Episerver Commerce

A problem-solution approach

Quan Mai

Episerver Commerce: A problem - solution approach

Solving problems, one at a time

Quan Mai

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

This version was published on 2019-02-23

For my daughter.

Contents

Chapter 1: Catalog System recipes

Problem 1.0: How to identify a catalog content

Level: Beginner

When you integrate Episerver Commerce with an external system, such as an ERP or PIM system it's common to identify a specific content, likely an entry. Normally, you are given a code, in form a string, and you will have to get the content. How?

Let me introduce you a new friend: `ReferenceConverter`

This small (it's not so small like before) allows you to convert between code and `ContentReference`, and vice versa. To understand these conversions, we need to understand how Episerver Commerce works with identity.

There are 3 ways to identify a catalog content:

- `ContentReference`, to identify a content through APIs. The `ContentReference` is only unique within a system, and it consists of three parts: The `Id` of the content, the `WorkId` of the content (can be empty), and the provider name (can be empty, in case of default CMS content). An example for catalog content `ContentReference` is "123_456_CatalogContent". "CatalogContent" is a constant for all catalog content.
- The code. This is what external systems - price system, inventory system, etc. - understand. As we will learn later, this is used to handle prices, inventories, as well as lineitems. Note that code is only for `Entry` and `Node`, while Catalog uses `Name`.
- The guid. This is an internal identity (although it's supposed to be unique *globally*). For example, when you add a link to a catalog content to a `ContentReference` property, the `ContentReference` will be saved to database as *permanent link* - so the guid will be used to permanently identify the content. Like `code`, guid is permanent and should not be changed.

Now consider these scenarios:

- If you have a `ContentReference`, how to get the code of the product.
- If you have a code of a product, how to load it. (if you already have a `ContentReference`, it is easy.)
- If you have a `Guid` of a product, how to load it.

The preferred way of doing such things is by using `IContentLoader` - or the combination of `IContentLoader` and `ReferenceConverter`. `IContentLoader` already has methods to load contents by a `ContentReference`, or a `Guid`, so that's the easy part. Other than that, `ReferenceConverter` can be used to get the code from `ContentReference`:

```
var code = referenceConverter.GetCode(contentLink);
```

and

```
var contentLink = referenceConverter.GetContentLink(code);
```

Simple, right?

You might not believe the number of times I saw code sample to load the content to get the code from the content. While loading content seems to be easy and cheap to do, but thousands or tens of thousands of such API calls will be a significant cost, both in term of IO and memory (contents are supposed to be cached). `ReferenceConverter` is much more lightweight and faster - so use it whenever possible. Even better, it also has APIs to get codes or content links in batches - which mean you can have even better performance by reducing the overheads further.

One of the most important tips for great Commerce performance is to know which APIs to use. For that purpose, I'd suggest you to go through the developer guide, once. I know the documentation can be boring and dry to follow, but it can make a big impact in your daily work. Trust me, your effort will be rewarded.

Problem 1.1: Get all catalogs

Level: Immediate

From time to time, you will need to get all catalogs in your site. How? As previously stated, the catalog system is fully integrated to content provider system, so to get all catalogs, you would need to call `IContentLoader.GetChildren` of the parent content. But what is the parent content of a catalog?

All of the catalogs are direct children of `RootContent`. This is a virtual and unique content that is not saved anywhere to the database, but rather, is constructed on-the-fly. `IContentLoader.GetChildren` actually uses a `ContentReference`, how to get that of the `RootContent`?

`ReferenceConverter` comes to rescue. To get the `ContentReference` to the `RootContent`, you can use `ReferenceConvert.GetRootLink()`.

To be honest, `GetRootLink()` should have been a property instead. It returns same value all the time, so it does not make sense to be a method.

The code to get all catalogs looks like this:

```
var catalogs = _contentLoader.GetChildren<CatalogContent>(_referenceConverter.GetRoo\
tLink());
```

This code is actually with a gotcha. this overload of `IContentLoader.GetChildren` will use the current culture of the site to get the catalog. So if your site has `en` language as active language, but one of your catalogs is only available in `sv` language, you might be surpised to find that it will not be returned in `catalogs`. If you develop a site with a known set of catalogs (or may be just one), this issue can be discovered quite early. But if you are working on different catalog sets which are unknown when you build the site, or even more, if you are working on a framework level (like I am), then this can be well hidden from your testing, only to come back and bite you later. Solution? You should use the other overload of `GetChildren` which takes a `LoaderOptions` parameter, which allows you to fall back to master language.

This is the right way to do it:

```
var catalogs = _contentLoader.GetChildren<CatalogContent>(_referenceConverter.GetRoo\
tLink(), new LoaderOptions() { LanguageLoaderOption.MasterLanguage() });
```

Problem 1.2: Traverse the catalog.

Level: Beginner

One in a while, you might need to iterate over the catalog items.

There are two ways to do that, the old, legacy way of `ICatalogSystem`, and the new way of `IContentLoader`. If you come from the Commerce R1/R2 (which sounds like decades ago), then this is the way you should do it:

```
CatalogSearchOptions options = new CatalogSearchOptions();
CatalogSearchParameters searchParams = new CatalogSearchParameters();
int totalCount = 0;
//Check for total count
catalogSystem.FindItemsDto(searchParams, options, ref totalCount);
while (totalCount > 0 && recordsCount < totalCount)
{
    options.RecordsToRetrieve = 500; //Batches of 500
    options.StartingRecord = recordsCount;
    var responseGroup = new CatalogEntryResponseGroup();
    var catalogEntriesDto = catalogSystem.FindItemsDto(searchParams, options, ref to\
talCount, responseGroup);
    foreach (var row in catalogEntriesDto.CatalogEntry)
    {
        //Do your stuffs here.
    }
}
```

While using `ICatalogSystem` is something "in the past", batch operations are still something missing from the new content APIs. I would wholeheartedly recommend to use the new APIs whenever possible, but this one place where you might find `ICatalogSystem` works better. When we use the Search APIs like `FindItemsDto` above, it returns a `CatalogEntryDto`, which allows you to do editing and save it back. If you were using the content APIs, you would have to save every content one by one.

Catalog Search APIs are also fairly powerful and flexible. For example, what if you just want to search for a specific kind of entries? You can use `CatalogSearchOptions.Classes` for such purpose

```
var searchOptions = new CatalogSearchOptions
{
    CacheResults = true,
    StartingRecord = 0,
    ReturnTotalCount = true,
    RecordsToRetrieve = 10000,
    Classes = new StringCollection { "CustomVariationContent" }
};
```

where `CustomVariationContent` is the name of the `MetaClass` you want to filter on.

But in a way, `ICatalogSystem` is so 2010! `FindItemsDto` is useful for several scenarios, but you might want to use some more "modern" APIs.

If you want to move away from `ICatalogSystem` entirely, `IContentLoader` has everything you need for loading content. Here's the snippet for traversing the catalog structure using the content APIs:

```
public virtual IEnumerable<T> GetEntriesRecursive<T>(ContentReference parentLink, Cu\
ltureInfo defaultCulture) where T : EntryContentBase
{
    foreach (var nodeContent in LoadChildrenBatched<NodeContent>(parentLink, default\
Culture))
    {
        foreach (var entry in GetEntriesRecursive<T>(nodeContent.ContentLink, defaul\
tCulture))
        {
            yield return entry;
        }
    }

    foreach (var entry in LoadChildrenBatched<T>(parentLink, defaultCulture))
    {
        yield return entry;
    }
}

private IEnumerable<T> LoadChildrenBatched<T>(ContentReference parentLink, CultureIn\
fo defaultCulture) where T : IContent
{
    var start = 0;

    while (true)
    {
        var batch = _contentLoader.GetChildren<T>(parentLink, defaultCulture, start,\
 50);
        if (!batch.Any())
        {
            yield break;
        }

        foreach (var content in batch)
        {
            // Don't include linked products to avoid including them multiple times \
when traversing the catalog
            if (!parentLink.CompareToIgnoreWorkID(content.ParentLink))
            {
                continue;
            }

            yield return content;
        }
        start += 50;
    }
}
```

Here we are using `IContentLoader` to load the content recursively. The building block is `LoadChildrenBatched` - which allows us to load the children of a specific type, from a parent link. The reason we need a batch size of 50 here, is because contents are much heavier than the DTO in previous snippet. We can easily load 500 rows for the DTO in each go, but that number of contents might be able to slow the system down.

We also need to make sure that we only return the contents which are true “children” of the parent. If you comes from CMS world, you should know that in Commerce, the catalog items can be linked to other. So on entry can be true children of a node, and yet it is still linked to other several nodes. That is the check `parentLink.CompareToIgnoreWorkID(content.ParentLink)` is for.

Now with `LoadChildrenBatched`, we can get the entries from a starting point, recursively. First we get the children nodes of that starting point, and then call `GetEntriesRecursive` on those nodes - recursively. Then we call `LoadChildrenBatched` on the starting point to get its direct children.

This content API approach allow you to use the strongly typed content types. And because everything is in the content, you don't have to load the `MetaObject` separately. In scenarios when you just want to load contents, and don't have to save it back, this can work very well, and should be your preferred option.

This will be used for later recipes - you will need it every time you need to do something catalog-wise.

