# Umbraco Training

*Umbraco Application Integration*

|  |  |
| --- | --- |
| Your Name: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Date Attended: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Trainer: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

# Agenda

We have a number of topics to discuss when considering application integration:

## Solution Overview

### Shop project overview

### Umbraco Architecture overview

### Implementation Choices

## Getting shop data onto Umbraco pages

### Display albums from the store Integrate with Macros

## Hijack Umbraco pages by Route

### Display Product details with an Umbraco template

## Finding content

### 301 redirect to new Umbraco urls

### Members and customers Using Umbraco Membership with Store Orders Backoffice integration

### Product Picker in AngularJS

### Custom Examine Indexing

Indexing external content with Examine

### 

# Solution setup

Your course teacher will provide you with a pre-made Umbraco solution, using Umbraco version 7, running on SQL CE and IIS Express.

All you need to do is run the solution to get started.

## Environment

* SQL CE 4
* IIS Express
* Visual Studio 2013 or 2015
* Umbraco 7

## Bundled items:

* Document Types
* Basic content structure
* Templates
* JavaScript and CSS

## Training Website user credentials

Login: [admin@](mailto:admin@umbraco.training.co.uk)admin.com  
Password: 12345678

# 

# 

# Solution overview

Our entire day will be spent working on 3 projects in a combined Visual Studio Solution, as we progress through the exercises we will add features from the store part of the website into the Umbraco website.

Our starting point is a basic ecommerce application, seperated into 2 projects, an MVC project and a project containing the services, classes and interfaces used. We will reuse these classes and interfaces as we migrate the UI portion of the store to our Umbraco website.

At the end of the day, we will have covered the major integration points in Umbraco, giving you a solid toolbox of options and best practices in your day to day work with Umbraco and external data and businesslogic.

## The 3 projects

### Training.MVC.OnlineStore.UI

The original store application UI which we will pick features from to integrate into our Umbraco website. It’s a standard MVC application which implements standard shop functionality like products, groups and a cart.

### Training.Services.OnlineStore

Core classes, interfaces and services for our store, these are currently used by the OnlineStore.UI project, and will become part of the integration into the Umbraco website.

### Training.Umbraco.WebSite.UI

A standard Umbraco 7.x website, installed via Nuget, running on SQL CE. The website already has the UI parts sorted out, so comes with pre-made content, a couple of views, css and javascript files to handle a basic website.

# 

# 

# Introduction to the store app

Follow along as your instructor goes through the store application. The store contains a set of fairly standard commerce features, this introduction will go through which features will be migrated and how they fit into the Umbraco website.

## The features

### Browse products

Products will be pulled from the current webshop service layer, using the already made Viewmodels, Umbraco will handle presentation

### Sorting and filtering

UI of the filtering and sorting will be moved to Umbraco and the webshop services-layer will keep handling the sorting logic and return the collection of products

### Add to cart

Add to cart functionality will be moved into a partial view in Umbraco from a partial view in the webshop project. The implementation will be nearly identical between the 2 sites.

### Register

Account creation and member login functionality will be moved to partial views in Umbraco. The code remains virtually unchanged.

### Shopping Cart

Shopping cart functionality will also be moved to partial views in Umbraco. As with much of the other functionality, the code remains mostly the same.

### Checkout

The checkout functionality will also be moved. You may notice some checkout methods are not fully implemented. This is intentional as specific checkout requirements will vary widely between different projects.

### Admin Area

The current webshop uses an MVC convention known as Areas for the administration side of the shop. We will convert some of the admin functionality to Umbraco Property Editors using AngularJS so the products can be selected in the Umbraco backoffice when editing content.

# Umbraco Architecture 101

Before we start integrating the store parts into the Umbraco site, let's very briefly go over the different parts of the Umbraco solution which will affect the way we work.

### The content cache

The In-memory representation of the entire content and media structure of our website. As it is cached, it allows us to do very quick lookups of data to display. All these lookups are done primarily through 2 methods:

* Model.Content model in razor views, representing the current cached page being displayed
* UmbracoHelper, the API allowing us to query the content cache from a controller

### Umbraco database schema

As Umbraco has a very generic content model, based on configured ContentTypes - it also has a very generic database schema, this means you have no efficient way to query for data directly in the Database, and you should never attempt to do so. You should either use the appropriate Service API to query for data, or use the cached published Content API mentioned above.

