
    }
}
```

You can even remove `base.AcceptChanges()`. That means you don't want to save the order into database in normal way. You might save it to your custom persistent backend, a document database might be. Note that if you goes that way, the static methods in PurchaseOrder class to search and load for orders, such as `LoadByCustomer(Guid CustomerId)` will no longer works and some features depend on it, like the OrderManagement in Commerce Manager, will break (unless you go even further and extend them as well). You're on your very own now.

And now the magic parts: To let the system know that you want to use MyPurchaseOrder instead of PurchaseOrder. Just go to Configs\ecf.order.config, and update the line in MappedTypes/PurchaseOrderType with your new type, note that it must be the fullname of the type:

```
<PurchaseOrderType name="MySite.Order.Classes.MyPurchaseOrder, MySite" />
```

Now build it and you're done. Try to place an order and see the magic happens.

The above step applies for front-end site. If you want to use it in Commerce Manager as well, make sure to deploy your assembly to its bin folder, and update the its config file as well.

This method also works with other metaclasses, such as Cart or PaymentPlan, and even OrderForm and LineItem. In the end, you have the power all over the order system!

Work with "workflows".

No I lied, at least partly. It's not the workflows anymore. Episerver Commerce has a special, open-sourced project (Mediachase.Commerce.Workflow) [51], which was based on Windows Workflow Foundation (WF) 3.5 in Commerce 8 and earlier. In Commerce9, the engine has been rewritten to remove the dependencies on WF, and the "workflow" has now been renamed to ActivityFlow, to indicate that it now longer depends on WF. Still, the term workflow is more familiar and we will use that term in this book.

The reason Episerver Commerce open-sources Mediachase.Commerce.Workflow is to allow you to customize the workflows to fit with your business requirements. The choice is yours - either attach the project into your solution and modify it directly, or inject your "activity" into the existing workflows replace the default one.[52]

How workflows are registered

If you worked with Windows Workflow Foundation before, you'll probably know about XOML (Extensible Object Markup Language) files. They were used to define the workflows in Commerce 8 and prior, and here's how an XOML would look like

[51] Can be downloaded from here

[52] It was a known limitation with Commerce 8 and prior that if you reference to the workflow assembly, you will have to stick with EPiServer.Framework package which your EPiServer.Commerce.Core package was built with . This was caused by the build engine of WF as it skips the assembly binding configured in web.config. Such limitation was removed in Commerce 9. One more reason to upgrade your solution to the latest Commerce version and stay on top of your game?

```
<SequentialWorkflowActivity x:Class="Mediachase.Commerce.Workflow.CartCheckoutWo\
rkflow" x:Name="CartCheckoutWorkflow" xmlns:ns0="clr-namespace:Mediachase.Commer\
ce.Workflow.Activities" xmlns:ns1="clr-namespace:Mediachase.Commerce.Workflow.Ac\
tivities.Cart" xmlns:ns2="clr-namespace:Mediachase.Commerce.Workflow.Activities.\
PurchaseOrderActivities" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" \
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
<IfElseActivity x:Name="ifElseActivity1">
        <IfElseBranchActivity x:Name="ifElseBranchActivity1">
                <IfElseBranchActivity.Condition>
                        <CodeCondition Condition="RunProcessPayment" />
                </IfElseBranchActivity.Condition>
                <ns1:ProcessPaymentActivity Warnings="{x:Null}" OrderGroup="{ActivityBind Cart\
CheckoutWorkflow,Path=OrderGroup}" Payment="{x:Null}" x:Name="processPaymentActi\
vity1" />
        </IfElseBranchActivity>
</IfElseActivity>
<ns0:CalculateTotalsActivity x:Name="calculateTotalsActivity1" OrderGroup="{Acti\
vityBind CartCheckoutWorkflow,Path=OrderGroup}" Warnings="{x:Null}" />
<ns1:AdjustInventoryActivity Description="Adjusts Inventory." x:Name="adjustInve\
ntoryActivity1" OrderGroup="{ActivityBind CartCheckoutWorkflow,Path=OrderGroup}"\
 Warnings="{ActivityBind CartCheckoutWorkflow,Path=Warnings}" />
<ns1:RecordPromotionUsageActivity UsageStatus="Used" Warnings="{x:Null}" OrderGr\
oup="{ActivityBind CartCheckoutWorkflow,Path=OrderGroup}" x:Name="recordPromotio\
nUsageActivity1" />
</SequentialWorkflowActivity>
```

You might guess - this is the `CartCheckout` workflow, and it has a quite simple logic. If the conditon `RunProcessPayment` is true, then run `ProcessPaymentActivity`. Then run `CalculateTotalsActivity` and `AdjustInventoryActivity`. And finally `RecordPromotionUsageActivity`.

Not too complicated, right? But there are several major drawbacks:

- It's overly complicated for a simple logic. If it was written in C#, it would be on 4-5 lines of code.
- It's hard to know an activity belongs to which workflows. You can only search, instead of using the Go to implementation or Find all references provided by Visual Studio.
- It's hard to edit and even worse, hard to debug. The workflows had always puzzled me when it came to breakpoints and so.

Luckiky for us (myself included), Commerce 9 removed the dependencies of Workflow Foundation, and moved to a new, almost "POCO" (Plain Old C# Object) approach. Here's how the above activity looks like now:

```
using Mediachase.Commerce.Engine;
using Mediachase.Commerce.Orders.Managers;
using Mediachase.Commerce.Workflow.Activities;
using Mediachase.Commerce.Workflow.Activities.Cart;

namespace Mediachase.Commerce.Workflow
{
    /// <summary>
    /// This class represents the Cart Checkout workflow
    /// </summary>
    [ActivityFlowConfiguration(Name = OrderGroupWorkflowManager.CartCheckOutWork\
flowName)]
    public class CartCheckoutActivityFlow : ActivityFlow
    {
        /// <inheritdoc />
        public override ActivityFlowRunner Configure(ActivityFlowRunner activity\
Flow)
        {
            return activityFlow.If(() => ShouldProcessPayment())
                                .Do<ProcessPaymentActivity>()
                            .EndIf()
                            .Do<CalculateTotalsActivity>()
                            .Do<AdjustInventoryActivity>()
                            .Do<RecordPromotionUsageActivity>();
        }
    }
}
```

Tell me how long did it take you to understand what it intends to do? I even included the using statements and the doc comments and it's still shorter than the XOML version. I can now see where an activity is used easily by features provided by Visual Studio, and you can debug without sweating and grinning. Life just got easier!

OK, so let's break down how the new "workflows" are registered.

At first, it's supposed to be inherited from `ActivityFlow` (as we mentioned earlier, it's the way Episerver used to distinct this from Workflow term - so we know it's the new engine). By convention, the class will have the suffix -`ActivityFlow`, you are free to choose otherwise, but it's easier to stick with the pattern.

The class must be decorated with an `ActivityFlowConfiguration` attribute. It has two important properties:

- `Name`, which is mandatory. In this example, the name of the workflow is "CartCheckout", which is supposed to be a system workflow.

- `Priority`, which is optional. In case there are multiple wokflows with a same name, the workflow with highest priority will take effect. By default, priority = 0, then any priority higher than take will take effect.

So if you want to have your own CartCheckout but don't want to mess with the default one, it's simple as this:

```
    /// <summary>
    /// This class represents the Cart Checkout workflow
    /// </summary>
    [ActivityFlowConfiguration(Name = OrderGroupWorkflowManager.CartCheckOutWork\
flowName, Priority=10)]
    public class MyCartCheckoutActivityFlow : ActivityFlow
    {
    }
```

And finally, a method to configure how the workflow acts:

```
public override ActivityFlowRunner Configure(ActivityFlowRunner activityFlow)
{
}
```

You can use a fluent API:s on activityFlow to configure the condition an action, with three basic elements: `If(Func<bool>)`, `EndIf()` and `Do<T>()`. `If` will be the conditional branching, while `Do<T>` will execute an activity.

There is another attribute for `ActivityFlowConfiguration` you might need to know about - `AvailableInBetaMode`. Only workflows decorated with this attribute are available in BETA mode (VNextFeatures enabled). However, at the time this book completes, I would expect this attribute to be removed (along with all BETA tags in the API:s)

If you have customer workflows from Commerce 8, make sure to check out the workflow migration tool[53]. It's pretty cool and will convert your workflow project into new format in (almost) just one click!

Let's go through an example on how you can customize the workflow project. Normally, when a customer adds items to the cart, the system will have to check the inventory to see if there are enough goods in the store, and then when he or she actually places an order, the system will have to adjust the quantities. However, this is not necessary if your website only sells downloadable games. You can remove those steps altogether for simpler processing.

There are three "Activity" which relate to such actions:

[53] http://world.episerver.com/download/Items/EPiServer-Commerce/episerver-commerce-9-tools-and-code-samples/

- `GetFulfillmentWarehouseActivity` to get the warehouse which can fulfill the order.
- `CheckInstoreInventoryActivity` or `CheckInventoryActivity` to check if the inventory can fulfill the order.
- `AdjustInventoryActivity` to actually adjust the inventory according to purchasing number.

Also, `ActivityFlow` has a property named `ShouldAdjustInventory`, to indicate if the activity flow should adjust the inventory. Because in this case it's always false, to we will remove it as well.

We can start by removing above activities form the project. Because some `ActivityFlows` use them, they need to be updated as well. For example, here's how `PORecalculateActivityFlow` looks like before