Problem 1.2.2: Find unpublished variants

Level: Immediate

In previous recipe we talked about how to traverse the catalog. Let's try to use that recipe in something useful. One real-world scenario: A customer has multiple languages (8 of them). They need to make sure all variants are published in all languages. That is of course a reasonable request, but there is no feature builtin for such requirement. But good news is that can be done with ease. If you want to try this as practice, go ahead – I think it's a good exercise for your Episerver Commerce-fu skills.

The code to do that would be this:

```
foreach (var catalogContent in catalogContents)
{
    foreach (var variant in GetEntriesRecursive<VariationContent>(catalogContent.Con\
tentLink, catalogContent.MasterLanguage))
    {
        var versions = _contentVersionRepository.List(variant.ContentLink.ToReferenc\
eWithoutVersion());
        var missingLanguages = versions
            .GroupBy(l => l.LanguageBranch)
            .Where(g => g.All(v => v.Status != VersionStatus.Published))
            .Select(v => v.Key);
        if (missingLanguages.Any())
        {
            _log.Information($"Variant {variant.Code} with content link {variant.Con\
tentLink} is not published in {String.Join(",", missingLanguages)}");
        }
    }
}
```

At first we need to get a list of catalogs available in the system, and use each of them as the starting point for `GetEntriesRecursive`. And for each variant, we use `IContentVersionRepository` to list all versions, then group them by language. If one language has no published version, we log information about that version. Here for simplicity, we just use the `ILogger`. But in real world scenarios, you might want to store the information somewhere, and even generate a mail to send to responsible people on a regular basis.

Problem 1.3: Read only Catalog UI

Level: Immediate

It's not uncommon to update the catalog data by an external system, mostly from a PIM – Product information management system. In such cases, it might not make senses to enable editing in Catalog UI. You might need the new UI for the other parts, such as Marketing UI, but you wouldn't want the editors to accidentally update the product information – because those would be lost, anyway.

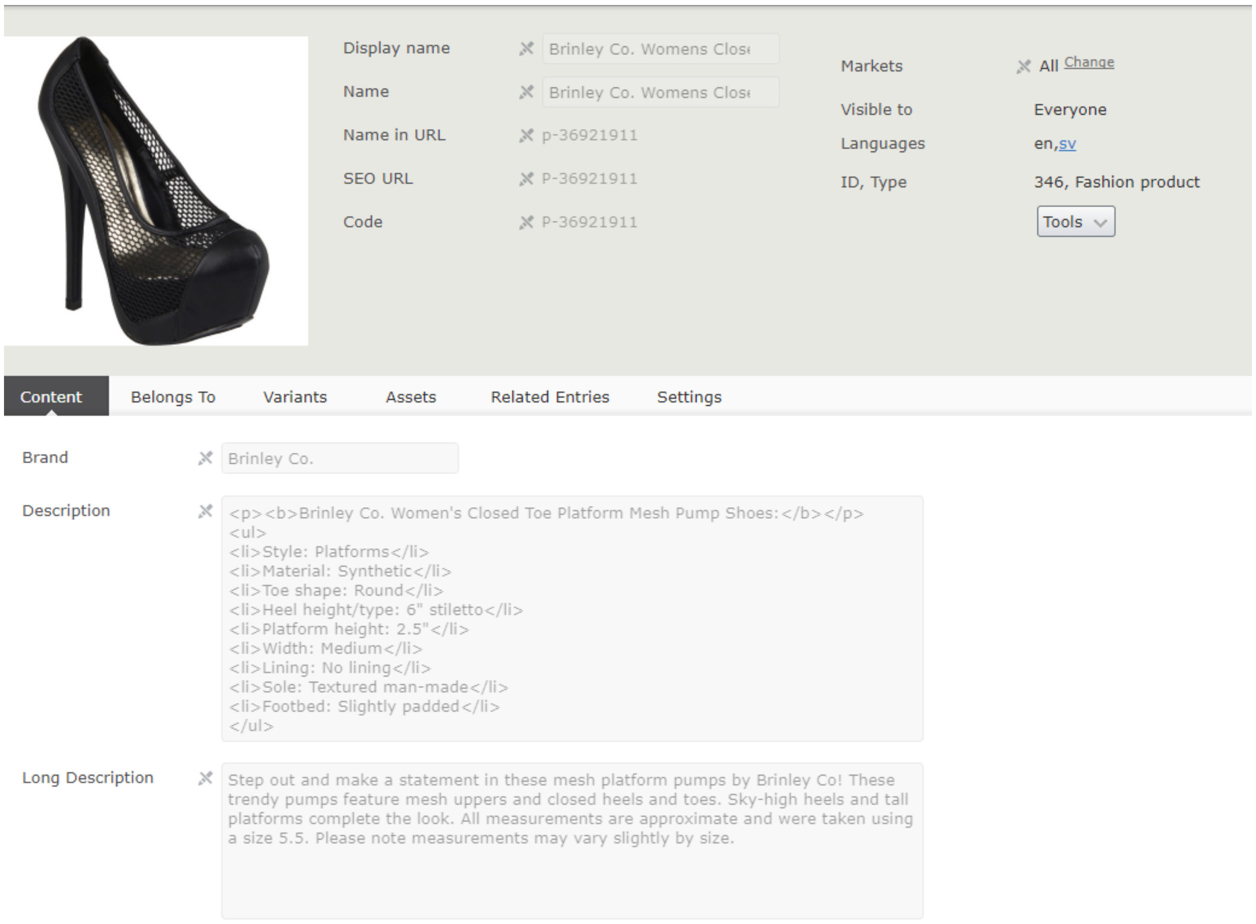
Is there away to do it? Yes, there is.

Catalog UI is built around the content provider concept. CatalogContentProvider is the provider for catalog content, and it has distinct features compared to the default content provider. Having no waste basket, for example, is one of them. By overriding the "provider capacity", CatalogContent-Provider tells the content provider system that "Hey, I don't have, and don't want to support, waste basket concept". And the content provider system respects that by hiding the waste basket ("Move to waste basket" command from the context menu is hidden explicitly in Catalog UI code). Can we use that for our cause? Luckily, yes.

using EPiServer.Core;

```
namespace EPiServer.Reference.Commerce.Site.Features
{
    public class ReadonlyCatalogContentProvider : CatalogContentProvider
    {
        public override ContentProviderCapabilities ProviderCapabilities => ContentP\
roviderCapabilities.MultiLanguage;
    }
}
```

And then, register `ReadonlyCatalogContentProvider` as the implementation of `CatalogContentProvider` you want to use. With that simple code, we would have this in the UI



Display name: Brinley Co. Womens Clos
Name: Brinley Co. Womens Clos
Name in URL: p-36921911
SEO URL: P-36921911
Code: P-36921911

Markets: All Change
Visible to: Everyone
Languages: en,sv
ID, Type: 346, Fashion product

Brand: Brinley Co.

Description:

```
<p><b>Brinley Co. Women's Closed Toe Platform Mesh Pump Shoes:</b></p>
<ul>
<li>Style: Platforms</li>
<li>Material: Synthetic</li>
<li>Toe shape: Round</li>
<li>Heel height/type: 6" stiletto</li>
<li>Platform height: 2.5"</li>
<li>Width: Medium</li>
<li>Lining: No lining</li>
<li>Sole: Textured man-made</li>
<li>Footbed: Slightly padded</li>
</ul>
```

Long Description: Step out and make a statement in these mesh platform pumps by Brinley Co! These trendy pumps feature mesh uppers and closed heels and toes. Sky-high heels and tall platforms complete the look. All measurements are approximate and were taken using a size 5.5. Please note measurements may vary slightly by size.

Read-only Catalog UI, well, almost

This does not make the inventories and prices truly "read-only", because they are not managed by the content provider system. We will get into those parts in later recipes.

In Commerce 11.6, Commerce introduced the ability to handle permission for catalog items (which, at first, only limited to catalogs and categories. Entries will inherit permission settings from their direct parents.). That would be the preferred and more flexible way to control who can do what in the Catalog UI (and even at the API:s level). However, for this specific purpose of "read-only catalog UI", this still works well given the effort you have to put in.

Problem 1.4: Choose an URL style

Level: Beginner

There are two url styles in Episerver Commerce, the hierarchical one where the urls reflect the catalog structure, so it's in form of "https://yourwebsite.com/Catalog/Parent-Node/Child-Node/Product". And the other one where you link directly to a product by a SEO URI "https://yourwebsite.com/produ
name". It's up to your business to decide which suite you most, but when you make up your mind, you need to know where to change between style.

By default installation, the hierarchical url style is used. That is done by this registration in an initialization module:

```
CatalogRouteHelper.MapDefaultHierarchialRouter(RouteTable.Routes, false);
```

`MapDefaultHierarchialRouter` takes two parameter - a `RouteCollection`, and a boolean value to let the framework knows if you want to build outgoing URLs in SEO URI format. Then how to to switch to other style? When, you can simply use other method:

```
CatalogRouteHelper.MapDefaultHierarchialRouter(RouteTable.Routes, true);
CatalogRouteHelper.SetupSeoUriPermanentRedirect();
```

Build it and tada! Now your products will be redirected to the Seo Uris.

Problem 1.4.1: Switch between url styles

Level: Immediate

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. Episerver Commerce in particular and Episerver in general does not handle changed Uri:s). 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.

Redirecting all hierarchical urls to SEO Uri:s is easy, we just did it in the previous recipe. But vice versa it’s 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 mimic 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.