### Service APIs

When you use any of the Umbraco specific base classes like UmbracoApiController or SurfaceController, you get access to a list of Service APIs under the Services property. All these APIs store data directly in the database, and ensure internal caching is updated. However, you should never use these Services on the website for presenting content. They are simply not made for frequent usage, they are meant for management tasks performed by the backoffice or private sections of a website.

# 

# Implementation Choices

Before we begin any integration we want to consider which parts of the existing MVC webshop make sense to move into Umbraco and which parts make sense to leave in place. Some of the obvious parts to move to Umbraco are those that will be used by a content editor, such as the product picker. Parts that make sense to leave as part of the existing webshop are the existing services, interfaces, and models.

Other considerations include the following, but of course there are more things to consider and what is important will vary from project to project:

### Should our data be imported as Umbraco content ?

This really depends on where the data will be managed. If the data is content and will be managed like content it can make sense to import it as Umbraco content. On the other hand, if the data is very specific in its usage and management, as with products, it can make sense to leave it as its native format which will, generally, already be optimized for the specific type of data. (If data is imported regularly from an external source, daily xml feed, etc be very careful about importing and syncing with Umbraco nodes, be aware of the number of versions being created by the import each day of each node)

### Should we use Umbraco membership ?

Using Umbraco membership has advantages in that you can manage members from the Umbraco backoffice and take advantage of the existing Umbraco support for members. Leaving membership as part of the MVC webshop may have the advantage of already existing relationships between members, accounts, and other webshop functionality such as the cart and checkout.

### One database or more than one ?

In many cases it can make sense to create custom tables in the existing Umbraco database. This has the advantage of using the existing Umbraco database context for access to the data and can simplify the deployment process. On the other hand, keeping existing data in a database already optimized for that specific data has the obvious advantage of simplicity and, potentially, performance.

For our integration we will leave the existing MVC webshop intact along with the product database and concentrate on integrating the already existing parts into our Umbraco site. We will create an Umbraco Property Editor to allow our Content Editors to select Products for display and also create some dynamic views (e.g. Macros) to allow products to be displayed in various ways.

Take a few minutes to explore the solution and get an understanding of the 3 projects. The Umbraco project should look familiar to you and the MVC webshop is also using a standard MVC structure. Note there is both an Umbraco database (umbraco.sdf) and a webshop database (OnlineStore.sdf).

# 

# 

# Exercise 1: Getting started

The first thing to do in an integration project like this is to understand what we can reuse from the old project. Then establish an initial connection between the current store and the Umbraco website, so we have something to build on top of.

### In this exercise we will:

* Look at the classes and views we will reuse from the current store
* Look at the base classes we’ve added to the Umbraco project to make integration easier
* Make our first connection between Umbraco and the store

## ViewModels and Views Reuse

The current store uses the concept of ViewModel classes, so for simplicity, we will continue using this convention. For a start, we’ve copied over all the relevant classes from the store’s /ViewModels folder into a /ViewModels folder in the Umbraco site.

These classes will be the backbone in the way we will display data from the store database.

With this integration we will be reusing the store Html, and so we have also copied over the Views and PartialViews, from the store application into the Umbraco site.

## Automapper Configuration

The current store also uses AutoMapper to map database models to ViewModels - we’ve copied this configuration into the Umbraco site as well - available in /App\_Start/AutoMapperHandler.cs

**Note** that we have copied some code for the MediaHelper from the store /helpers folder to the Umbraco /helpers folder. The only change we made to the code was to update the namespace to be Training.Umbraco.WebSite.UI.Helpers

## Adding Umbraco baseclasses

As we start integrating store services into Umbraco - we want to avoid wiring up the same code again and again, so to make our future tasks easier, 3 base classes have already been added to the Umbraco project:

* StoreSurfaceController - extending SurfaceController
* StoreTemplatePage - extending UmbracoTemplatePage
* StoreRenderMvcController - extending RenderMvcController

These three classes all add access to the Stores services like so:

public class StoreTemplatePage : UmbracoTemplatePage

{

protected readonly IProductService ProductService;

protected readonly ICategoryService CategoryService;

public StoreTemplatePage() : base(){

ProductService = new PetaPocoProductService("onlineStoreDb");

CategoryService = new PetaPocoCategoryService("onlineStoreDb");

}

}

So, it simply exposes IProductService and ICategoryService and makes the connection to the database one time, so we don’t need to do this again each time we want to use the store services.