```
public override ActivityFlowRunner Configure(ActivityFlowRunner activityFlow)
{
    return activityFlow.Do<ValidateLineItemsActivity>()
                    .Do<GetFulfillmentWarehouseActivity>()
                    .If(() => ShouldAdjustInventory())
                        .If(() => ShouldCheckInstoreInventory())
                            .Do<CheckInstoreInventoryActivity>()
                        .Else()
                            .Do<CheckInventoryActivity>()
                        .EndIf()
                    .EndIf()
                    .If(() => ShouldAdjustInventory())
                        .Do<AdjustInventoryActivity>()
                    .EndIf()
                    .If(() => ShouldRecalculateOrder())
                        .Do<ProcessShipmentsActivity>()
                        .Do<RemoveDiscountsActivity>()
                        .Do<CalculateTotalsActivity>()
                        .Do<CalculateDiscountsActivity>()
                        .Do<CalculateTotalsActivity>()
                        .Do<CalculateTaxActivity>()
                        .Do<CalculateTotalsActivity>()
                    .EndIf()
                    .Do<CalculatePurchaseOrderStatusActivity>();
}
```

And after:

```
    public override ActivityFlowRunner Configure(ActivityFlowRunner activityFlow)
    {
        return activityFlow.Do<ValidateLineItemsActivity>()
                        .If(() => ShouldRecalculateOrder())
                            .Do<ProcessShipmentsActivity>()
                            .Do<RemoveDiscountsActivity>()
                            .Do<CalculateTotalsActivity>()
                            .Do<CalculateDiscountsActivity>()
                            .Do<CalculateTotalsActivity>()
                            .Do<CalculateTaxActivity>()
                            .Do<CalculateTotalsActivity>()
                        .EndIf()
                        .Do<CalculatePurchaseOrderStatusActivity>();
    }
```

If you are using the new promotion system - aka having `VNextFeatures` flag enabled, it is important to enable `AvailableInBetaMode` when you register your `ActivityFlow`:

```
    [ActivityFlowConfiguration(Name = "CartCheckout", Priority=10, AvailableInBe\
taMode = true)]
    public class MyCartCheckoutActivityFlow : ActivityFlow
    {
    }
```

Otherwise, your customized `ActivityFlow` will not be available and the default one will be used.

Workflows optional parameters

As we can see above, you can use If-Else statements to control workflows. Instead of having two or more worflows, you can have only one, and when you call it, depending on the conditions, you might want the flow to act differently. This can be done by passing one or more optional parameters to the workflow.

To define a parameter in a workflow, simply add a property with a special attribute to it:

```
    [ActivityFlowContextProperty]
    public virtual bool SkipRecalculateOrder { get; set; }

    public bool ShouldRecalculateOrder()
    {
        return !SkipRecalculateOrder;
    }
```

By default, this property has value of `false`, therefore `ShouldRecalculateOrder` return true and the acitivities will be executed. If we want to skip recalculating order, we need to set the flag to true, by sending a `Dictionary<string, object>` to the workflow:

```
    var param = new Dictionary<string, object>();
    param.Add("SkipRecalculateOrder", true);
    OrderGroupWorkflowManager.RunWorkflow(purchaseOrder, OrderGroupWorkflowManag\
er.PORecalculateWorkflowName, false, param);
```

If your workflow has more than one flag, simply add additional parameters to the dictionary `param` before calling the workflow.

The legacy continues

Coming from a long history even before the acquisition, Commerce Manager can be considered as the legacy part of Episerver Commerce. It might not have the shiny, responsive design with widgets like the new Catalog UI, but it does it job quite well. Some of its users even liked it, for it efficiency. Truth be told, Commerce Manager has not been actively developed. Important bugs are fixed, but no new features will be added. With the dawn of ASP.NET core (Formerly ASP.NET 5), which does not support WebForms, the foreseen future of Commerce Manager will be clinging on life-support, and eventually phrased out. It has had it days.

But that will not happen anytime soon. Commerce Manager will continue to run, even silently in hundreds of implementations of Episerver Commerce. It's not dead website. And its architecture, which might be confusing at first, allow you to extend the functionalities when you need. It might not be the most customizable, or powerful framework out there, but it does it job.

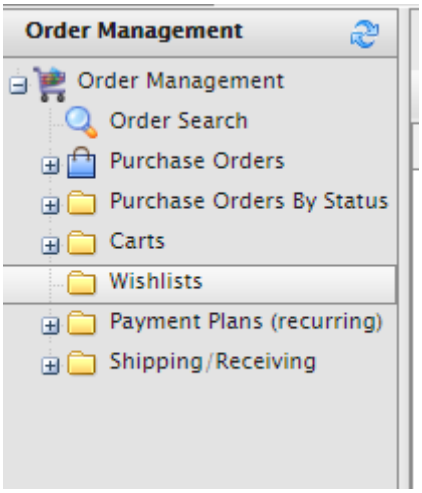
Let's us look in a specific case as I get from a question in Episerver World forums: How to add the wish-list feature to the Order management system of Commerce Manager. We will go through the steps needed and also examine the structure of Commerce Manager.

At first, we have to add a new menu to the navigation tree of Commerce Manager. Commerce Manager define that tree by multiple parts, each part represents one of the main features of the system (Customer, Catalog, Order and so on). Because we're looking to add wish-list menu, it should be in the Order subsystem. We then should edit `Apps\Order\Config\View\Navigation\LeftMenu.xml`.

Don't let its content scare you. It can be confusing at first, but in the end, you only need to replicate the Carts menu. This can be added right after Order_Carts Link in `<Navigation>/<Tabs id="mainTag">/<add>` section:

```
<Link id="Order_Wishlist" text="Wishlists" order="45" iconUrl="" iconCss="treeIc\
on" type="Carts" command="cmdWishlists"
       permissions="order:mng:view"/>
```

And magic happens, we added a menu to the left tree:



Wishlists, woohoo

Wait a second, we don't have the extra submenus. That the extra benefit from TreeSource.aspx. We might add those later, but let's keep it simple for now.

The menu does not really do anything now - we need to add a command for it: <Command id="cmdWishlists"> <CommandType>ClientAction</CommandType> <ClientScript>CSManagementClient.Chang ‘Wishlist-List’, ‘’)</ClientScript> <EnableHandler type="Mediachase.Commerce.Manager.CommandHandlers.Perm Mediachase.ConsoleManager" /> </Command> Node that here, we set the EnableHandler to be `PermissionEnableHandler`. That mean, Commerce Manager will only allow users which granted `order:mng:view` permission to access the view - same as the menu for ShoppingCart does.

So now it's actually doing something: When you click on it, it will try to load the `Wishlist-List.xml` view (in same folder as `Orders-List.xml`). So let's go to `Apps\Order\Config\View\Grids` and add a file there. We, again, want to keep it simple, therefore we will just copy the Orders-List.xml file, except the last section needs to be changed:

```
<ViewConfig>
        <setAttributes id="Wishlist-List" isNameDynamic="true" controlUrl="Order/Wishli\
st.ascx" permissions="order:mng:view" help="Order+Management"></setAttributes>
</ViewConfig>
```

Note the `Wishlist.ascx` part - it's something we have to implement ourselves. Open your Visual Studio and create an empty webform project. You'll need to add reference to Mediachase.WebConsoleLib and Mediachase.CommerceManager, then add a new user control like this:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Wishlist.ascx.cs" I\
nherits="EPiServer.Commerce.Sample.Wishlist" %>
<%@ Register Src="~/Apps/Core/Controls/EcfListViewControl.ascx"
    TagName="EcfListViewControl" TagPrefix="core" %>
<%@ Register TagPrefix="orders" Namespace="Mediachase.Commerce.Orders.DataSource\
s"
    Assembly="Mediachase.Commerce" %>
<orders:OrderDataSource runat="server" ID="OrderListDataSource"></orders:OrderDa\
taSource>
<core:EcfListViewControl id="MyListView" runat="server" DataSourceID="OrderListD\
ataSource" AppId="Order"
    ViewId="Wishlist-List" ShowTopToolbar="true"></core:EcfListViewControl>
```

And the designer.cs file should look like:

```
namespace EPiServer.Commerce.Sample {
    public partial class Wishlist {
        protected global::Mediachase.Commerce.Orders.DataSources.OrderDataSource\
 OrderListDataSource;

        protected global::Mediachase.Commerce.Manager.Core.Controls.EcfListViewC\
ontrol MyListView;
    }
}
```

And now we can try to implement the actual code. First thing - we need to initialize the datasource, in this case, the list of wishlist:

```
//We need to inherit from `OrderBaseUserControl`
public partial class Wishlist : OrderBaseUserControl
{
    private void InitDataSource(int startRowIndex, int recordsCount, bool return\
TotalCount, string orderByClause)
    {
        if (String.IsNullOrEmpty(orderByClause))
        {
            orderByClause = String.Format("OrderGroupId DESC");
        }

        OrderSearchOptions options = new OrderSearchOptions();
        OrderListDataSource.Options.Namespace = "Mediachase.Commerce.Orders";
        OrderListDataSource.Options.Classes.Add("ShoppingCart");
```