Problem 1.5: Remove the catalog name from product url

Level: Immediate

Episerver Commerce supports multiple catalogs, and it’s not uncommon to have more than one in your system. However, it’s also uncommon to have only one. And then it’s only nature to remove the catalog name from the product url. That helps shortening the url (which makes it easier for customers to remember, but I don’t really think anyone remember the urls these days), and it helps hiding your catalog structure (You don’t want to expose it unnecessarily, right?).

The hierarchical routing system in Episerver Commerce is built based on catalog structure. By default, you will register it by this snippet:

```
CatalogRouteHelper.MapDefaultHierarchialRouter(RouteTable.Routes, false);
```

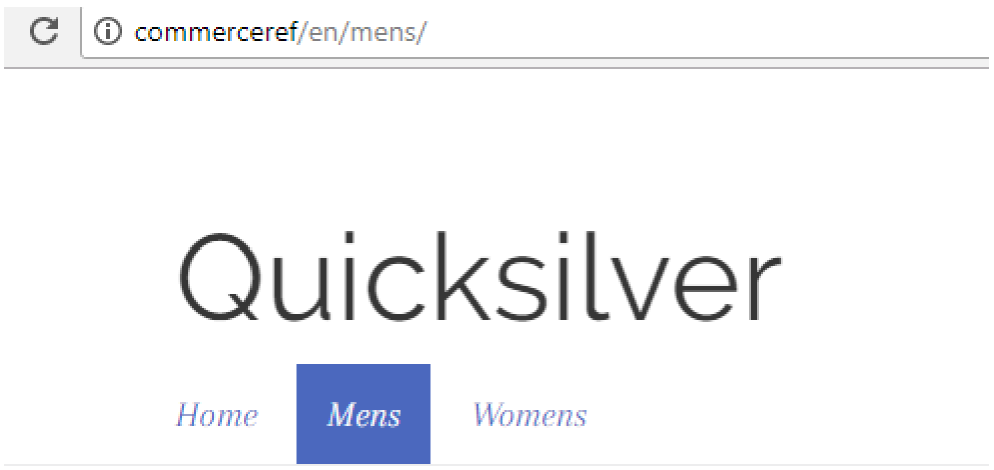
`CatalogRouteHelper` is a wrapper so it’s easier to use, but if you want to remove the name of the catalog from url, we have to leave `CatalogRouteHelper` out of play, but use `HierarchicalCatalogPartialRouter` directly.

```
var referenceConverter = context.Locate.Advanced.GetInstance<ReferenceConverter>();
        var contentLoader = context.Locate.Advanced.GetInstance<IContentLoader>();
        var commerceRootContent = contentLoader.GetChildren<CatalogContent>(referenc\
eConverter.GetRootLink()).FirstOrDefault();
        Func<ContentReference> startingPoint = () => ContentReference.IsNullOrEmpty(\
SiteDefinition.Current.StartPage) ?
               SiteDefinition.Current.RootPage :
               SiteDefinition.Current.StartPage;
        RouteTable.Routes.RegisterPartialRouter(new HierarchicalCatalogPartialRouter\
(startingPoint, commerceRootContent, false));
```

Much longer code, right? But in a way, this is “less magic” - you can look into how thing works and understand them yourself. The first two lines are quite simple, we just get instances of `ReferenceConverter` and `IContentLoader`. Note that this code is usually run in the context of `Initialize` method of an `IInitializeModule`, so here we have `content` parameter which is of type `InitializationEngine`, and the way to get a registered instance is a bit different from a common pattern `ServiceLocator.Current.GetInstance`.

Now the important part is the `commerceRootContent`. This is the point you want `HierarchicalCatalogPartialRouter` to stop looking. Because we want to skip the catalog name, we pass the first catalog here (`CatalogRouteHelper.MapDefaultHierarchialRouter` pass the Catalog Root, which is the parent of all catalogs). You can of course pass another catalog, a simple LINQ statement would be easy enough.

The second important part is the page you want to “anchor” your catalog to. In this case, we use the start page (which is also the most common option). Note that we fall back to root page as a safe check (for example, in the context of scheduled job, start page might be empty, and therefore any routing request to catalog content can fail). You can, of course, have a CMS page, named “Products” as your starting point, just update `startingPoint` to return the `ContentReference` of that page.



We removed catalog name from url, and use start page as the starting point

This, however, doesn't work if you have more than one catalog, which is understandable (how does the system know which catalog to look into?) Technically, you can write your own implementation of `HierarchicalCatalogPartialRouter` and process the first level nodes yourself. But that would be out of scope for this recipe.

Problem 1.6: Multiple catalogs with same url

Level: Immediate

This is from a question I received today. A business is having an Episerver Commerce instance with multiple sites and multiple catalogs set up. They want to make sure each site will use one catalog, and all of them will share the same url for catalog structure. So it'll be "https://site-a.com/products/category/", and "https://site-b.com/products/category/". Site A and site B are using different catalogs.

Is this doable? Yes! It's just a matter of magic with the routing. This time, we would need to do an implementation of `HierarchicalCatalogPartialRouter` ourselves. First, let's create a template for it:

```
public class MultipleSiteCatalogPartialRouter : HierarchicalCatalogPartialRouter
{
    private readonly IContentLoader _contentLoader;

    public MultipleSiteCatalogPartialRouter(Func<ContentReference> routeStartingPoin\
t, CatalogContentBase commerceRoot, bool enableOutgoingSeoUri, IContentLoader conten\
tLoader)
        : base(routeStartingPoint, commerceRoot, enableOutgoingSeoUri)
    {
        _contentLoader = contentLoader;
    }
}
```

We need to make sure each site uses the catalog it's assigned for by overriding `FindNextContentInSegmentPair`. When the router works on the url, it will start with from the left most segment, and figure out which content maps to that segment. We need to override that to set the catalog we want to use.

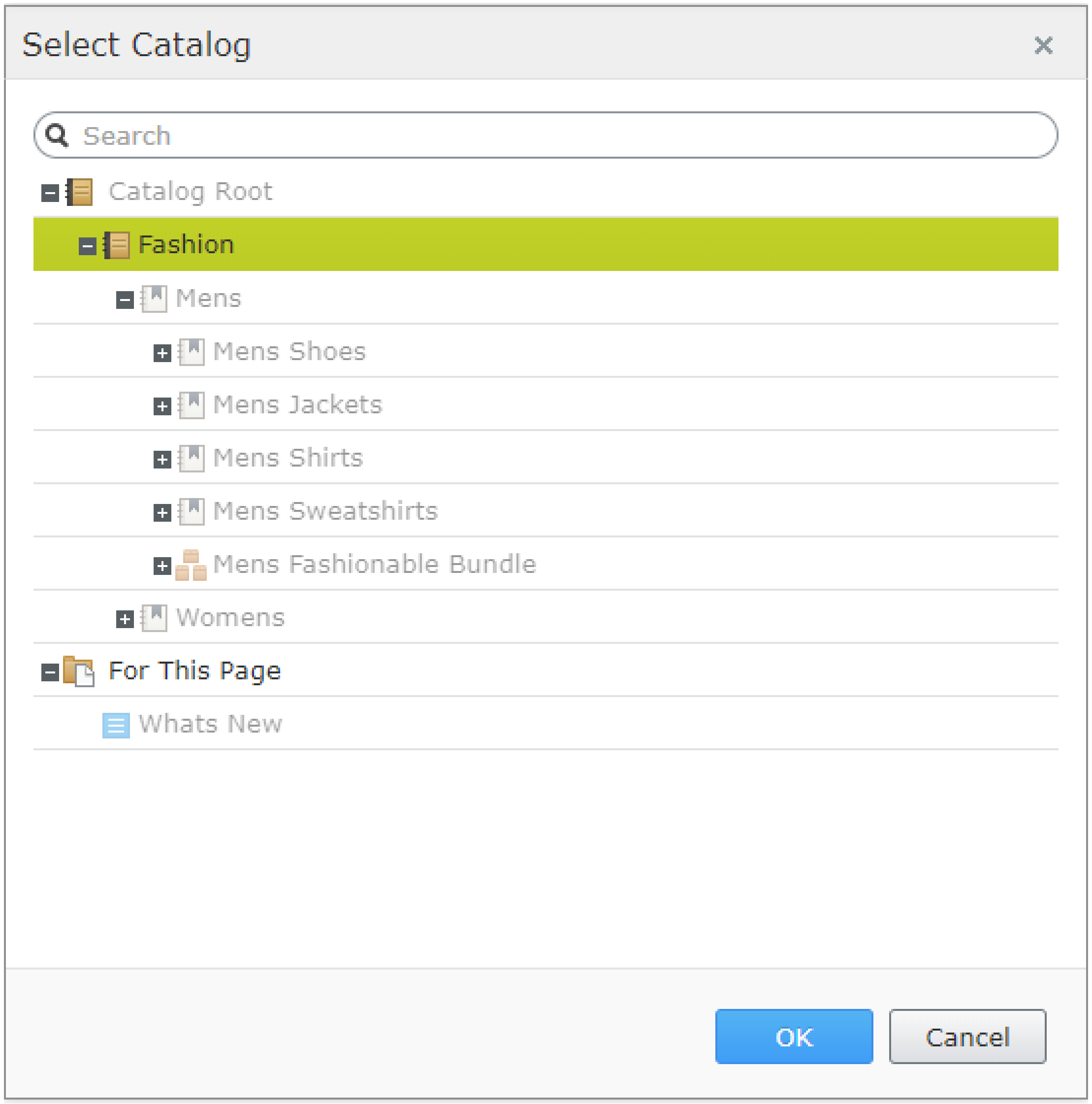
```
protected override CatalogContentBase FindNextContentInSegmentPair(CatalogContentBas\
e catalogContent, SegmentPair segmentPair,
    SegmentContext segmentContext, CultureInfo cultureInfo)
{
    if (catalogContent.ContentType == CatalogContentType.Root)
    {
        CatalogContent definedCatalogContent;
        var definedCatalogLink = _contentLoader.Get<StartPage>(RouteStartingPoint).C\
atalogLink;
        if (_contentLoader.TryGet<CatalogContent>(definedCatalogLink, cultureInfo, o\
ut definedCatalogContent))
        {
            return definedCatalogContent;
        }
    }
    return base.FindNextContentInSegmentPair(catalogContent, segmentPair, segmentCon\
text, cultureInfo);
}
```

This will come with a side effect is that your catalog part in your url no longer matters. It can be "https://commercerecipes/en/abc/womens/" or "https://commercerecipes/en/fashion/womens/" and the router can still figure it out, like magic. You might or might not want this side effect, so you can check if the segment matches an expected, defined value (we will discuss next), and return null if it does not match.