We’re not going to cover dependency injection as part of this course.

### What is PetaPoco ?

PetaPoco is a tiny, fast, single-file micro-ORM for .NET (http://www.toptensoftware.com/petapoco/) and is used by Umbraco as the data layer to connect to the Umbraco DB, it works with POCO classes and SQL statements, we will be using it to access the Store database - you could use your favourite data access ORM here instead.

## Making the first connection

So with the basic classes copied to Umbraco, we are ready to make the first small step into our integration, which is simply to list our products as a list on the frontpage of our Umbraco website.

We will keep it simple by replacing the default UmbracoTemplatePage classes with our own StoreTemplatePage and read directly from the ProductService.

Open the /Views/Homepage.cshtml file, change the page type and make a simple call to the service like so:

@inherits StoreTemplatePage

@{

Layout = "Layout.cshtml";

}

<ul>

@foreach (var product in ProductService.GetTopRated(10))

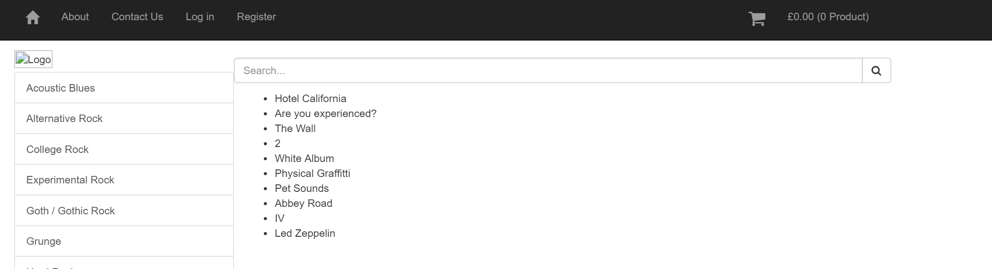
{

<li>@product.Name</li>

}

</ul>

The point here is to demonstrate we can start displaying data from the MVC store database, it is possible! But we’re not necessarily using the correct pattern - we will get to that part in the next chapter! - if the code above works, you will see something like this:



Connection established! - now it’s time to do things the right way!

# 

# 

# Exercise 2: Integrating albums on the homepage

Now that we have a connection to the store data, lets get a proper display of Albums of the homepage in place - to replicate the content of the current stores homepage.

If you’ve worked with MVC or have attended the Umbraco MVC Masterclass the implementation part of this will look familiar. If you haven’t done either of those you are about to use a simple integration technique in Umbraco. Using the Umbraco standard SurfaceController we will call a ChildAction method in a Controller to render a PartialView using a Model populated with products.

This is a very common MVC pattern and one you can use in many different scenarios. We’ll add this to the homepage template that already exists.

### In this exercise we will:

* Create a HomePageController
* Add a FeaturedProducts action to the controller
* Make the controller query the ProductService for top albums
* Return a partial view of these albums

## Create a controller

Add a new file to /Controllers - call it HomePageController.cs and make it inherit from the baseclass: StoreSurfaceController, so the class looks like this:

public class HomePageController : StoreSurfaceController

{

}

## FeaturedProducts action

Now, we need an action to return album data, our base class provides access to ProductService so we can easily query data and return a partial view.

**Notice:** the partial view used here (views/shared/\_featureproducts.cshtml) is imported from the original store, so we are reusing html and implementation logic.

**Notice:** AutoMapper is used to map from the database model (Product) to the ViewModel (ProductPreviewViewModel) - this ensures our view will only get data we intend to display on the page, instead of the entire domain model - this is a common MVC pattern to follow.