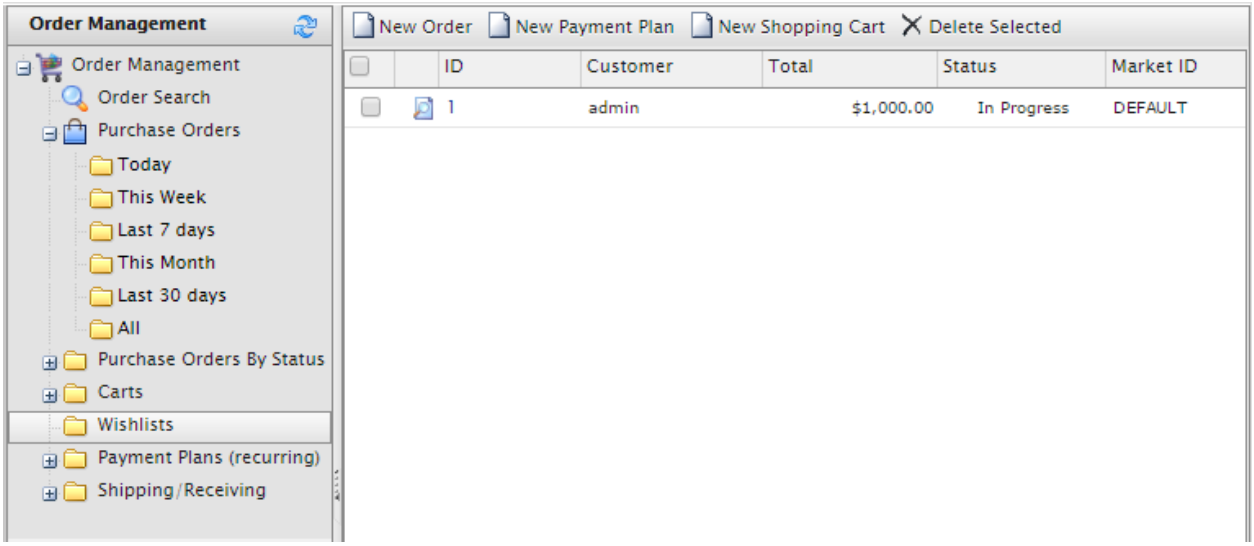
```
        OrderListDataSource.Parameters.SqlWhereClause = " Name = 'Wishlist'";
        MyListView.DataMember = OrderDataSource.OrderDataSourceView.CartsViewNam\
e;

        OrderListDataSource.Options.RecordsToRetrieve = recordsCount;
        OrderListDataSource.Options.StartingRecord = startRowIndex;
        OrderListDataSource.Parameters.OrderByClause = orderByClause;
    }
}
```

The code should explain itself. We are loading the shopping carts which has the name of 'Wishlist'. It's only the default name for the wishlist in the system, and if you used another name, just change it. By default, we will sort it by the Id, but the EcfListViewControl allows sorting by column.

The final piece is to actually bind the data and a bit of magic:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack || String.Compare(Request.Form["__EVENTTARGET"], CommandMana\
ger.GetCurrent(this.Page).ID, false) == 0)
    {
        if (!IsPostBack)
        {
            MyListView.CurrentListView.PrimaryKeyId = EcfListView.MakePrimaryKey\
IdString("OrderGroupId", "CustomerId");
        }

        InitDataSource(0, int.MaxValue, true, "");
        DataBind();
    }
}
```

Build the project and drop the assembly to the bin folder of Commerce Manager. And of course, `Wishlist.ascx` to AppsOrder folder. And refresh your browser.



Woo hoo, finally the wish list

It's not that hard, right? Extending Commerce Manager means dealing with legacy code, WebForms, ViewState, IsPostBack. The old glorios days - but I hope you don't have to deal with too often.

We barely scratched the surface of Commerce Manager - and there are other stuffs can do? You still want to go?

Extending Commerce Manager has one caveat you need to know: Everytime you update your Commerce Manager website, your changes are likely to be overwritten by the files from Episerver. Keep them in separated folder and then apply the changes when you upgrade.

Access rights and Permissions.

Security is a big subject and I highly recommend you to not take it lightly. However, this book is not about security, and I am, sadly, not a security expert. If you are looking to secure your site, follow the best practices, and consult an expert on the topics.

One of the most frequently asked question on Episerver World Forums is if the catalog contents support access rights the way CMS contents do. The answer is sadly, no. There are only two level of access rights for catalog contents: All or nothing. Either you have access to entire catalog system, or you don't (Strictly speaking, "all" means the access rights affect the whole catalog. There are still levels of access rights, not only one). There are plans to make the access rights more sophisticated, but it'll not happen in near future. So we'll have to live with it, at least for Commerce 9.

However, there are still available settings for access rights and permissions in Commerce Manager.

In Commerce 8.15 and earlier, the access rights and permissions settings (aka Role Management) can be handled entirely in Commerce Manager. However, from Commerce 8.16, the roles have been moved entirely to CMS. If you go to CMS Admin/Config/Security/Permission for Functions, you can see a list of permissions which can be set to Commerce Manager's functionalities. These permissions can be granted to certain roles

By default, users in Administrators and WebAdmins group can edit catalog items

So let's say we want to add permission to the new wishlist view we just added in previous section (Normally it should share the same mission as the order view, but we are taking it as example). We need to add a permission type for it, and then register that permission type for the wishlist view.

First, let's change the permission for wishlist view from `order:mng:view` to `wishlist:mng:view`

```
<Column width="80" allowSorting="true" dataField="OrderGroupId"  headingText="{S\
haredStrings:ID}" columnType="CustomTemplate" permissions="wishlist:mng:view">
    <Template id="LinkTemplate" controlUrl="GridTemplates/OrderLinkTemplate.ascx\
"></Template>
</Column>
```

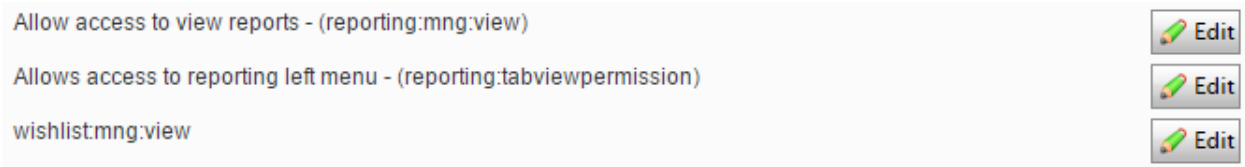
We need to define this new `PermissionType`:

```
PermissionType wishlistView = new PermissionType(Permissions.GroupName, "wishlis\
t:mng:view");
```

`Mediachase.Commerce.Permissions.GroupName` is a constant defined by Commerce - we'll use it so our new permission type will be grouped with other one. Of course, you can define another group name for custom permission types. Now the final step is to register this `PermissionType` into system - via `PermissionTypeRepository.Register(PermissionType)`. `PermissionTypeRepository.Register` will return true if the `PermissionType` was successfully registered, and false if it is already existed - so we can put the code either in an `IInitializationModule` or a `MigrationStep`:

```
var permissionTypeRepo = ServiceLocator.Current.GetInstance<PermissionTypeReposi\
tory>();
permissionTypeRepo.Register(wishlistView);
```

And that's it. Our new permission type is now accessible via Admin UI:



Our new permission type is now available

It's now available to add to assign to users or groups to grant access to wishlist.

However, you might want to have a nice description of the permission type - then you're lucky. Those can be localized - just add this to your language file:

```
<?xml version="1.0" encoding="utf-8" ?>
<languages>
<language name="English" id="en">
    <admin>
    <permissiontype>
        <groups>
        <EPiCommerce>
            <permissions>
            <wishlistview>Allows access to view wishlist (wishlist:admin:view)</\
wishlistview>
            </permission>
        </EPiCommerce>
        </groups>
    </permissiontype>
    </admin>
</language>
</languages>
```

These are pretty much what you can do to the permission system in Commerce Manager. We of course hope newer versions of Commerce will provide more control over access rights and permissions.

Payment providers

As we promised in Order system part, we now discuss payment providers - the essential part of any Commerce sites.

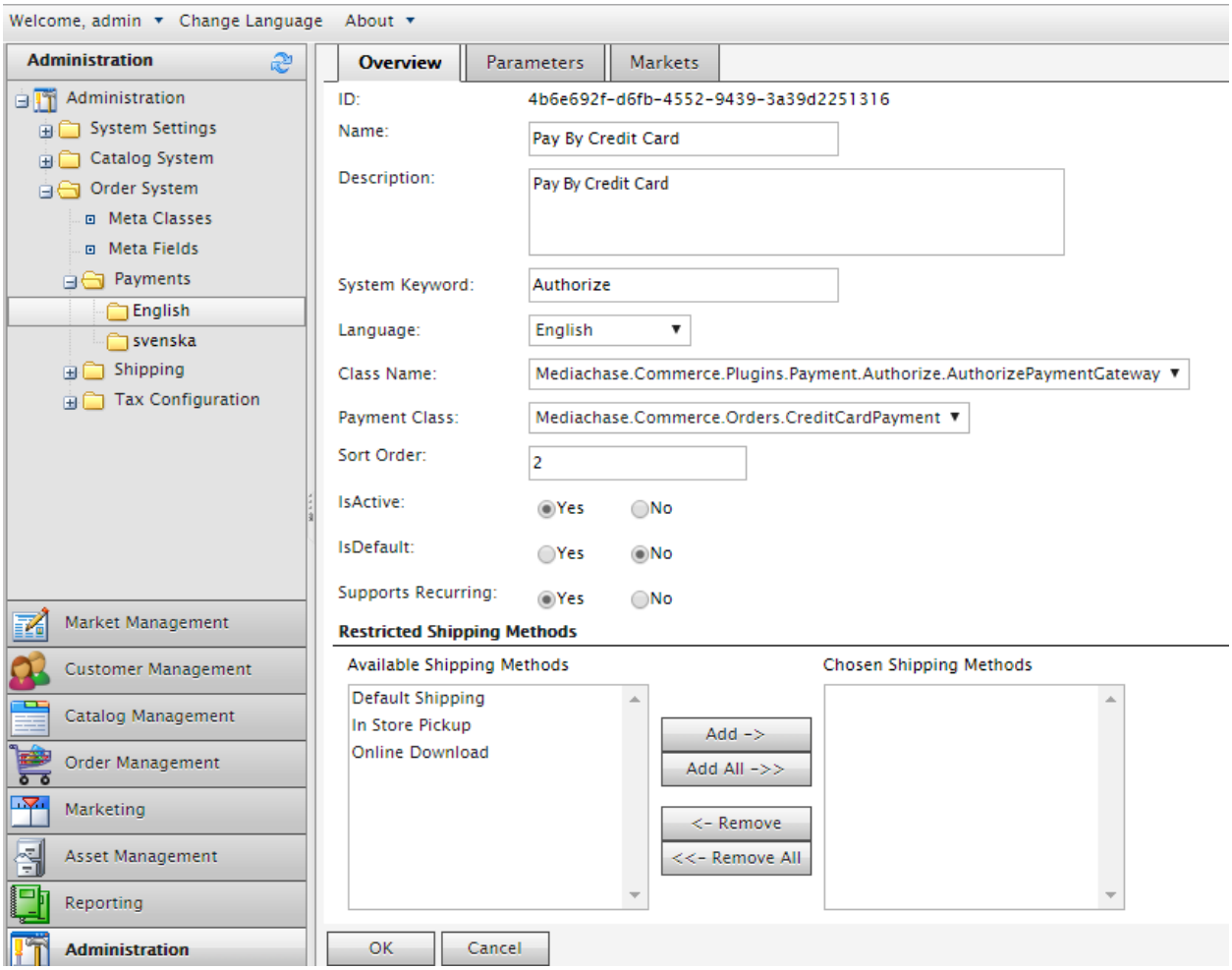
There are several ways to call the “payments” - they can be called either Payment providers, Payment methods or Payment gateways. In this section we will use term “Payment providers” or “Payment gateways” for the payments themselves. DIBS, Paypal, for example, are Payment providers. Once we configured them to work with Episerver Commerce, we have “payment methods”. In short, payment methods are configured payment providers. This is our terminology in this book, it might or might not be the most correct way in real world.

Episerver provides some payment providers as example, which you can download from here[54] (login required). The sample ones include:

- DIBS
- Paypal
- Datacash

In the past, there was also Klarna payment provider, but it was retired with Klarna released first-party Klarna payment provider addon for Episerver Commerce.

[54] http://world.episerver.com/download/Items/Episerver-Commerce/episerver-commerce-9-payment-providers/



A payment provider configuration in Commerce Manager

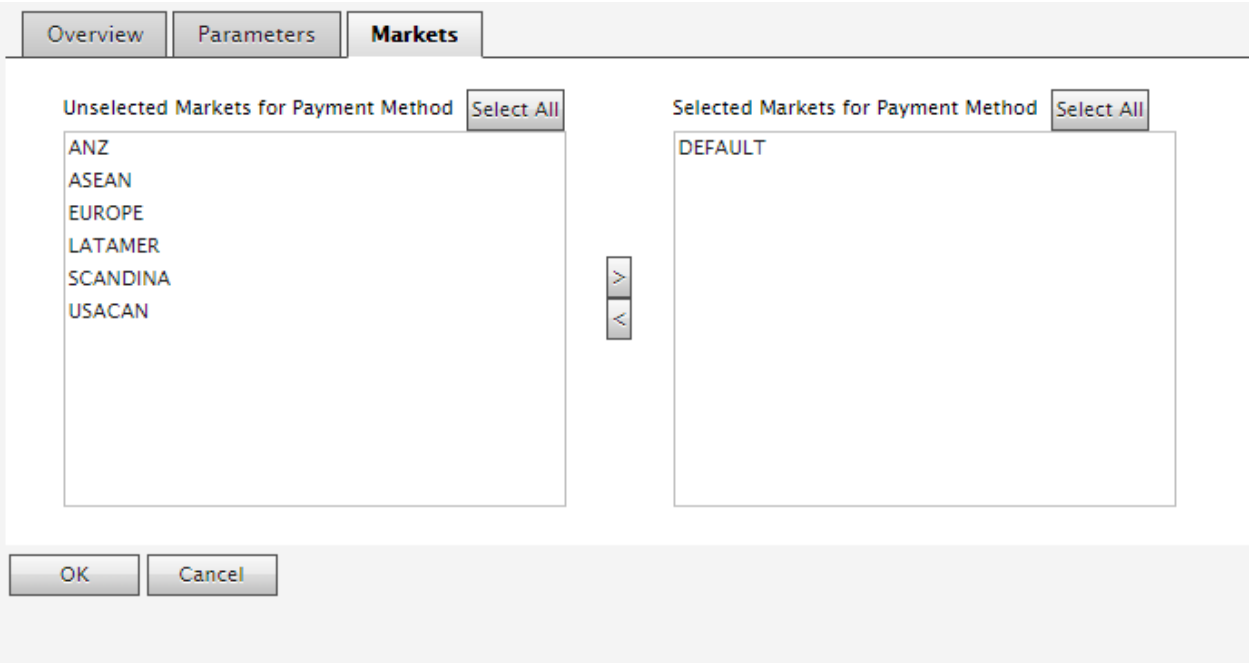
Configure payment methods.

A payment provider has plenty configuration - and most of them are rather important. Let's go through them one by one.

- The `Name` and `Description` are to show to end-customers.
- `System Keyword` once set, is readonly and can be changed. This is used by the system to identify your payment provider, so it must be unique. Commerce Manager will expect an user control (.ascx file) in `Apps\Order\Payments\Plugins\<System Keywork>` with name of `ConfigurePayment.ascx` - it will load that user control to allow configuring the payment provider.
- `Language` decides where the payment provider belongs to - as we mentioned before the payment providers are categorized by languages, not markets. So if you want to have a payment

provider for both “English” and “Svenska”, you’ll have to create two payment providers, with `System Keyword` as “MyPayment-En”, and “MyPayment-sv”, which are available in “English” and “Svenska”, respectively.

- `Class Name` - is the full name of your payment provider class - which implements `IPaymentGateway` interface.
- `Payment Class` - the class inherits from `Payment` which your payment provider class supposes to return. We will talk more about `Class Name` and `Payment Class` in next section.
- `Sort Order` - this can be used to determine the order of payment providers show to customers, for example, you can have “Credit Card”, “Cash On Delivery”, “Gift Card” - in that order at checkout page.
- `IsActive` shows if the payment is active, however it’s not used by other parts of the system. You can use this flag to hide a payment method in checkout page - but again it’s up to you to handle that.
- `IsDefault` and `Support Recurring` decides if it is default or if it supports recurring payments (to be used for `PaymentPlan`). Like `IsActive`, it is not really used by other parts of the system.
- A payment provider can be configured to be used only with certain shipping methods, or in certain market.



This payment provider is only available in `DEFAULT` market

IPaymentGateway

For a payment provider to be available for selection, it must implement `IPaymentGateway` interface, which is a quite simple interface:

```
public interface IPaymentGateway
{
    /// <summary>
    /// Returns the configuration data associated with a gateway.
    /// Sets the configuration gateway data. This data typically includes
    /// information like gateway URL, account info and so on.
    /// </summary>
    IDictionary<string, string> Settings { get; set;}

    /// <summary>
    /// Processes the payment. Can be used for both positive and negative transa\
ctions.
    /// </summary>
    bool ProcessPayment(Payment payment, ref string message);
}
```

`Settings` is simply an object for configuration variable, while `ProcessPayment` is an interesting method. It is called both for payment (when customers place order) and refund (when customers return items, for example). To determine which action to take, check `Payment.TransactionType` property. Here's the values available (Note that `Payment.TransactionType` is string type, so you will need to convert this enum to `string` for comparision):

```
public enum TransactionType
{
        /// <summary>
        /// Requests to charge a cardholder.
        /// An approved Authorization transaction reduces the cardholderâ€™s open-to-bu\
y amount (the amount that can currently be charged against the account), but doe\
s not actually capture the funds.
        /// </summary>
        Authorization = 1,
        /// <summary>
        /// Used to complete an Authorization transaction that was successfully authori\
zed through the payment gateway.
        /// Note: An Authorization and Capture together are considered one complete tra\
nsaction. Once the Capture is submitted, the transaction will be sent for settle\
ment.
        /// </summary>
        Capture = 2,
        /// <summary>
        /// Charges the specified amount against the account, and marks the transaction\
 for immediate funds transfer (capture) during the next settlement period.
```

```
        /// </summary>
        Sale = 4,
        /// <summary>
        /// Used to cancel an original transaction that is not yet settled and prevents\
 it from being sent for settlement.
        /// </summary>
        Void = 8,
        /// <summary>
        /// Returns the specified amount (posts a credit) to the cardholderâ€™s account.
        /// </summary>
        Credit = 16,
        /// <summary>
        /// Other transaction type.
        /// </summary>
        Other = 32,
    /// <summary>
    /// Used to capture the remaining balance.
    /// </summary>
    /// <remarks>
    /// In some payment provider, only a single Capture transaction can be submi\
tted against an Authorization, even if we do not capture the full amount with ou\
r Capture.
    /// If there is an available amount remaining, we may use the Capture Only t\
ransaction type to capture the remaining balance.
    /// </remarks>
    CaptureOnly = 64
}
```

`ProcessPayment` takes a `Payment` instance - with the certain type as you configred the payment provider. Here's the payment types available out-of-box:

- `ExchangePayment`
- `InvoicePayment`
- `OtherPayment`
- `GiftCardPayment`
- `CashCardPayment`
- `CreditCardPayment`

Of course, you can create your customized types of payment, they just need to implements `Payment` abstract class.

Commerce uses reflection to load the payment providers (which are supposed to implement `IPaymentGateway` interface). It's sometimes a bug when there are no selection available in the `Class Name` dropdownlist - that means there are something wrong with your Commerce Manager bin folder. A dependent assembly might be missing or has wrong version.

Manage payment methods programmatically.

Once a payment provider is configured, it is stored in the database. Then when the customer checks out, you can show all the available payment providers. While configuring payment providers is mostly done manually, getting payment methods is done by code.

Episerver Commerce provides `PaymentManager` to do such tasks. It provides plenty of methods to get a `PaymentMethodDto` - which can be filtered by language, market, with active or inactive payment methods.

How payments work

In Part 3 we talked about how to add payment(s) into an `OrderGroup` (or `IOrderGroup`). But how does it work internally?

When the customer checks out an order, our payment module - which implements `IPaymentOption` - will be called. `IPaymentOption.PreProcess(OrderForm form)` is expected to return a `Payment` instance - which can be one of the builtin types (`OtherPayment`, `CreditCardPayment`, etc.), or your own implementation. Here, your payment provider should set the correct `TransactionType` for the `Payment` - it can be `Sale`, `Authorization` or `Transaction`.

We then set the required information to this `Payment`, and add it to `OrderForm.Payments`:

```
payment.PaymentMethodId = paymentMethodRow.PaymentMethodId;
payment.PaymentMethodName = paymentMethodRow.Name;
payment.BillingAddressId = orderForm.BillingAddressId;
payment.Amount = orderForm.Total;
orderForm.Payments.Add(payment);
```

Then we need to process the payments - either via workflow (By `CartCheckout` workflow, which will call `ProcessPaymentActivity`, or by `DefaultPaymentProcessor`). The workflow/processor will try to create an `PaymentGateway` from `Payment.PaymentMethoidId`, and then call `PaymentGateway.ProcessPayment(Payment, ref string)`.

`PaymentGateway.ProcessPayment` is what your payment provider needs to implement. Depends of the way your payment provider works, it's important to check for `Payment.TransactionType` for relevant action. For example:

```
public override bool ProcessPayment(Mediachase.Commerce.Orders.Payment payment, \
ref string message)
    {
        if (payment.Parent.Parent is PurchaseOrder)
        {
            if (payment.TransactionType == TransactionType.Capture.ToString())
            {
                //Capture the amount
            }

            if (payment.TransactionType == TransactionType.Credit.ToString())
            {
                //Refund the amount
            }
            message = "TransactonType is not valid";
            return false;
        }
    }
```

It's worth noting that Episerver Commerce also supports split payments - if you payment method implements `ISplitPaymentGateway` instead of `IPaymentGateway`:

```
public interface ISplitPaymentGateway : IPaymentGateway
{
    bool ProcessPayment(Payment payment, Shipment shipment, ref string message);
}
```

This method will be called when you process the payment. This allows your customer to pay for each `Shipment`, instead of the whole order at once.

ProfileModule

Several features of Episerver frameworks in general and Episerver Commerce in particular - are provided by `IHttpModule` implementations. When you need to register an action as soon as the site starts - this is probably the best place to be. Here are the default modules in a standard Commerce installation:

```
<modules runAllManagedModulesForAllRequests="true">
  <add name="BusinessFoundationInitializeModule" preCondition="managedHandler" t\
ype="Mediachase.Commerce.Core.Modules.BusinessFoundationInitializeModule, Mediac\
hase.Commerce" />
  <add name="InitializationModule" type="EPiServer.Framework.Initialization.Init\
ializationModule, EPiServer.Framework" preCondition="managedHandler" />
  <add name="UrlRewriteModule" type="EPiServer.Web.RoutingUrlRewriteModule, EPiS\
erver" preCondition="managedHandler" />
  <add name="ShellRoutingModule" type="EPiServer.Shell.Web.Routing.ShellRoutingM\
odule, EPiServer.Shell" />
  <remove name="WebDAVModule" />
  <add name="ProfileModule" type="EPiServer.Business.Commerce.HttpModules.Profil\
eModule, EPiServer.Business.Commerce" />
  <add name="CatalogNodeModule" type="EPiServer.Business.Commerce.HttpModules.Ca\
talogNodeModule, EPiServer.Business.Commerce" />
</modules>
```

> You can ignore `CatalogNodeModule` because it's obsoleted and does not really do anything (future release of Commerce will get rid of that). And you should not touch other modules because they provide critial functions in Episerver Framework and Commerce. The (only) one we can change here is the `ProfileModule`.

So what we can do with `ProfileModule`? `ProfileModule` registers its methods to `AuthenticateRequest` event, so it will kick in when a customer logs in. It will try to assign the last order (placed when anonymous) to the current user profile, and merge carts (if not empty) to the existing carts (there might be multiple carts in a multi markets set up, and wishlist is also counted as one), if the carts are not empty. However, that might not be the behaviors we want, and we're going to replace that.

First, we need to implement `IHttpModule`. `IHttpModule` is just a simple ASP.NET interface:

```
public interface IHttpModule
{
    // Summary:
    //     Disposes of the resources (other than memory) used by the module that\
 implements
    //     System.Web.IHttpModule.
    void Dispose();

    // Summary:
    //     Initializes a module and prepares it to handle requests.
    void Init(HttpApplication context);
}
```

`Init(HttpApplication context)` is the place where we want to register our customer implementation, while `Dispose` can be used for cleaning up garbage - such as deregister the method from the event, to avoid any possible memory leaks.

```
    public void Init(HttpApplication context)
    {
        if (context == null)
        {
            return;
        }
        context.AuthenticateRequest += Application_AuthenticateRequest;
    }
```

Now you need to implement your `Application_AuthenticateRequest` method, and then simply replace the registration of `ProfileModule` in `<modules>` section with your type - the `type` value should be the full name of your type.

Technically it's possible to have both the default and your custom "profile" module. Just keep in mind that the module higher in the list will be processed first, if both of them register to same event. If your custom module also listens to `AuthenticateRequest` event, think if it should execute before or after `ProfileModule`.

CustomerContact events

You might notice the lacking of events in some parts of the system. We have events for catalog system, for order system, for prices and inventories changes, but that's not enough. You might want to have events - or at least - the ability to know when something happens. For example, when a customer contact is changed, or edited, or deleted, it would be very nice to do some extra actions. Sending emails, updating external systems, etc.

Such events are not available out-of-box, so we have to implement our own. How? We don't have `ICustomerContactService` (or something similiar) interface where we can write our implementation to replace the default service (and even if there is, it would be a big task to do so). So there's no "ordinary", framework-way to do that.

However, `CustomerContact` is built on Business Foundation system, and BF, at its core, is all above extensible and pluggable. We don't have `ICustomerContactService` interface, but we have `IPlugin` who can do the same, and even more.

As we learned in previous chapter, `CustomerContact` is just another `EntityObject` and all operations are still done via `BusinessManager.Execute(Request)` - even we have some nice wrapper methods to make working with it easier. And when `Execute(Request)` is called, it also runs all *registered* `IPlugin` modules. That's it, we can simply write an `IPlugin` implementation to inject our code when some changes are done to a customer contact.

```
public class CustomerContactHandler : IPlugin
{
    public void Execute(BusinessContext context)
    {
        var request = context.Request;
        if (request.Target.MetaClassName != "Contact")
        {
            return;
        }
        if (request.Method == RequestMethod.Delete)
        {
            //Do something
        }
    }
}
```

`IPlugin` has only one method `Execute(BusinessContext)` - which we implement here. We care about `context.Request`: if the target object being "handled" is of actual `CustomerContact` type -

otherwise we just skip it to avoid any performance overload, because our module will be called for every Business Object of any types. Here's we are "listening" to `Delete` action, but you're free to listen to other actions, here's the full list:

```
public const string InitializeEntity = "InitializeEntity";

public const string Create = "Create";
public const string Update = "Update";
public const string Delete = "Delete";
public const string Load = "Load";
public const string List = "List";

public const string Import = "Import";
public const string InitializeMappingDocument = "InitializeMappingDocument";
```

Normally, the CRUD operations are the ones we care about. Now build your project contains the class above and drop the assembly into bin folder of Commerce Manager, then try to delete a contact.

Wait a minute. Nothing happens.

Well, yes, because we haven't *registered* it. `BusinessManager` only cares about the plugins registered in `Configs\baf.data.manager.config`. Let's go there and add this simple line to <businessManager><plugins> section:

```
<add method="*" metaClass="Contact" eventStage="PostMainOperation" type="EPiServ\
er.Commerce.Sample.CustomerContactEvent, EPiServer.Commerce.Sample" />
```

The attributes speak for themselves - here we are listening to all methods for `Contact` meta class. Of course, you can change the method to listen to, "Create" only, for example. You can also change to listen to all meta classes, by changing "Contact" to "*", or listen to another Business Foundation `MetaClass`.

`evenStage` is an interesting attribute - here we define when we want our plugin executed. There are 4 possible values:

```
PreMainOperation = 1
PreMainOperationInsideTranasaction = 2,
PostMainOperationInsideTranasaction = 8,
PostMainOperation = 16,
```

As the names might suggest, the stages are before the main action, before or after the main action, but within same transaction (if your module does some data modification, it should be here), or after

the main action. In our example, we just want to have some notifications when a contact is deleted, so we register our plugin to be `PostMainOperation`.

There's a caveat here, we don't really know the result of the main operation. The only thing we know is there was no error (no exception thrown). So we just assume that it completes successfully.

The final attribute is only the full name of our little type, with the assembly name.

And now, try deleting a contact again. And this time, magic should happen!

The Migration Steps

Migration Step is one of the hidden features of Episerver Commerce. It's been there since Episerver Commerce 7.10 (I remember it better than anything else because it can be considered as my baby). It's not public, it's even in a separated assembly with "Internal" part in its name (EPiserver.Commerce.Internal.Migration). And if you browse its API:s, it has document clearly states that these API:s should only be used for Episerver infrastructure. Episerver also does not support any backward compatibilites on the API:s, so if they want to change the interfaces or classes, to add or remove a method, they are free to do so without having to obsolete anything or have to wait for a major version. You're on your own and will need to update your class if that happens.

Then should you use it?

Well, the answer, as always, is: it depends. A Migration Step is supposed be one-time only operation. The initialization modules provide one-time operation per *every site startup*, but here, it's one per website. Episerver has been using Migration steps as the data-modifications for updates (it's not necessary to have one migration step per update, in facts, the chance you run into migration steps for a minor update is relatively small) . They have the upgrade SQL files to do DDL operations, such as update stored procedures, add/remove columns, etc, and Migration steps for migrating data, fixing something up. It's not strictly separated, thought. Sometimes upgrade SQL files can be used to migrate data if it's easier and faster that way, but you can see where each one should fit.

The idea of Migration Step is quite simple. You implement a special interface (Can you guess it name? Well, `IMigrationStep`), which has one method named `Execute`. The method will return `true` or `false` to indicate its result. Every time your site start, `MigrationManager` will check if there is any steps which are not run, and depends on your setting, it might redirect you to a special page to run remaining steps, or start them automatically. So, migration steps are run after initialization modules, and before anything else. There's a machinism to make sure you have to complete migration steps (for safety reasons, of course. You won't want to run your site with incorrect data). When a step returns true to indicate it ran successfully, `MigrationManager` will store the information in the database to remember for the next time. Otherwise, it does mean some errors happened, you might need to correct them before trying again. When all migration steps are done, you can return to normal operations.

EPiServer Commerce Migration

As a part of updating EPiServer Commerce you may be required to execute migration steps in the context of the web site. Below you can see pending as well as historical migration steps.

Status

Migration completed. Go to start page.

Show migration steps | Show log | More information

Order	Done	Name	Description	Last Run
10	✓	Update Asset Types	Correct the asset type of images for catalog content, in order to show thumbnails in Catalog UI. There was a bug in CatalogImport released with EPiServer Commerce 7.5 which wrongly set the asset type of images to be IContentMedia, which causes missing thumbnails in Catalog UI. Assets added via Catalog UI are not affected by this bug.	2/25/2016 9:47:25 AM
20	✓	Update Permanent Link	Make sure links to catalog contents which were created in a version earlier than EPiServer Commerce 7.10 will still work.	2/25/2016 9:47:25 AM
40	✓	Update Inventory Data	Migrate existing inventory data into tables for the updated provider.	2/25/2016 9:47:25 AM
50	✓	Update Contact Data	Updates user id of existing contacts to the membership user name.	2/25/2016 9:47:25 AM
70	✓	Update Permanent Links For Blocks	Make sure links to catalog contents in blocks, which were created in a version earlier than EPiServer Commerce 7.10, are migrated correctly.	2/25/2016 9:47:25 AM
90	✓	Populate User Permission table	Insert existing permissions from RolePermission into tblUserPermission	2/25/2016 9:47:25 AM
91	✓	Migrate Legacy Admins Role	Migrates users of old legacy Admins role to Administrators role	2/25/2016

Migration steps view

OK let's try to implement a Migration Step.

```
public interface IMigrationStep
{
    /// <summary>
    /// Gets the order of the step. The one with lowest order executes first.
    /// </summary>
    int Order { get; }

    /// <summary>
    /// Gets the name of the step
    /// </summary>
    string Name { get; }

    /// <summary>
    /// Executes the step.
    /// </summary>
    bool Execute(IProgressMessenger processMessenger);

    /// <summary>
    /// Gets or sets the description.
    /// </summary>
    string Description { get; }
}
```

The two things you need to care about is `Order` and `Excecute`. `Order` defines which migration step

to run first - the lower Order, the higher it is in the queue. `Execute` simply runs the code, it takes an `IProcessManager` to let you display the progress in Migration view.

Migration settings:

You can configure the Migration Steps to automatically run when the site starts (and if there are steps which haven't run successfully, of course). You can set it either by setting (Add a key named `AutoMigrateEPiServer` to your appSettings section, with value as `true`), or by code, in your `IInitializationModule`:

```
public void Initialize(InitializationEngine context)
{
    context.Locate.Advanced.GetInstance<MigrationSettings>().AutoMigrate = true;
}
```

If the setting by configuration will always override the one by code.

The migration steps reside in EPiServer.Commerce.Core package, but the migraton view resides in EPiServer.Commerce.UI one. This can be an issue if you have installation without EPiServer.Commerce.UI (in a load-balancing environment for example). If you configured the migration steps to automatically run, then it will run without any issue, but if you didn't, an exception will be thrown during site startup. To avoid that issue, always start the installation with EPiServer.Commerce.UI first.

The migration steps cannot run in the context of Commerce Manager. However, there is a safe check which will throw an exception if you start your Commerce Manager website without having the migration steps completed.

Server Error in '/' Application.

There are incomplete migration steps. The site can be in unstable state. Please navigate to front-end site and complete remaining migration steps, then restart Commerce Manager site. If you see this message on the front-end site, it means no Catalog Content models (decorated with the CatalogContentType attribute) were found. Add a Catalog Content type, or set the appSetting 'episerver:SkipCatalogContentModelCheck' to true to bypass this check. Warning! Bypassing this check in the Commerce Manager site or other application which does not have the content model types available could lead to data loss!

It is not scary as it seems to be

In such case, simply run the migration steps in the front-end site, then restart Commerce Manager.

Part 9: Advanced stuffs

Finally, some advanced stuffs

Caching

> There are only two hard things in Computer Science: cache invalidation and naming things.

– Phil Karlton

My boss likes this quote so much that he usually quotes it when we have a discussion about naming in our reviews. But of course, cache invalidation is even harder and it's critical to your code correctness.

For a high performance site, you will sooner or later have to use cache (Fact: the sooner, the better). Getting cache right is hard - it's an art of balancing between performance, memory and ease (Of course, correctness is a non-compromise requirement).

There are very little few guidelines for caching in a Episerver Commerce site, it depends a lot in your implementation. However, those following best practices are something you should really follow:

- Use `ISynchronizedObjectInstanceCache`. This interface comes from Episerver Framework and it's basically the barebone of caching system in entire Commerce. Unlike its parent interface - `IObjectInstanceCache`, `ISynchronizedObjectInstanceCache` will work in system which is scaling over servers or sites. Some older classes, such as `CatalogCache`, was updated to use `ISynchronizedObjectInstanceCache` internally. It's a quite simple interface with some methods to play with.

> One thing to keep in mind is that `ISynchronizedObjectInstanceCache`, at least the default implementation, only guarantees that the cache validation is synchronized. So if you call `Remove(key)` in your class, you're guaranteed that other instances of your site will not read the obsolete cache value anymore.

- Think about the cache key. Keep it short (while still being unique). The cache engine will have to match a lot of keys and long keys will give more pressures on your CPUs, and they themselves consume memory.
- Don't cache too much. Episerver Commerce already caches (almost) everything it needs, from contents to carts. You should only cache your own stuffs when you really know it'll hit the database if you don't. Having too many objects floating around in cache is a bad thing - it will consume memory (to some points, your system will throw `OutOfMemory` exception, or even worse, start saving RAM to page file), and having too many key will slowdown the cache validation (key matching should still be very fast, but it's the master keys and dependent keys which is the issue).

You might ask what are actually cached in the system, well, more than you'd think:

- Catalog content are cached by `ContentProvider`, except when a content it was loaded with `ContentReference.GetPublishedOrLatest` is true. This cache is managed by CMS system.
- DTO:s are cached by default.
- `Carts` are cached by default.
- `IPriceValue`:s are cached by default.
- Think carefully about the dependent keys, master keys, and the caching mode. It's best when you can fire-and-forget. Having an obsolete cache item might harm your code correctness, and in some situations it's even worse than no cache at all. Internally, Episerver applies a pattern like this, and it works quite well:
  - Define a cache key prefix which is short and unique, following the defined convention, for example: `EP:EC:Catalog`, where Catalog is the area we want to have cache.
  - Define a master key from the cache key: `EP:EC:Catalog:*`.
  - Define a cache key by the prefix + unique id, for example, `EP:EC:Catalog:1`, and when add the cache, add the master key to the list as well.
  - To clear the cache structure, simple call `ISynchronizedObjectInstanceCache.Remove(masterKey)`.
- Think carefully about the expiration time of the cache. Most of the times, it's bad idea to cache indefinitely. Only when you know for certain that something needs to be in cache the entire time of the application, and/or the cache object is expensive to build, it's better to specify an expiration time in your `CacheEvictionPolicy` - a null `TimeSpan` parameter in constructor means that the `CacheEntry` will stay in the cache forever - well - until the cache trimming thread kicks in, which (almost always) a bad thing. It's better to let the cache management knows when to trim cache when it's no longer needed. If you thing a cache entry is likely to be accessed often, just set the `CacheTimeoutType` to `Sliding` instead of `Absolute` - then when your cache object is read, its lifetime will be reset.
- It is hugely important to validate cache properly. Yet validing cache in race condition is hard and can be confusing. In simple cases, locks can be used to avoid caching an outdated object. Upcoming CMS 10 comes with `ReadThrough` extension of `IObjectInstanceCache` which handles that automatically for you.
- Think about the master keys to use - so you cache will automatically be invalidated when a specific object is invalidated. Episerver Commerce has this special class `MasterCacheKeys` - which can be used to get the master keys for almost everything which is cached:
  - `RootKey`: Is the master key for all Commerce caching, so if you call `ISynchronizedObjectInstanceCache.Remove(MasterCacheKeys.RootKey)`, you effectively remove all Commerce cache.
  - `AnyEntryKey` and `AnyNodeKey` are the master keys for all nodes, and entries, respectively.
  - If you want your cache to be dependent on a specific node or entry, you can use `MasterCacheKeys.GetEntryKey` and `MasterCacheKeys.GetNodeKey`.

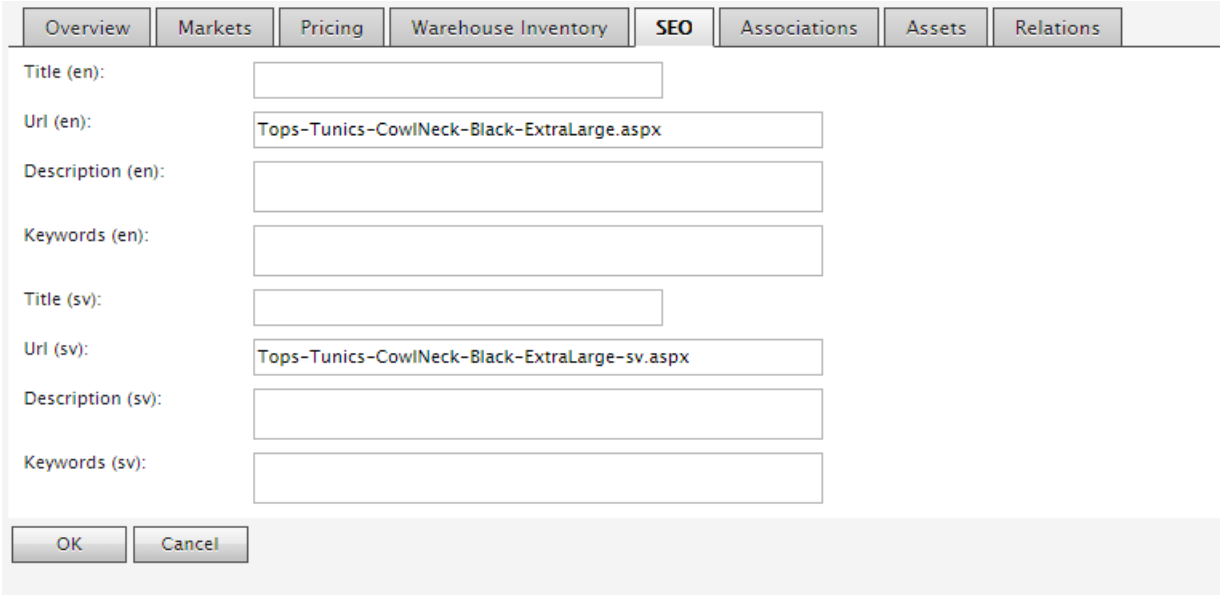
Routing

Routing is important for any sites, and it's even more important for a Commerce site - it helps driving traffic, which usually result in sales. People nowadays search for everything, and ranking higher in the search results yield much better chance of your products being viewed and purchased. (One of) The keys for higher ranking in search engine? Having the best URI you can have!

There are two routing systems in Episerver Commerce (as always!). The old one is based on SEO URI, has been there since Commerce R1, and the new one, HierarchicalCatalogPartialRouter, which is a partial router, was introduced in Commerce 7.5.

The SEO Uri approach

This approach is based on the SEO Uri, so the link to your product will be in form of `http://<yourdomain>/your-product.aspx`. Note that .aspx is the default, but you are free to choose another extension, or even no extension at all. The SEO Uri can be edited in both Commerce Manager and Catalog UI



SEO Uri editing in Commerce Manager

Those information are stored in `CatalogItemSeo` for both Nodes and Entries, which each row represents the information of one node or one entry in one language. Note that only the Uri is unique and is used by the system, it's up to your site implementation to utilize other information such as `Title`, `Description` - just follow best SEO practices.

The SEO Uri regeneration

The `Uri` is unique per language, and if you don't set it explicitly, the catalog system will automatically generate them based on the entry/node's name while still ensure the uniqueness. The default implementation will work like this:

- If the SEO Uri is empty, generates it from the name by replacing any invalid characters, possibly adds the language code, and appends the .aspx extension.
- If the `Uri` being saved is unique, save it.
- If it is not unique, generate again by adding a random string to the end. (it's the `8` first characters of a newly created `GUID`)

This behavior can be overridden by inheriting from `Mediachase.Commerce.Catalog.Data.UniqueSeoGenerator`, and then register your implementation as the default instance.

`UniqueSeoGenerator` is a simple class and all of its methods and properties are `virtual`, to override the `Uri` generation, you can override three following:

```
protected virtual string UriExtension { get; }

public virtual string GenerateSeoUri(string name, string languageCode, bool incl\
udeRandomToken)

protected virtual string GetRandomToken()
```

Usually, you will have the Uri set by some other external systems, or manually, but there are still cases when the Uri:s are automatically generated. There are several tips to make the `Uri` generation more effective:

- Change the default `UriExtension`. It was kept ".aspx" for the backward compatibility, but I don't think that kind of extension is popular anymore.
- Make sure your entries, especially your variations, to have unique names. This will generate better SEO-friendly URIs - no matter how you override `GetRandomToken` - having some artificial random string append to your Uri can't be a good thing.
- When you made your `Uri` public, stick with it. Nothing worse than a hoping customer sees an `404` error for a product they planned to buy, because the `Uri` has been changed. Episerver Commerce has no mechanism to detect such changes to redirect customers to the new Uri, so don't change your Uri:s until absolutely necessary.

How it works

SEO Uri:s are handled by `SeoUriRouter`, which is an implementation of `ISimpleAddressRouteHandler`. `ISimpleAddressRouteHandler` has only one method:

```
SimpleAddressRouteValues GetRouteValues(string urlSegment, ContentReference rout\
eRoot, SegmentContext segmentContext);
```

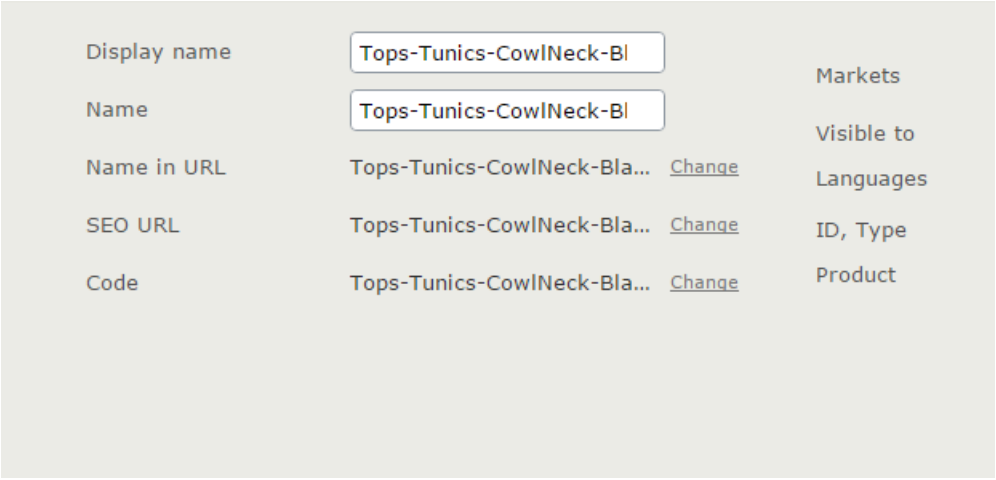
`SeoUriRouter` implements this, and it will only handle the `urlSegment` without any slash (/) in between - so `/this-can-be-a-seo-uri/` is processed, but `/this/cannot/be/a/seo/uri/` is not. It then try to get the content from the SEO Uri, and if you are wondering, there are methods in `ICatalogSystem` to do that - `GetCatalogEntryByUriDto(string seoUri, string language)` and `GetCatalogNodeDto(string seoUri, string language)` [55]. If a matching content to be found, then a SimpleAddressRouteValues value will be returned, indicating this router has processed successfully the `urlSegment`.

Simple enough, right? However, the design of `SeoUriRouter` has some limitations: It's registered by default, and with the routing system, the first router to be able to "catch" the uri wins, so it's be a challenge if you want to replace the behavior. We'll look for more detail in the next section.

The hierarchical approach:

This "new" approach (today it's not new anymore, perhaps we can call it newer?) was introduced in Commerce 7.5 to make the routing of Catalog contents inline with CMS content.

The core concept of this approach is to reflect the structure of contents to the structure of the `Uri`, for the discoverability. With SEO Uri approach, customers will never know how to get to the parent node, to see the other items in same category. With the partial routing, they can. To allow that, every catalog content has a piece of information - `RouteSegment`. This is saved in `CatalogItemSeo` as `UriSegment`, but is hidden for editing in Commerce Manager[56]



RouteSegment, known as "Name in URL" in Catalog UI

[55] I know, I've been telling you to avoid `ICatalogSystem` as much as possible and embrace the new content API:s. This is *almost* always true. However, there are some functions provided by `ICatalogSystem` which are not replaced by new content API:s. In such case, you'll have to use `ICatalogSystem`. It's been improved a lot internally anyway.

[56] Except for the `UriSegments` of catalogs, which are saved in `CatalogLanguage` table.

`RouteSegment` is also supposed to be unique, but only within its sibling. Same as SEO Uri, `RouteSegment` is generated by `UniqueSeoGenerator`, but by another method:

```
public virtual string GenerateUriSegment(string name, bool includeRandomToken)
```

It works pretty similar with `GenerateSeoUri`, except without extension part.

How it works

`HierarchicalCatalogPartialRouter` implements `ICommerceRouter`, which itself inherit from `IPartialRouter<TContent, TRoutedData>`, `IPartialRouter` has two methods for incoming and ongoing URL:

```
PartialRouteData GetPartialVirtualPath(TRoutedData content, string language, Rou\
teValueDictionary routeValues, RequestContext requestContext);
object RoutePartial(TContent content, SegmentContext segmentContext);
```

For incoming URL, the router will parse the URL from the left to the right in `RoutePartial`. The first segment is supposed to match the "root content" (not the `RootContent`, but the configured root) - and the next segment is supposed to match the `RouteSegment` of one of the children of the content match the previous segment, recursively. If a content matches the entire route, it'll be returned.

For outgoing, `GetPartialVirtualPath` will build the URL form right to left. Starting with the content is being routed, it'll append the RouteSegment of its parent content to the left of the URL, until it reach the root content.

`HierarchicalCatalogPartialRouter` has three important parameters to pass to its constructor:

- `RouteStartingPoint`, which is a `Func<ContentReference>`. It is supposed to return a `ContentReference` to the starting point of the route. If you call `CatalogRouteHelper.MapDefaultHierarchialRou routes, bool enableOutgoingSeoUri)`, it'll be the ContentReference.StartPage (aka SiteDefinition.StartPage). Because this is a Func<T>, it'll be evaluated everytime an URL being routed, then it's really dynamic. If you have multiple start pages, it can detect the language the customer is on, and use the right segment. If you set the Shortcut setting for your start page, it can use that, too. Just remember to use a fast Func<ContentReference>, otherwise it'll be hit bad.
- `CommerceRoot`: if you call `CatalogRouteHelper.MapDefaultHierarchialRouter(RouteCollection routes, bool enableOutgoingSeoUri)`, it'll be the "true" Commerce Root, but you are not limited by that. You can construct an instance of `HierarchicalCatalogPartialRouter` which takes a `CatalogContentBase` and then register it directly. This can be done in your initialization module:

```
    var referenceConverter = ServiceLocator.Current.GetInstance<ReferenceConvert\
er>();

    var contentLoader = ServiceLocator.Current.GetInstance<IContentLoader>();

    //Assuming you want to use your only catalog
    var commerceRootContent = contentLoader.GetChildren<CatalogContent>(referenc\
eConverter.GetRootLink()).First();

    //RegisterPartialRouter is an extension from EPiServer.Web.Routing
    RouteTables.Routes.RegisterPartialRouter(new HierarchicalCatalogPartialRoute\
r(startingPoint, commerceRootContent, enableOutgoingSeoUri));
```

This is useful in case you want to hide parts of your catalog structure - because it might be sensitive or boring to show it to customers. For example, using the default `CommerceRoot` means your catalog `RouteSegment` will be included in the url, like this: `http://<your-site>/en/departmental-catalog/Departments/Fashion/` If you have only on catalog and don't want it to be shown, simply point it to your catalog. Your URL will appear to be `http://<your-site>/en/Departments/Fashion/` only. Of course, nothing prevent you from pointing your `RouteStartingPoint` to a category, if you want to. In this example, I can set the Departments category to be the `RouteStartingPoint` and remove it from the URL.

- `enableOutgoingSeoUri`, which is a boolean - if it's true, then `HierarchicalCatalogPartialRouter` will work with SEO URI instead. Any hierarchical URL which matches a catalog content will be redirected to its SEO URI.

Redirecting

A very good thing about Episerver Commerce is that even you enabled the partial routing system, the old SEO Uri:s will continue to work (of course, as long as you keep the Uri:s unchanged). So they can coexist in same website - but that might not what you want - you might want to stick with only one routing system - it's been told that the more popular your URL is, the higher rank it gets in the search engine results.

From hierarchical URL to SEO URI

This one is pretty easy, just call this during your site's initialization, the true parameter means you want to enable Outgoing Seo URI, and the next statement configures the redirect:

```
CatalogRouteHelper.MapDefaultHierarchialRouter(RouteTable.Routes, true);
CatalogRouteHelper.SetupSeoUriPermanentRedirect();
```

From SEO URI to hierarchial URI:

This one is tricky, there is no such built-in method allows us to do so. We can't even override `SeoUriRouter` because it's registered automatically and there is no way to guarantee that our router will be able to run before it. (If a matching router is found, the processing will stop). Let's explore other options.

`EPiServer.Web.Routing.ContentRoute` has two events which might be interesting - `RoutingContent` and `RoutedContent`. As their names might suggest, the former is fired before any routers take place, while the latter is fired after that. We might not want to listen to RoutingContent because it's too early, we'll have to process all URI:s sent to our site, which will slow it down. It's better to only process after the URI has been routed successfully, we prefer it is processed by `SeoUriRouter`. We can do some checks and redirect if necessary.

First, we need to register to `RoutedContent` event:

```
ContentRoute.RoutedContent += Routed_SeoUri;
```

And then implement it. It's another void method which takes an object as sender and an `RoutingEventArgs` as the `EventArgs`. RoutingEventArgs has `SegmentContext` property named `RoutingSegmentContext`, this is what we need. A successfully routed route will set the `RouteObject` of `RoutingSegmentContext` to the content it found. So we can check if that content is a catalog content or not, to see if we should redirect.

But how can we know if the URI was SEO URI, or hierarchial URI? By checking for slash in the between? Not really effective, because the top level content only has one segment. We can mimick SeoUriRouter and try to load the Entry/Node by Uri? Yes that might work but it is not really fast. If we are lucky and the URI was SEO URI, the DTO should have been cached, but if it was hierarchial URI, we will have to hit the database to find nothing!

Solution, well, `HierarchicalCatalogPartialRouter` will process to the last segment, and for each segment, the `LastConsumedFragment` is assigned to next segment, so it'll be empty at the end. `SeoUriRouter`, in other hand, does not really process any segment, so the `LastConsumedFragment` it returns will be the same path as requested.

With that information, we can have this as our method:

```
private static void Routed_SeoUri(object sender, RoutingEventArgs e)
{
    var context = e.RoutingSegmentContext;
    //RoutedObject is supposed to not be null here
    if (!(context.RoutedObject is CatalogContentBase))
    {
        return;
    }

    if(string.IsNullOrEmpty(context.LastConsumedFragment))
    {
        return;
    }
    var urlResolver = ServiceLocator.Current.GetInstance<UrlResolver>();
    context.PermanentRedirect(urlResolver.GetUrl(context.RoutedContentLink));
}
```

And now we can redirect the SEO URI to Hierarchical URI in an effective way.

Appendix

How to ensure a successful upgrade.

I, as an Episerver employee, always recommend our customers to upgrade to the latest version possible. Well, of course that is constrained by a number of things - time (you might be in a deadline next Tuesday), money (upgrading might mean license renewal and that costs money. I'm not allowed to discuss about the licensing, but one thing for sure it's not free), uncertainty (our system is working, why upgrade), etc. But if you can, you always should:

- Newer version is better. More features. More bugs fixed. More performance. Period. Try to import your catalog in Commerce 8. Then try with Commerce 9. It's not even close.
- Fresh code is better. Says, you have on problem and you ask Episerver developers for help in World forums. They (aka me and my colleagues) will try to help you the best they can, but they need to understand the problem first. If you are in very recent version, they can fire up Visual Studio, look at the code and say "OK, it works this way". But if you're in a very old version, it'll be like this "Anyone remember how did it work in <dinosaur> version. I think we changed it last time, right? No one remembers?" and then they will have to dig through a very long history, just to find an answer. You can wait?
- Bug fixes come sooner. Episerver developers don't fix bug, they write perfect code. No I lied. Despite their best effort to avoid bugs and QA's best effort to catch bugs, it still happens, sometimes. If you have a bug in a very recent version, it'll be easier to fix and you can get the fixed version in 2-3 weeks (the time it takes for you to report the bug, then it's triaged, fixed, and tested). If you were in last major version, or even older, there will be no hotfixes for you. And in the end you'll have to upgrade. Why not stay ahead in your game?

Of course, upgrading is not an hassle-free task. (But it has been much more easier in recent years). If you have gone through all the way to upgrade your site from Commerce R1/R2 to latest version, then Congratulations! Next time you're in Stockholm, I will buy you one beer or two. If you are planning to do that, here's something that might help you:

- Backing up is the key. Always back up your site and your databases before you start. I know you've done it, but I never feel that I can urge this enough.
- It's better to work with small steps than a big one. You will have more control. (If you play games, you will know why this is so important. It's better to replay the last 15 minutes when you made a wrong decision, because you saved before an important checkpoint than throwing your controller because you would have to replay the entire day). Always upgrade to next

major version, instead of upgrade to the latest major version in one run. (And in term of major version, Commerce 7.5 counts as one). Upgrade, check if everything, including basic features in both front-end and back-end sites work normally. Back up your sites and databases. Then continue. You can try to upgrade to 2-3 major versions above, and it might work but I would not recommend that. When something happens (and it always happens), you won't know what went wrong.

- In most cases, you should upgrade your front-end site first. Database changes are done there. Migration Steps are done there. When everything is good, you're free to upgrade your Commerce Manager.
- Watch the migration steps closely. All of them should be run successfully before you start using your sites. Note that from Commerce 8.16, Migration steps have been mandatory, you can't no longer, intentionally or accidentally, prevent them from starting.

I've seen a site where migration steps appear to be completed, but they did not successfully run, resulting in some very weird and serious issues, which could not be diagnose until I looked the databases closely. Simply put: Migration steps are to be run and you should not tamper with them in any ways.

- If you upgrade from an older version where catalog content versions do not exist (Commerce R3a or earlier), it does make sense to export your catalogs before upgrading, then import them after upgrading. Truth is, the catalog import/export is much more robust than upgrade scripts and migration steps. This can also apply for newer versions, if you do not care for draft versions and such.

Now what

Phew. It's been a long run. Writing this book is probably the biggest side project I've had. It took me hundreds of hours to think, to write and to rewrite, and I sincerely hope I made it worth the time you spent reading it (and the money you spent buying it, of course).

I tried to cover the most, in the best ways I can, to help you understand the system, how it works, how to use it and use it properly and efficiently. But truth is, Episerver Commerce is a big framework, and with the extensibility, no book can cover everything. This book intends to be a beginning, when you can start your long but exciting and rewarding road with Episerver Commerce development.

I wish you luck in your projects with Episerver Commerce! And I'll meet you on the field.