There are several way to decide which catalog to use, but because here we want it to attach to a site, we will get it from the start (aka Home) page. Just add a property to the start page.

```
[AllowedTypes(typeof(CatalogContent))]
[UIHint(UIHint.CatalogContent)]
[Required]
public virtual ContentReference CatalogLink { get; set; }
```

This would render to a nice selector which is required and only allows catalogs:



Only catalogs are allowed

Because it's possible that an editor will forget to set a linked catalog in the start page, we need don't want the site to blow up, so we will just fallback to the default router.

This also explains why `StartingPoint` needs to be a `Func<ContentReference>` instead of just a `ContentReference` - Episerver needs to support multiple sites scenarios. If `StartingPoint` just takes a `ContentReference`, it'll be a static start page for that instance. By taking `Func<ContentReference>` we can get the start page from the context, which is what we are going to do.

But that's not yet enough. We make sure each site will use a different catalog for routing. That's good, but we still all the catalogs to appear the same in the URL. With above implementation, we handled the incoming URL, but not yet outgoing URL.

Problem 1.6.2: Building the outgoing URLs

Level: Immediate.

In previous recipe we talked about multiple catalogs with same "UriSegment" - which we had a working implementation for incoming URL, i.e. when a customer visit a product url, we know which catalog we should choose from. But we still need to cover the generation of outgoing URL. I.e. when we link a product (For example, from a campaign page), we need to generate an URL which take the "catalog-less" pattern into account.

We need to understand how the outgoing URL is built. The hierarchical router builds the URL by the `RouteSegment` of contents. However, we want to the urls appear to have same catalog, so the `RouteSegment` part for the catalogs must be the same, regardless of the true catalogs. Because all catalogs are on same level, their `RouteSegment` must be unique - and this is enforced from Framework level (which is understandable, otherwise, how can it know which content to choose). We can't also rely on name, which is required to be unique.

What can we do?

Well - we already have magic router which automatically figures out which catalog to use regardless of the `RouteSegment` for catalog, so we can just return some magic value. The tricky part is to know which method to override. Luckily for us, Commerce provides us `TryGetRouteSegment`, which returns the `RouteSegment` of a content in a specific language.

```
protected override bool TryGetRouteSegment(ContentReference contentLink, string lang\
uage, out string segment)
{
    CatalogContent catalogContent;
    if (_contentLoader.TryGet<CatalogContent>(contentLink, CultureInfo.GetCultureInf\
o(language), out catalogContent))
    {
        segment = "products";
        return true;
    }
    return base.TryGetRouteSegment(contentLink, language, out segment);
}
```

The code is fairly simple, we try to load the catalog content, and if succeed, and return a fixed value - “products”, otherwise, we fallback to the default implementation.

We return something hard-coded here, but you can of course make that a configurable value. Just keep in mind that changing URLs will hurt your SEO and also make your customers confusing, so do it as little as possible. An alternative is to have a plugin to handle the “changed” URLs for you, so at least it will redirect your customers to the correct, working link.

The final part is to register our new router. This is almost as same as with our previous recipe to remove the catalog name from url - we just need to use our new class here.

```
Func<ContentReference> startingPoint = () =>
            ContentReference.IsNullOrEmpty(SiteDefinition.Current.StartPage)
                ? SiteDefinition.Current.RootPage
                : SiteDefinition.Current.StartPage;
var referenceConverter = ServiceLocator.Current.GetInstance<ReferenceConverter>();
var contentLoader = ServiceLocator.Current.GetInstance<IContentLoader>();
var commerceRootContent = contentLoader.Get<CatalogContentBase>(referenceConverter.G\
etRootLink());
RouteTable.Routes.RegisterPartialRouter(new MultipleSiteCatalogPartialRouter(startin\
gPoint, commerceRootContent, false, context.Locate.ContentLoader()));
```

And that would mean your outgoing URLs will respect the fixed catalog segment you wanted them to.

Problem 1.6.3: Beyond multiple sites

Level: Immediate.

In the previous recipe we talked about how you can implement routing for multiple sites scenario. But the it’s not just that. I got another request recently, and the scenario is slightly different than what we discuss previously. Instead of having two sites, https://yoursite.com and https://yoursite.se, they only have one site, but with two landing pages: https://yoursite.com/private for individual customers, and https://yoursite.com/business for business customers. They also have two categories, “private”, and “business”, which are direct children of the catalog, and should be used for each landing pages. So they would want to have https://yoursite.com/private/cars pointing to “car” category under “private”, and https://yoursite.com/business/trucks pointing to “truck” category under “business”.

Can that be done?

Yes, but not with the same technique we have done previously. Episerver Commerce routing still does not support multiple roots, at least not officially. It can be done, but you would have to do a lot of works yourself, not to mention that if you run into problems, you're out of luck.

One of the rules when it comes to Episerver support service is if you need custom development - you will need to use expert services, which is not free.

Things can be much easier if you think a little outside of the box. In this case, why not make "private" and "business" the category in the catalog structure. By doing so we can eliminate the entire "routing" problem. It's even builtin and works out of the box. With the new Catalog UI, you can do as much as editing as you want with the content.

Well, almost.

There might be reasons that editors would still want "private" and "business" as CMS pages. I'm going to be honest here: the CMS pages still have the (slight) advantages over Commerce contents when it comes to editing. Better support for blocks, for example. Also the catalog can be imported from an external source (a PIM system, most likely), making editing the Commerce content less appealing.

But that can be fixed easily. There is no one saying you can't display a CMS page for a catalog content.

We first need define a landing category type

```
[CatalogContentType(
    GUID = "a23da2a1-7843-4828-9322-c63e28059f6b",
    MetaClassName = "LandingNode",
    DisplayName = "Landing Node",
    Description = "Market for landing node.")]
[AvailableContentTypes(Include = new[]
{
    typeof(NodeContent)
})]
public class LandingNode : NodeContent
{
    [AllowedTypes(typeof(PageData))]
    [Required]
    public virtual ContentReference LandingPageLink { get; set; }
}
```

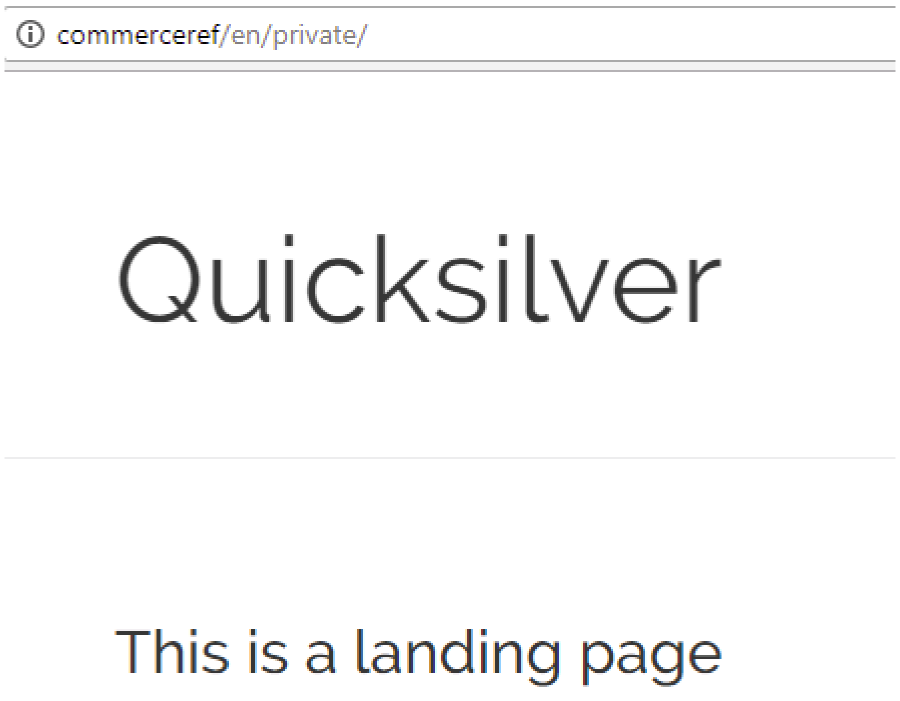
Note that here I'm using a property to point to the true landing page, but you are not limited to do so. The `ContentReference` property is used for simplicity, but you can find the page by name, or by defined values in Start page. What we need is just a controller which takes care of this content type:

```
public class LandingNodeController : ContentController<LandingNode>
{
    private IContentLoader _contentLoader;

    public LandingNodeController(IContentLoader contentLoader)
    {
        _contentLoader = contentLoader;
    }

    // GET: LandingNode
    public ActionResult Index(LandingNode currentContent)
    {
        var landingPage = _contentLoader.Get<LandingPage>(currentContent.LandingPage\
Link);
        //Get the model and return the view
        return View("~/Views/LandingPage/Index.cshtml", landingPage);
    }
}
```

The final piece would be removing the catalog name from the Url, which is discussed in a previous recipe.