Below is the complete FeaturedProducts method called ProductService, mapping to ProductPreviewModel and returning it with the \_FeaturedProducts view, notice the ChildActionOnly attribute which ensures this method is not routed to directly but only available as part of another page request:

[ChildActionOnly]

public ActionResult FeaturedProducts(int count = 3)

{

var featuredProducts = ProductService.GetFeatured(count);

var items = Mapper.Map<IEnumerable<ProductPreviewViewModel>>(featuredProducts);

return PartialView("\_FeaturedProducts", items);  
}

Finally, we need to make the frontpage show the featured albums, so go back to the /Views/Homepage.cshtml file and include a call to the action like so:

@inherits StoreTemplatePage

@{

Layout = "Layout.cshtml";

}

<div class="row">

<div class="col-md-12">

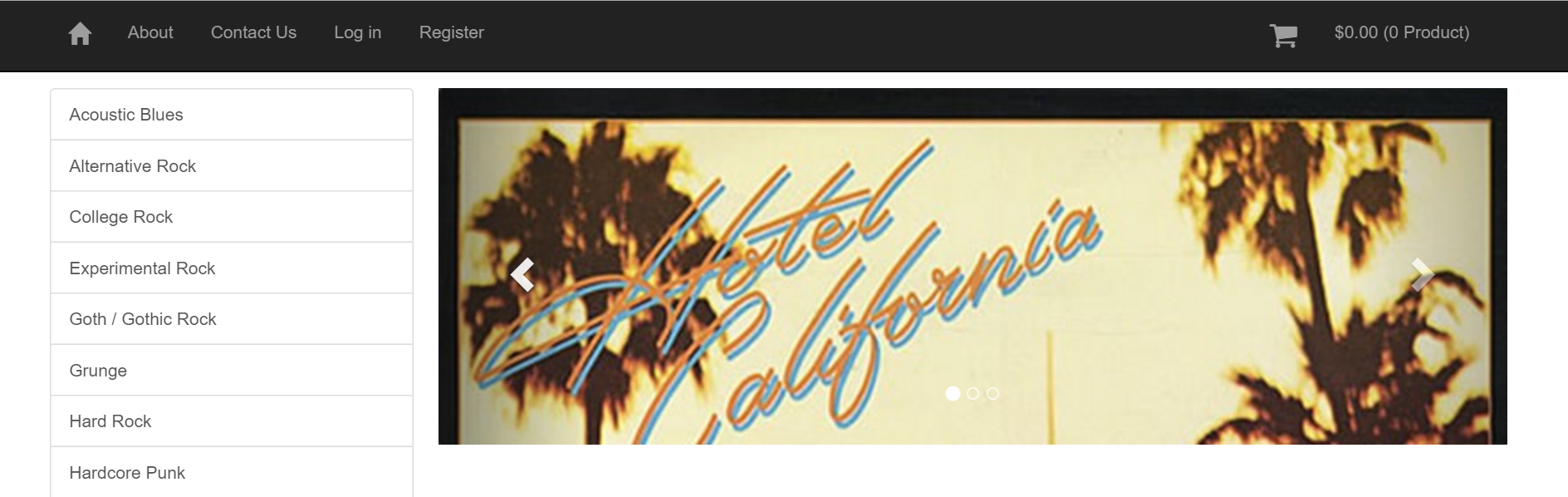
<div class="row carousel-holder">

**@Html.Action("FeaturedProducts", "HomePage")**

</div>

. . .

This calls the action FeaturedProducts on the HomePageController and will output the result of that action - in this case it will render the partial view \_FeaturedProducts.cshtml.



# Exercise 3: Top Selling Albums in a Macro

As we continue to move store homepage functionality to Umbraco, we will also start using more of Umbraco’s core features. For the next step of getting top selling albums on the front page, we will wrap the integration in a Macro, this gives us greater control, encapsulation and caching.

We already have a Controller and action setup to fetch our albums, so it is a small task to get the code running inside a macro, and we can then focus on the benefits the macro gives us.

### In this exercise we will

* Create a macro to list albums
* Add store code to the macro to fetch album data
* Add a macro parameter to the macro to filter albums
* Add caching to the macro

First of all we need an Action on our HomePageController to return top selling albums, so we will add an action to the HomePageController.cs file like so:

[ChildActionOnly]

public ActionResult TopSelling(int count = 6)

{

var topSelling = ProductService.GetTopSelling(count);

var items = Mapper.Map<IEnumerable<ProductPreviewViewModel>>(topSelling);

items = items.OrderByDescending(x => x.Rates);

return PartialView("\_HighlightProducts", items);

}

**Notice**: we follow the same pattern as before, fetch products, map them to ViewModel classes and return a partial view - this implementation is nearly identical as the logic inside the current store - simply in an Umbraco context.

Now, go to the developer section in the Umbraco backoffice and create a new macro called “Top Selling”. Associate your new macro with the “\_TopSelling.cshtml” MVC Partial view. On the parameters tab of the macro editor, also add a parameter with:

* Alias set to “numberOfItems”
* Name set to “Number of items”
* Type set to “ Numeric”

Now open the /Views/macropartials/\_TopSelling.cshtml file in the Umbraco project in Visual Studio and modify it so it calls the TopSelling action on the HomePageController like so:

@using Umbraco.Web.Models

@inherits Umbraco.Web.Macros.PartialViewMacroPage

@Html.Action("TopSelling", "HomePage",

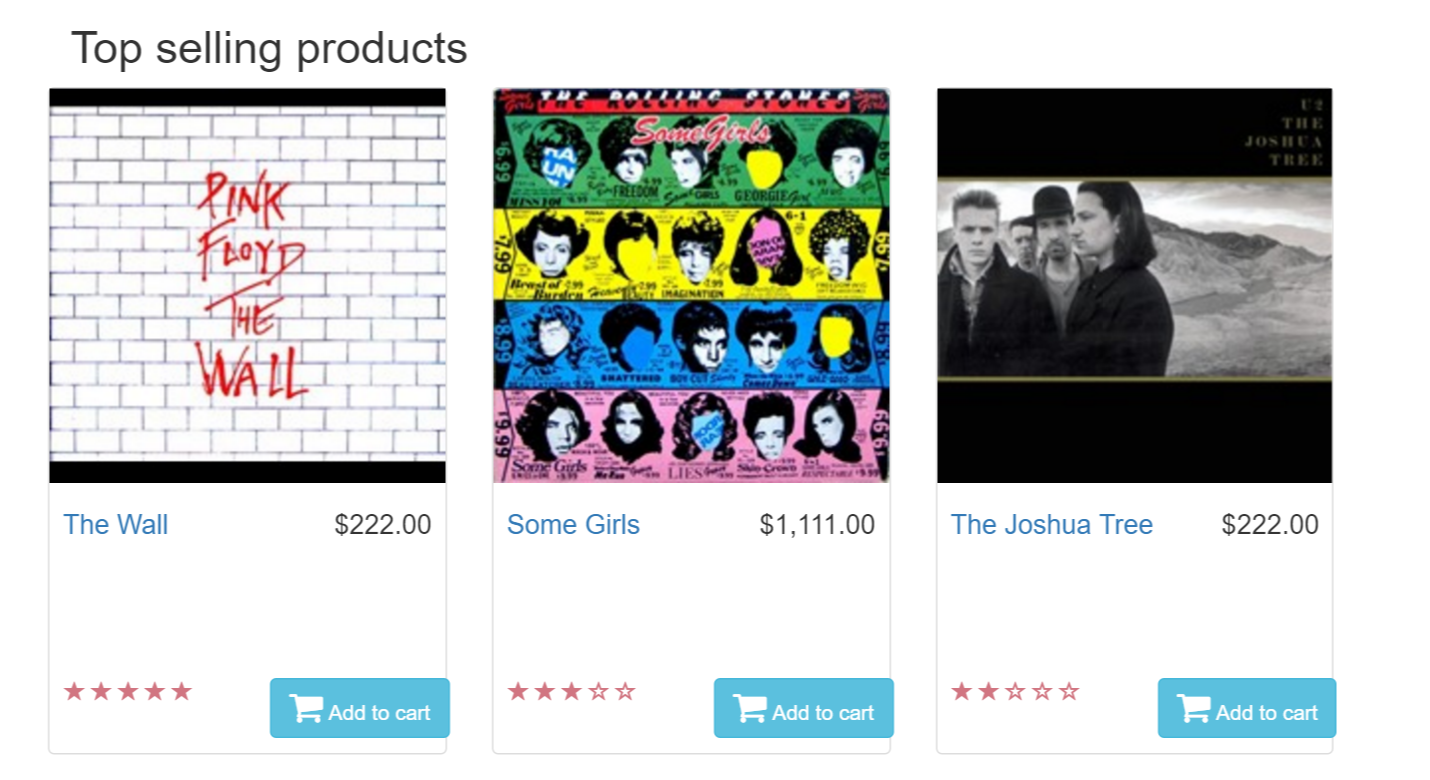
new { count = Model.GetParameterValue("numberOfItems", 3)}