Great success, well, just not too fancy

Problem 1.6.4: Url without catalog name in multiple catalogs scenario

Level: Advanced

In previous recipe we talked on how to remove the catalog name from the url. But that only works in one catalog scenario. How's about you have multiple catalogs?

We can reuse the recipe above to make it works for us. It works great with http://yoursite.com/en/fashion/mens/mens-shoes/p-36127195/ or http://yoursite.com/en/products/mens/mens-shoes/p-36127195/. But it does not work with http://yoursite.com/en/mens/mens-shoes/p-36127195/. At least not yet.

In previous recipe we return the catalog from the `FindNextContentInSegmentPair`, based on the linked catalog content from start page. That's why the catalog name is there in the url. How can we ignore that? Well, we don't return the catalog, we return the first-level category.

The updated router will look like this

```
protected override CatalogContentBase FindNextContentInSegmentPair(CatalogContentBas\
e catalogContent, SegmentPair segmentPair,
SegmentContext segmentContext, CultureInfo cultureInfo)
{
    if (catalogContent.ContentType == CatalogContentType.Root)
    {
        CatalogContent definedCatalogContent;
        var definedCatalogLink = _contentLoader.Get<StartPage>(RouteStartingPoint).C\
atalogLink;
        if (_contentLoader.TryGet<CatalogContent>(definedCatalogLink, cultureInfo, o\
ut definedCatalogContent))
        {
            var nodes = _contentLoader.GetChildren<NodeContent>(definedCatalogLink);
            return nodes.FirstOrDefault(n => n.RouteSegment.Equals(segmentPair.Next,\
 StringComparison.OrdinalIgnoreCase));
        }
        return null;
    }
    return base.FindNextContentInSegmentPair(catalogContent, segmentPair, segmentCon\
text, cultureInfo);
}
```

It's not that complicated, right? If we are working on the `CatalogRoot`, and if we can get the catalog content as linked in start page (just like previous recipe), we return the first-level category that has `RouteSegment` matches with the next segment in the url. By doing that, we can skip the catalog, and go directly to the category

Mission accomplished!

Problem 1.6.5: Url without categories

Level: Advanced

Another routing recipe! Because each site might have certain requirements on how their URLs should look like. And I want to show how flexible the routing system in Commerce can be. This is another question from World forums[1]: "I would like all variations to have the following url structure, regardless of where they are in the catalog: https://sitename/products/variationcode. https://sitename/products/ is the product landing page".

Can it be done? Yes. But let's make it a bit different by allowing routing to products, instead of variants.

As usual we need to inherit from `HierarchicalCatalogPartialRouter`, this time, one more dependency is needed.

[1]https://world.episerver.com/forum/developer-forum/Episerver-Commerce/Thread-Container/2017/7/custom-catalog-routing/

```
public class DirectToProductCatalogPartialRouter : HierarchicalCatalogPartialRouter
{
    private readonly IContentLoader _contentLoader;
    private readonly ReferenceConverter _referenceConverter;

    public DirectToProductCatalogPartialRouter(Func<ContentReference> routeStartingP\
oint, CatalogContentBase commerceRoot,
    bool enableOutgoingSeoUri,
    IContentLoader contentLoader,
    ReferenceConverter referenceConverter)
        : base(routeStartingPoint, commerceRoot, enableOutgoingSeoUri)
    {
        _contentLoader = contentLoader;
        _referenceConverter = referenceConverter;
    }
}
```

First we need to register the `routeStartingPoint`:

```
var referenceConverter = context.Locate.Advanced.GetInstance<ReferenceConverter>();
var contentLoader = context.Locate.Advanced.GetInstance<IContentLoader>();
var commerceRootContent = contentLoader.Get<RootContent>(referenceConverter.GetRootL\
ink());
Func<ContentReference> startingPoint = () => ContentReference.Parse("23"); //Your la\
nding page
RouteTable.Routes.RegisterPartialRouter(new DirectToProductCatalogPartialRouter(star\
tingPoint, commerceRootContent, false, contentLoader, referenceConverter));
```

Now to handle the incoming and outgoing url, we need to override `TryGetRouteSegment`:

```
protected override bool TryGetRouteSegment(ContentReference contentLink, string lang\
uage, out string segment)
{
    var cultureInfo = string.IsNullOrEmpty(language)
        ? CultureInfo.CurrentCulture
        : CultureInfo.GetCultureInfo(language);

    ProductContent productContent;
    if (_contentLoader.TryGet<ProductContent>(contentLink, cultureInfo, out productC\
ontent))
    {
        segment = productContent.Code;
```

```
            return true;
        }

        segment = "";
        return true;
}
```

Simple, isn't it? If the `contentLink` points to a `ProductContent`, we return the product code, which will be in the url, but if not, we just return an empty string. The default implementation of `HierarchicalCatalogPartialRouter` will know how to handle such cases, and will not add it to the url.

The handle of `CultureInfo` is important. We need this to work in both case when "language" is set properly, and when it is null.

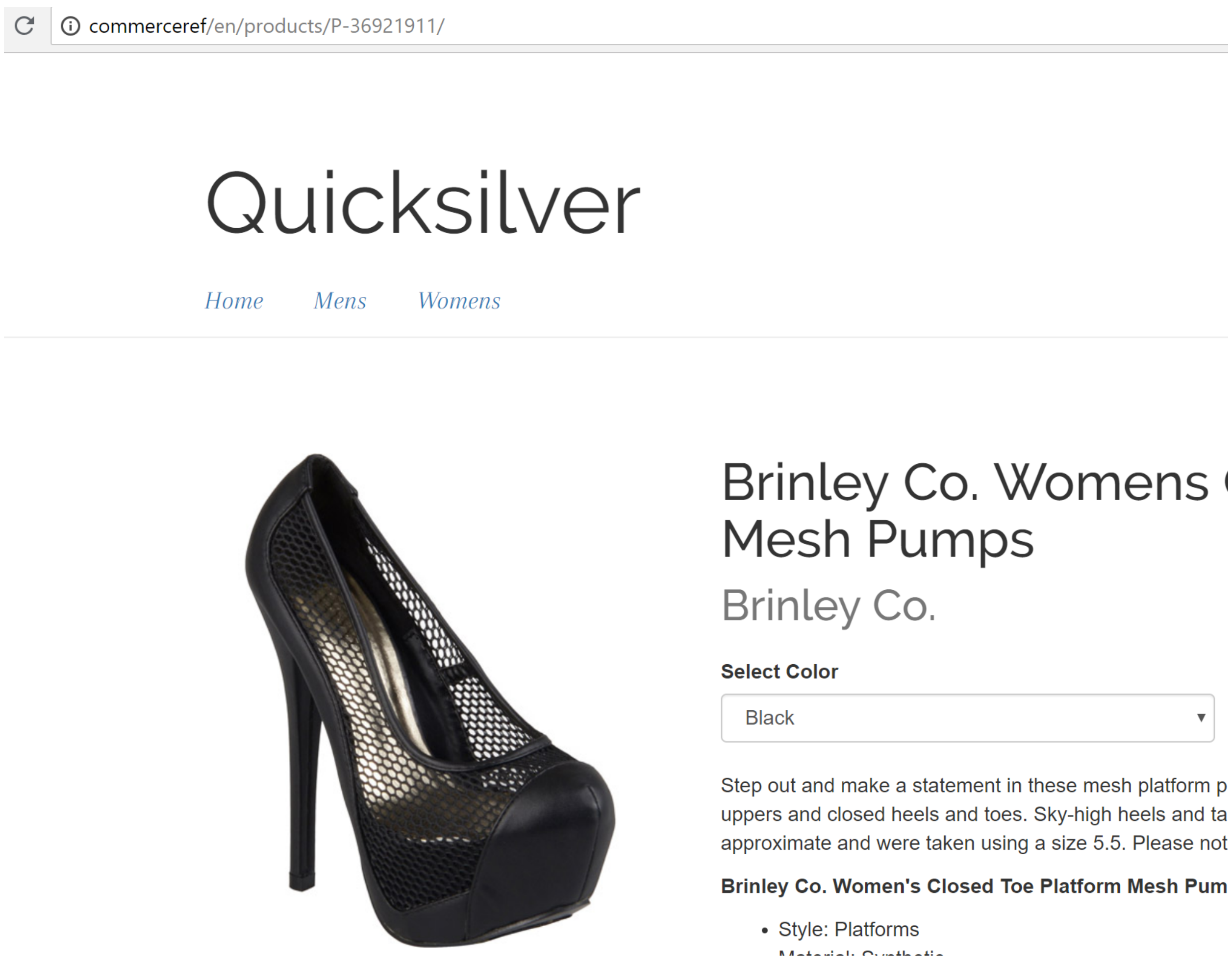
This will handle the outgoing Url, so when you call `UrlResolver.GetUrl(ContentReference)`, the url generated will look like this "http://commerceref/en/products/P-36921911/".

But when you open that link, it will return 404 because the router does not know how to handle the incoming urls yet, so from that link, it does not know which product to return. That's when `FindNextContentInSegmentPair` comes into play

```
protected override CatalogContentBase FindNextContentInSegmentPair(CatalogContentBas\
e catalogContent, SegmentPair segmentPair,
    SegmentContext segmentContext, CultureInfo cultureInfo)
{
    if (catalogContent.ContentType == CatalogContentType.Root)
    {
        var contentLink = _referenceConverter.GetContentLink(segmentPair.Next, Catal\
ogContentType.CatalogEntry);
        return _contentLoader.Get<ProductContent>(contentLink);
    }
    return base.FindNextContentInSegmentPair(catalogContent, segmentPair, segmentCon\
text, cultureInfo);
}
```

Because we start at the root, we know the next segment would be the product code. Simply use `ReferenceConverter` to get the `ContentReference` from the code, and then return the product from `IContentLoader.Get<ProductContent>`.

Here we are returning directly the product. But we can be more tolerant: If `TryGet` returns the product, return it, otherwise return null.



Brinley Co. Womens
Mesh Pumps

Brinley Co.

Step out and make a statement in these mesh platform p
uppers and closed heels and toes. Sky-high heels and ta
approximate and were taken using a size 5.5. Please not

Brinley Co. Women's Closed Toe Platform Mesh Pum

- Style: Platforms

And it works!

A few notes regarding this implementation:

- This abolishes the catalog structure, so "http://sitename/products/category/sub-category/product" will no longer work.
- We are still using the Catalog Root as the `commerceRootContent`, there for "/en/" still appears in the url. You know how to remove it.

Problem 1.7: Get products belong to a specific market

Level: Advanced

It's great if you have a comprehensive reporting system connected to your Episerver Commerce site. But if you don't, then there are time when you have to create the reports yourself. Let's consider one type of report that would be to find out which products belong to a specific market.

This would be fairly easy if you use the recursive approach we talked about earlier. Just get every entry and check for `MarketFilter`. If this list contains the market id you are looking for, then add it to our list.

Sounds simple, right? But if you want to have a few hundred thousands of products, this barely works effectively. Loading that much of products will be a waste of time and resource. As always, when the APIs can't provide an adequate performance, it's time to step up our game and do what is not (officially) supported.

To make it work, we need to need to understand about the underlying type of markets - dictionary types. Like it or not, you will have to work with dictionary types, sooner or later. They can be particularly useful when you need to check something - fast.

- 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' AND N\
amespace = 'Mediachase.Commerce.Catalog')
    SET @MetaDictionaryId = (SELECT MetaDictionaryId FROM MetaDictionary WHERE Value\
 = '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.

- Now let's go back to the original task we have. 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`.

```sql
    DECLARE @MetaFieldId INT
    DECLARE @MetaDictionaryId INT

    SET @MetaFieldId = (SELECT MetaFieldId FROM MetaField
    WHERE Name = '_ExcludedCatalogEntryMarkets' AND Namespace = 'Mediachase.Commerce\
.Catalog')
    SET @MetaDictionaryId = (SELECT MetaDictionaryId FROM MetaDictionary WHERE Value\
 = '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.

Problem 1.8: Move catalog items around

Level: Immediate

Catalog UI provides a very intuitive way to manage catalog items, including moving them around (which is not just “by accident” - Episerver invested a fairly big chunk amount of time for UX research, and then UI refinements to make managing catalog items really *works*). But sometimes it’s not enough. “Sometimes people deserve more.”. Sometimes people deserve to have their items moved programmatically.

There are two way to move a catalog item around: by using `IContentRepository.Move`, and by using `IRelationRepository.UpdateRelations`.

To use `IContentRepository.Move`, it is simple as this:

```
_contentRepo.Move(contentLink, destination, requiredSourceAccess, requiredDestinatio\
nAccess);
```

`requiredSourceAccess` and `requiredDestinationAccess` are the required access level of the user performing this action on the source content and the target content, respectively. It’s up to you to decide, moving a content is not technically deleting the old content and creating the clone of it in the new location, but rather, changing the relation between the content and old and new parent. There is no `AccessLevel` defined for such changes, so it’s really a matter of perspective here.

Or with `IRelationRepository`:

```
var relations = _relationRepository.GetParents<NodeRelation>(contentLink);
var relationToRemove = relations.FirstOrDefault(r=>r.Parent == oldParentLink);
relations.Remove(relationToRemove);
var relationToAdd = CreateNewRelation(contentLink, newParentLink);
relations.Add(relationToAdd);
_relationRepository.UpdateRelations(relations);
```

Now the tricky part is `CreateNewRelation`. There are two kinds of relations you can create: `NodeRelation` - between a node and a node, and `NodeEntryRelation` - between an entry and a node. Just remember the bigger entity is `Parent`, and the smaller entity is `Child`. For example, to create a `NodeEntryRelation`:

```
var relation = new NodeEntryRelation();
relation.Child = contentLink;
relation.Parent = newParentLink;
relation.SortOrder = 10;
```

Strictly speaking, `NodeEntryRelation` extends `NodeRelation`, so `NodeRelation` can be used to represent a relation between an entry and a node (and that still counts as a node relation, literally). However, to make thing easier to understand and follow, you should always use `NodeEntryRelation` to represent a node-entry relationship.

Now we have a new node-entry relation, just add it to the collection and save. Now this is a bit over complicated approach. You can just get the old relation and update `Parent` to point to the new parent link. But keep in mind that is not completely bullet proof. In case you move an entry which was a direct child of a catalog (i.e. it was not a child of any categories), then you will not find the existing `NodeEntryRelation`, and will have to create a new one.

There are something you should keep in mind: `IRelationRepository` works in a lower level, so it does not care about permission/access rights at all. It does not also fire content events (which you usually listen via `IContentEvents`). In fact for catalog content, `IContentRepository.Move` used to use `IRelationRepository` internally (That changed in Commerce 11). If such things are requirements for you, then you should stick with `IContentRepository.Move`.

If you are using Commerce 10 or earlier, then I have bad news for you. You would have to deal with `Source` and `Target` properties in `Relation`. I'm going to be honest with you: I can't remember which is which and always have to go back to read documentations to remind myself. That's also why you should upgrade to Commerce 11 or later, because the APIs have been vastly improved. `Source` and `Target` have been obsoleted, and are replaced by `Parent` and `Child`. Now it's much easier to understand which is which, because in a relation, the "bigger" entity is `Parent` and the smaller one is `Child`. That's all you have to remember.

Problem 1.8.2 Set the primary node

Level: Immediate

In previous recipe(s) we talked about the relation between nodes and entries. You will soon realize that an entry can belong to multiple nodes. And that is to solve a problem - how would you place a GoPro in your catalog? Under "camera" category or under "sport gear" category? Duplicating the entry to be in both categories is not the best solution, for obvious reasons. To solve it: well, just place it under "camera", then link it to "sport gear" - or the way around. Problem solved?

If you upgraded from previous version, the parent node with lowest `SortOrder` relation will be selected as the primary. That is not exactly how it should be used (we will get into that later). The primary node can also be set by moving the entries in Catalog UI. When you move an entry from a node to another node, the primary is updated to the new parent.

What if you want to set the primary node, programmatically?

This requires Commerce 11.2 at least. Previously you can add or remove a node-entry relation (via `NodeRelation`), but you can't set it explicitly to primary. This requires the new type `NodeEntryRelation` which was introduced in Commerce 11.2.