)

Finally, go to the homepage template and insert the macro, you can use the insert macro UI and get a form to enter number of items to show, or you can write it in Visual Studio as the following snippet:

@Umbraco.RenderMacro("TopSelling", new {numberOfItems="3"})

Refreshing the frontpage will give us the following:



# Exercise 4: Album detail page on a custom route

Now that we can list albums using the Umbraco way, we want these albums to link to a full album details page. As we do not want to have duplicate content between the store and Umbraco, we will need to add a custom route to Umbraco, to create nice Urls linking to pages using Umbraco templating.

### In this exercise we will

* Move the product ViewModels into the Umbraco project
* Create an Album details template
* Use the ProductViewModel to display album data
* Add custom Umbraco routing to use Album data as the model on a page
* Create a convention for nice Album Urls

**Notice**: *We have partially taken care of the first bullet point by copying the files necessary since the Umbraco site needs these ViewModels to build. With that said, it is important that you go through each one and make a note of what has been changed when migrating the ViewModels.*

First, we have copied the product ViewModels from the store project to ViewModels in the Umbraco site - the idea is to migrate this functionality rather than to reuse. The product ViewModels include:

* ProductViewModel.cs
* ProductPreviewViewModel.cs
* ProductBasicViewModel.cs

We will need to make the ProductViewModel inherit from Umbraco’s RenderModel so we can use it in our Umbraco Route, however in doing so, we will break the inheritance chain in the MVC app’s ProductViewModels, we cannot make the ProductViewModel inherit from both RenderModel AND ProductPreviewViewModel.

So in this instance we are going to merge the base properties of the ProductViewModels, into a single ProductViewModel but if we were continuing with the project this might be something we refactor later, when we integrate or move other functionality across.

We have also changed the namespace to match our Umbraco project’s namespace Training.Umbraco.WebSite.UI.ViewModels. Since we want control over our model but also still need access to Umbraco Content, we need to inherit from RenderModel. Our complete ProductViewModel will look as below:

using Umbraco.Core.Models;

using Umbraco.Web.Models;

namespace Training.Umbraco.WebSite.UI.ViewModels

{

public class ProductViewModel : RenderModel

{

public ProductViewModel(IPublishedContent content) : base(content)

{

}  
 //Comes from ProductBasicViewModel

public int Id { get; set; }

public string Artist { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }  
  
 //Comes from ProductViewModel

public string Description { get; set; }

public string CategoryName { get; set; }  
  
 //Comes from ProductPreviwViewModel

public string Summary { get; set; }

public int Rates { get; set; }  
  
 //We added this, as Umbraco supports image cropping and  
 //only needs a single image path  
 public string ImageUrl { get; set; }

}

}

Next we have copied the /Views/Product/details.cshtml file from the store to the Umbraco project’s /Views/Product folder. Now we need to replace the @model declaration at the top, to an @inherits declaration of UmbracoViewPage<ProductViewModel>. This will also require us to add a namespace: Training.Umbraco.WebSite.UI.ViewModels - this gives us a strong model to use on our product page

This is now using the ViewModel in our Umbraco project rather than the store ViewModel, and because we have used UmbracoViewPage<T> here we also have access to all those useful Umbraco helpers in our View.