```
var relations = _relationRepository.GetParents<NodeEntryRelation>(entryLink);
var relation = relations.First(r => r.Parent == parentLink);
relation.IsPrimary = true;
_relationRepository.UpdateRelations(relations);
```

Basically you find the matching `NodeEntryRelation` and set the property `IsPrimary` to true, and then save it back.

You don't have to update the entire collection, you can save only the specific one `_relationRepository.UpdateRelations(new [] { relation });`

There are a few things to keep in mind:

- An exception will be thrown if you try to save two or more relations of an entry with `IsPrimary` is true. An entry can only truly belong to one node (or none at all - in that case it is linked to all the nodes, its true parent is the catalog it is in)
- A new set primary node will override the existing one, if any.

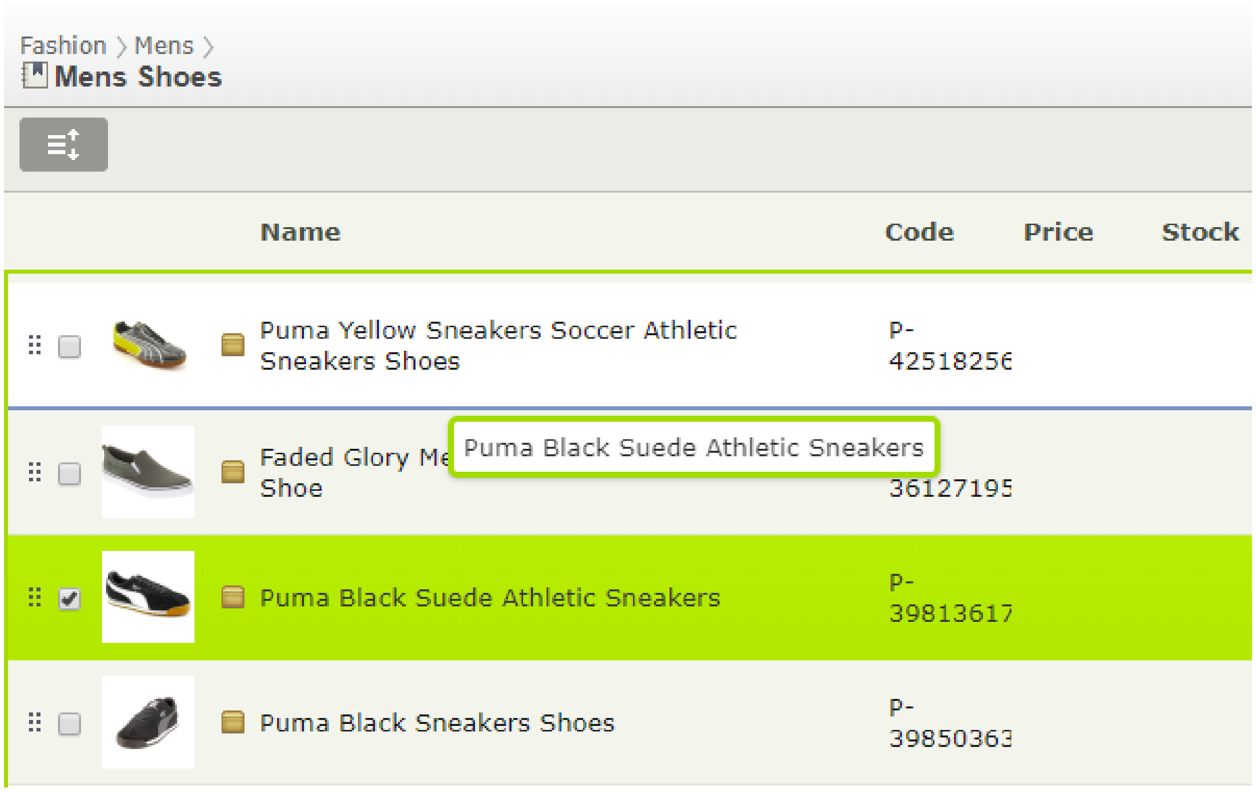
Problem 1.8.3: Adjust sort order

Level: Immediate

Even in the same category, the entries are not equal. For example if you have a "Phone" category, you would probably want the new shiny iPhone X, or Galaxy S9 to be on top of the category, for obvious reasons.

This is, however missing in Commerce 10.8 or earlier. The sort order was part of node-entry relation, but it was used incorrectly before, it was used to determined which node is the primary node: if an entry has multiple node-entry relations, then the node which has lowest sort order relation is the primary one.

Commerce 11 addressed this issue by introducing a new property in the `NodeRelation` - `IsPrimary` as we learned in previous recipe. By freeing `SortOrder` from that purpose, it can be used properly. That means you can now drag and drop an entry to adjust its sort order:



Fashion > Mens >
Mens Shoes

	Name	Code	Price	Stock
☐	Puma Yellow Sneakers Soccer Athletic Sneakers Shoes	P-42518256		
☐	Faded Glory Me Shoe	36127195		
✓	Puma Black Suede Athletic Sneakers	P-39813617		
☐	Puma Black Sneakers Shoes	P-39850363		

You need Sort Mode to drag and drop catalog items to change their Sort Order

Programmatically Sort Order can be set by using the APIs we discussed in previous recipes.

Problem 1.9: Build a breadcrumb

Level: Beginner

Breadcrumb is about the discoverability. When a customer visit a product page, they might want to check the parent category for similar products, and so on.

First let's talk about the definition of the models:

```
public class BreadCrumbViewModel
{
    public IList<CatalogContentModel> Models;
}

public class CatalogContentModel
{
    public string DisplayName { get; set; }
    public string Url { get; set; }
}
```

And the implementation would look like this

```
[HttpGet]
public ActionResult GenerateBreadCrumb(FashionProduct currentContent)
{
    CatalogContentBase content = currentContent;
    var models = new List<CatalogContentModel>();
    while (!(content is CatalogContent))
    {
        var parentContent = _contentLoader.Get<CatalogContentBase>(content.ParentLin\
k);
        var model = new CatalogContentModel()
        {
            DisplayName = parentContent is NodeContent ? ((NodeContent)parentContent\
).DisplayName : parentContent.Name,
            Url = UrlResolver.Current.GetUrl(parentContent)
        };
        models.Add(model);
        content = parentContent;
    }

    models.Reverse();
    return PartialView("BreadCrumb", new BreadCrumbViewModel() { Models = models });
}
```

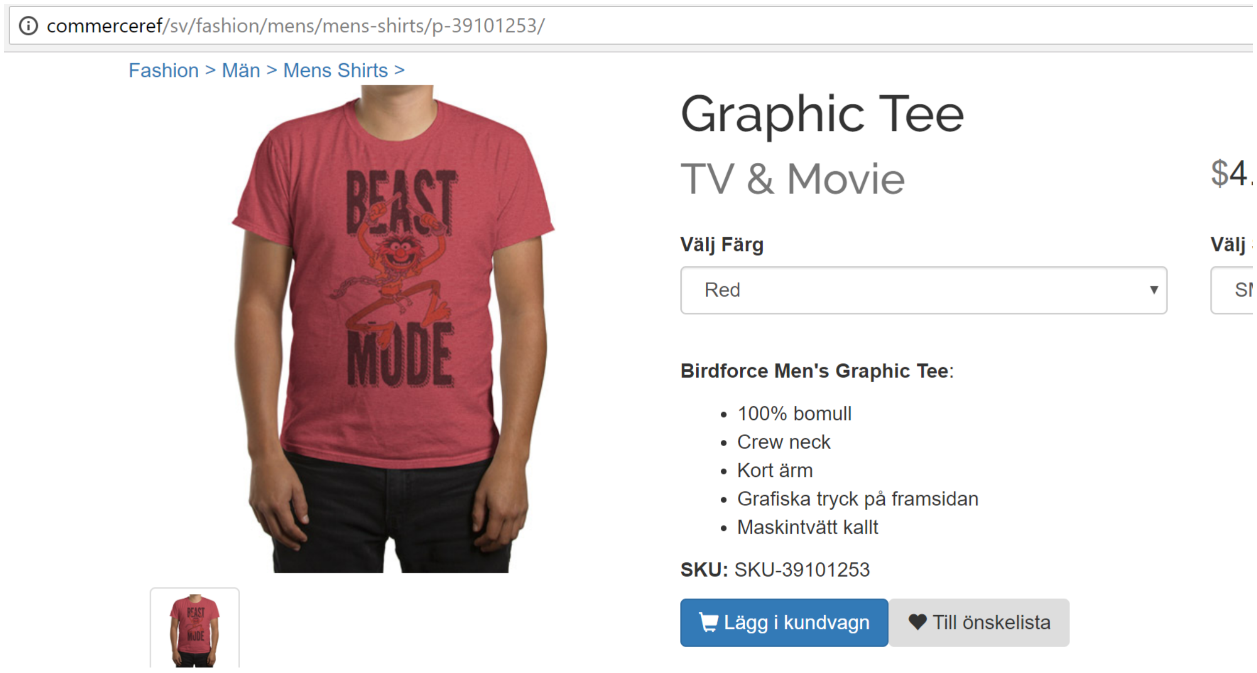
The idea is quite simple: we create a child `Action` in the product controller. The action takes the current content which is passed automatically (as long as it's named `currentContent`). From the current content, we get the parents until we reach the catalog. With each content, we create a model for it, consist of `DisplayName` and `Url`.

The `Reverse()` before returning the partial view is important, because the list was in reversed order (child first), while we would want the breadcrumb to be parent first.

And then a simple partial view would be enough to render the breadcrumb:

```
@model EPiServer.Reference.Commerce.Site.Features.Product.Controllers.BreadCrumbView\
Model