Then we create a ProductController and inherit from the already added StoreRenderMvcController.

namespace Training.Umbraco.WebSite.UI.Controllers

{

public class ProductController : StoreRenderMvcController

{

}

Now we’ll create a Details action that will fetch a specific product from the ProductService:

public ActionResult Details(RenderModel model, int id)

{

}

**Notice**: How the Details action has a RenderModel as a parameter, this RenderModel represents the content which Umbraco matches to the requested Url. This is required, since this controller action is responsible for returning the entire model for the page - we will get to how this RenderModel is found later.

To complete the Details action, let’s add the code that calls this ProductService and returns a view:

public ActionResult Details(RenderModel model, int id)

{

if (model == null || model.Content == null || id < 0)

{  
 //currently never fires as we always return a model

return HttpNotFound();

}

var product = ProductService.GetById(id);

var vm = new ProductViewModel(model.Content);

Mapper.Map(product, vm);

return View(vm);

}

**Notice:** we fetch a Product from the ProductService, and we then also create a new ProductViewModel based on model.Content, and finally we map the properties from one to the other with AutoMapper.

This is the complete controller which return a ProductViewModel instead of the normal IPublishedContent, only thing we need to do now is setup the routing.

### Routing

A note on routing here. Since we are integrating an existing store that already has public Urls, the best-practice is to keep these Urls unchanged. This allows any existing links to a specific Url to still work, preserves any SEO value, and will help make the integration more seamless.

This is done by registering a route in the OnApplicationStarted event in the class we’ve added in /App\_Start/UmbracoApplicationEventHandler.cs.

This allows us to specify which Controller and Action will be used for a route that matches the defined pattern.

For our product details Url we added the route to the OnApplicationStarted event handler like:

RouteTable.Routes.MapUmbracoRoute(

"Product details",

"Product/details/{id}",

new

{

controller = "Product",

action = "Details",

id = UrlParameter.Optional

},   
new UmbracoVirtualNodeByIdRouteHandler(1058) );

If you’ve worked with MVC routes before this looks familiar but we do have one addition. We use the UmbracoVirtualNodeByIdRouteHandler to instantiate the root node of our site so that any products returned will be represented as children of the site root. That’s a very Umbraco approach, but enables us to use the existing Umbraco Views/Templates which may contain logic for working with IPublishedContent with our external data.

Try adding <h1>@Model.Content.Name</h1> to your details page, you can see the Name of the underlying IPublishedContentfor this route - eg. we are passing the details of the homepage.

## 

## 

## Go further

Now that you know how we convert a store controller to a Umbraco-based controller, and setup correct routing for it, continue to work on porting the CategoryController.cs from the store project to the Umbraco project.

**Steps to complete**

* Create a CategoryController and inherit from the StoreRenderMvcController
* Copy over actions from the store project
* Wire up routing to match /category/{id}

When you copy over the actions from the store project notice that the models requires renaming, the below code is the modified action that will work with our new ProductViewModel class:

public ActionResult ProductsByCategory(RenderModel model, int id, int page, ProductSortBy sortBy = ProductSortBy.RateHighest)

{

int totalItems;

var category = this.CategoryService.GetById(id);

var filter = new ProductSearchFilter() { CategoryId = id, SortBy = sortBy };

var products = this.ProductService.GetAll(filter, out totalItems, page);

var vm = new CategoryViewModel(model.Content);

Mapper.Map(category, vm);

var pager = new Pager<ProductPreviewViewModel>(  
 Mapper.Map<IEnumerable<ProductPreviewViewModel>>(products),   
 totalItems,   
 page, urlFormat: string.Format("/category/{0}/{{0}}",  
 model.Content.Id));

vm.Products = new ListOfProductsPreviewViewModel()

{

List = pager, ListTitle = model.Content.Name

};

return View(vm);  
}

When completed, the category navigation on the homepage will be fully functional

## Migrating the shopping cart controller

The store project contains a controller which handles all the interaction with the cart, we need this migrated to the Umbraco site for the cart to function. However, there are not very many difference in the 2 controllers.

Compare the ShoppingCartController classes between the store MVC app and the Umbraco Site. We’ve migrated the ShoppingCart functionality by making the new ShoppingCartController in Umbraco inherit from our common StoreSurfaceController.