@foreach (var item in Model.Models)
{
    <a href="@item.Url">@item.DisplayName ></a>
}
```



commerceref/sv/fashion/mens/mens-shirts/p-39101253/

Fashion > Män > Mens Shirts >

Graphic Tee

TV & Movie

$4

Välj Färg

Red

Välj

Sl

Birdforce Men's Graphic Tee:

- 100% bomull
- Crew neck
- Kort ärm
- Grafiska tryck på framsidan
- Maskintvätt kallt

SKU: SKU-39101253

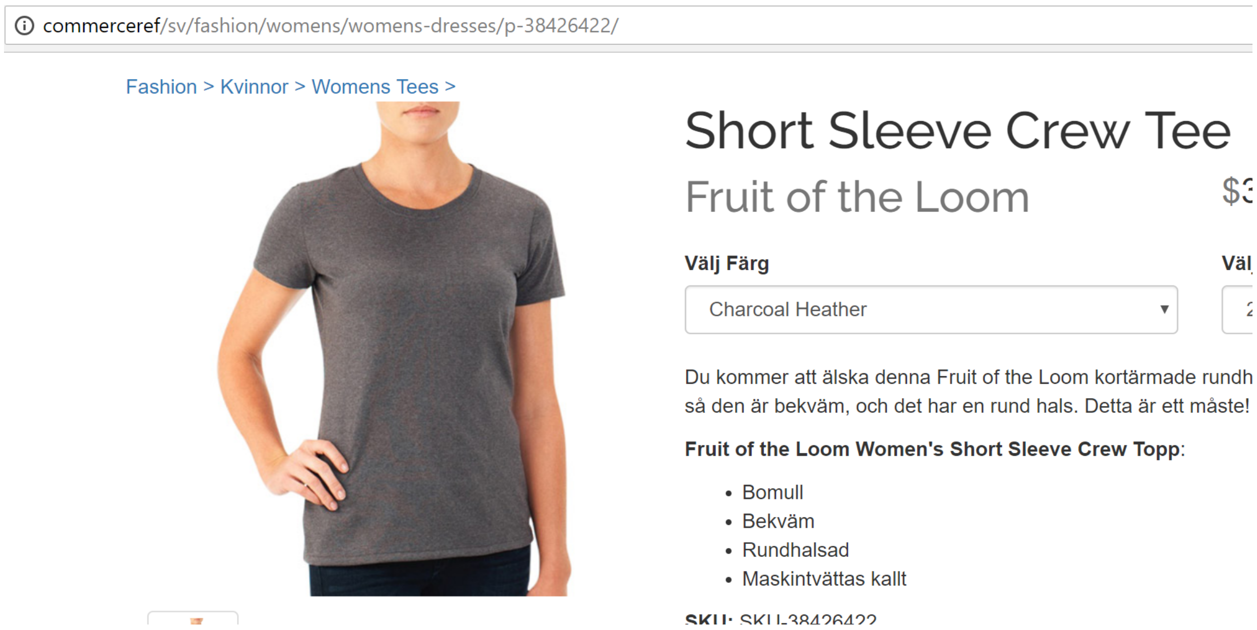
At least it works!

Well this is with minimal UX design. When you know the principles, you can of course make it as pretty as you want.

Problem 1.9.2: Build a breadcrumb: multiple categories scenario

Level: Immediate

In previous recipe we talked about how to build a breadcrumb. It’s not complicated, but it works quite well, but that’s with an issue: it does not always work correctly. If a product is linked to another category, when customers open it through that route, you would want to breadcrumb to reflect that. However, it is always rendered with the primary node. That is probably not something you want, so let’s fix it.


commerceref/sv/fashion/womens/womens-dresses/p-38426422/

Fashion > Kvinnor > Womens Tees >

Short Sleeve Crew Tee

Fruit of the Loom

Välj Färg

Charcoal Heather

Du kommer att älska denna Fruit of the Loom kortärmade rundh
så den är bekväm, och det har en rund hals. Detta är ett måste!

Fruit of the Loom Women's Short Sleeve Crew Topp:

- Bomull
- Bekväm
- Rundhalsad
- Maskintvättas kallt

SKU: SKU-38426422

Something is not right

To solve that problem we need to know about the context - we can't rely on the data to know which category is being used, because each time it'll be different. How can we do that? By using the router - the router already has access to the correct contents, so we can just store those information in some kind of cache and get them later for our breadcrumb.

Let's start adding an implementation for our router:

```
public class BreadCrumbCatalogPartialRouter : HierarchicalCatalogPartialRouter
{
    public BreadCrumbCatalogPartialRouter(Func<ContentReference> routeStartingPoint,
        CatalogContentBase commerceRoot, bool enableOutgoingSeoUri)
        : base(routeStartingPoint, commerceRoot, enableOutgoingSeoUri)
    {
    }
}
```

The tricky part is to know which method to override - and I will save you some time by letting you know, it's `FindNextContentInSegmentPair`, and here's what we are going to do:

```
protected override CatalogContentBase FindNextContentInSegmentPair(CatalogContentBas\
e catalogContent, SegmentPair segmentPair, SegmentContext segmentContext, CultureInf\
o cultureInfo)
{
    var foundContent = base.FindNextContentInSegmentPair(catalogContent, segmentPair\
, segmentContext, cultureInfo);
    var httpRequest = HttpContext.Current.GetRequestContext().HttpContext;

    var savedModel = httpRequest.Items["RoutedContents"] as BreadCrumbViewModel ?? n\
ew BreadCrumbViewModel();

    var model = new CatalogContentModel()
    {
        DisplayName = foundContent is NodeContent ? ((NodeContent)foundContent).Disp\
layName : foundContent.Name,
        Url = UrlResolver.Current.GetUrl(foundContent)
    };

    savedModel.Models.Add(model);
    httpRequest.Items["RoutedContents"] = savedModel;

    return foundContent;
}
```

`FindNextContentInSegmentPair` is responsible for loading the next content in the url, so we already have that. So we just need to use the `HttpRequestBase.Items` to store our model. If it is there, great, use it, otherwise create a new one. Note, that would mean a small change in `BreadCrumbViewModel`:

```
public class BreadCrumbViewModel
{
    public BreadCrumbViewModel()
    {
        Models = new List<CatalogContentModel>();
    }
    public IList<CatalogContentModel> Models;
}
```

To make sure `Model` is always properly initialized. When we add the new model into the list, we save it back to the `Items` so it can be reused next time. When the content is fully routed, we also have enough information for our breadcrumb.

With our new partial router, we need to register it. We already know how, but this time we are not demonstrating how to remove the catalog name from the url, so it would look like this:

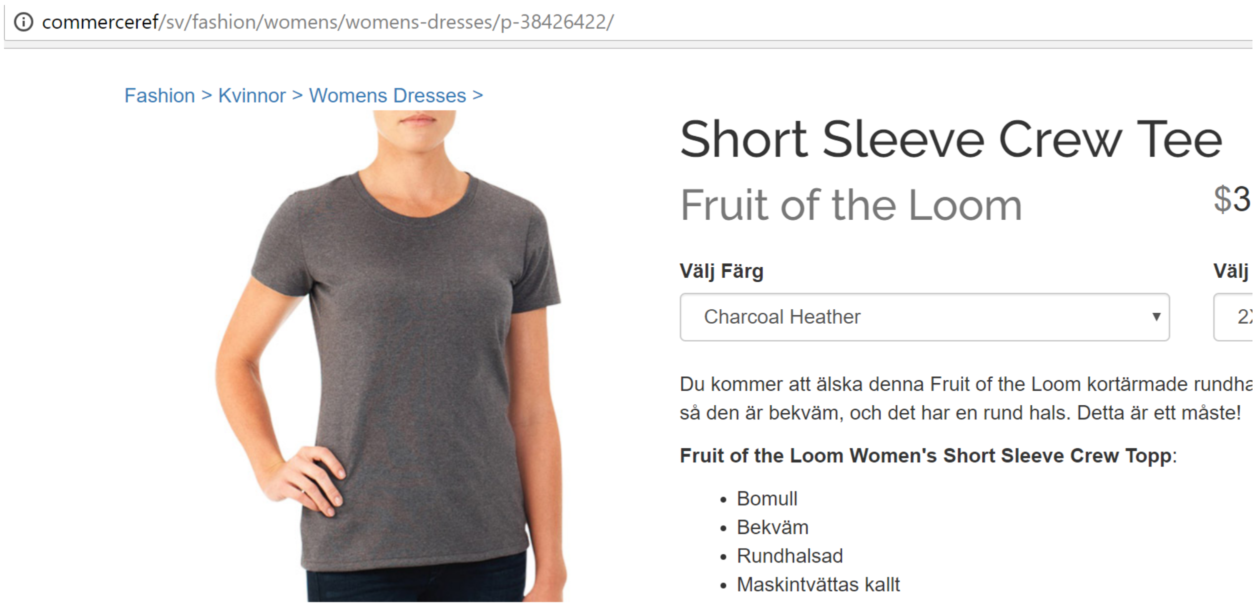
```
public void Initialize(InitializationEngine context)
{
    var referenceConverter = context.Locate.Advanced.GetInstance<ReferenceConverter>\
();
    var contentLoader = context.Locate.Advanced.GetInstance<IContentLoader>();
    var commerceRootContent = contentLoader.Get<RootContent>(referenceConverter.GetR\
ootLink());
    Func<ContentReference> startingPoint = () => ContentReference.IsNullOrEmpty(Site\
Definition.Current.StartPage) ?
        SiteDefinition.Current.RootPage :
        SiteDefinition.Current.StartPage;
    RouteTable.Routes.RegisterPartialRouter(new BreadCrumbCatalogPartialRouter(start\
ingPoint, commerceRootContent, false));
    ...
```

The final piece is to update our little `Action`, so it does not have to load the contents itself, but just from the `HttpContext`:

```
[HttpGet]
public ActionResult GenerateBreadCrumb(FashionProduct currentContent)
{
    var viewModel = HttpContext.Items["RoutedContents"] as BreadCrumbViewModel ?? ne\
w BreadCrumbViewModel();
    return PartialView("BreadCrumb", viewModel);
}
```

It’s worth noting that we don’t have to `Reserve` anything here. The reason was `FindNextContentInSegmentPair` already works from left to right - from the top level content to the leaf-most level, so our `ViewModel` is already in correct order.

And now it’s working correctly for a linked category, yay!


commerceref/sv/fashion/womens/womens-dresses/p-38426422/

Fashion > Kvinnor > Womens Dresses >

Short Sleeve Crew Tee

Fruit of the Loom $3

Välj Färg

Charcoal Heather

Välj

2

Du kommer att älska denna Fruit of the Loom kortärmade rundha
så den är bekväm, och det har en rund hals. Detta är ett måste!

Fruit of the Loom Women's Short Sleeve Crew Topp:

- Bomull
- Bekväm
- Rundhalsad
- Maskintvättas kallt

The breadcrumb aligns with the url