Then we’ve copied over the Actions from the MVC controller, notice with the common use of ViewModels between the two projects, a lot of the code is identical. The main difference is in redirecting to another action, eg for AddToCart:

return RedirectToAction("Index", "Home");

Becomes

return RedirectToUmbracoPage(1058);

Notice the hardcoded node ID (1058) - we will fix this problem in a later exercise.

The controller is added to the Umbraco site already, and you can therefore add products to the cart already.

# 

# Exercise 5: Provide Related Album Review content From Umbraco

Let’s go further and create our own rule for associating an Umbraco content item with our custom route.

In the previous exercise, we used the UmbracoVirtualNodeByIdRouteHandler with the hardcoded Id of our homepage, to provide the Umbraco Context for the Umbraco template to display the Album details. However you do not necessarily need to hardcode an Id here, you can create your own VirtualNodeRouteHandlers and implement your own logic to associate a more useful UmbracoIPublishedContent item with your custom route.

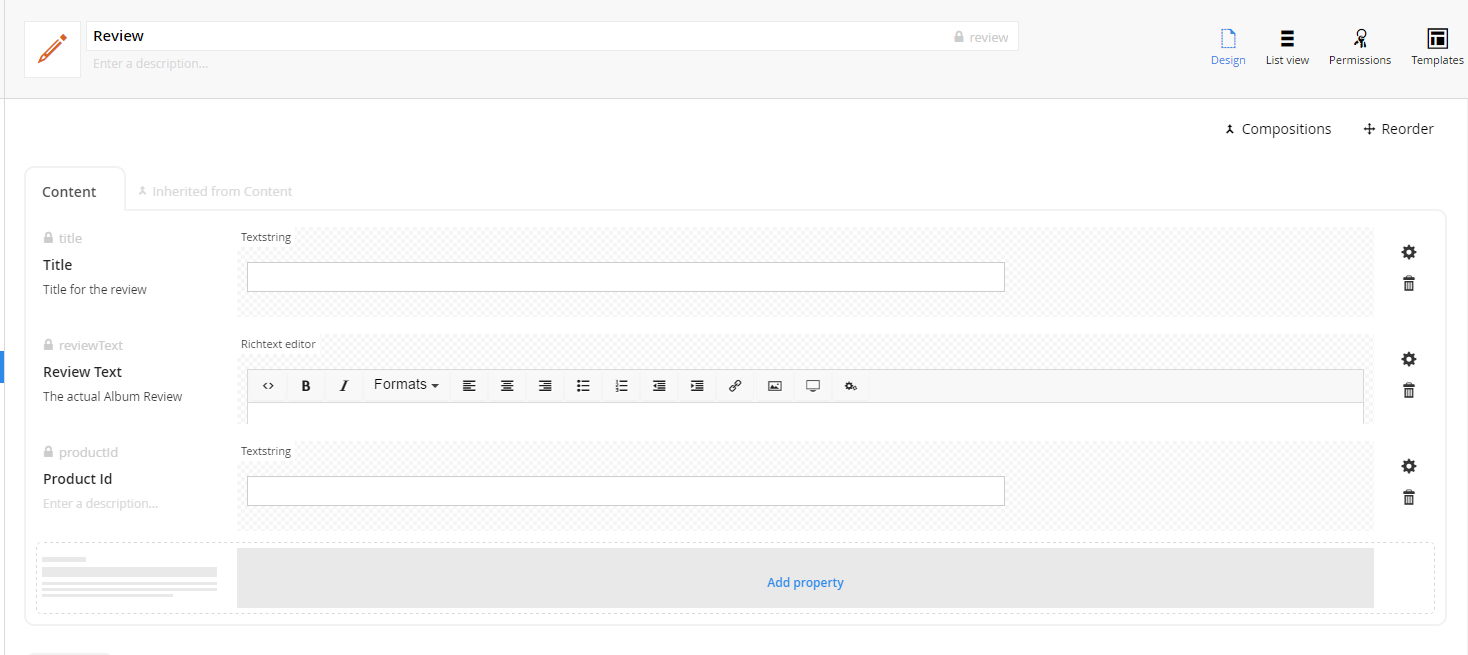
In this exercise we will

* Create a ‘Review’ Document Type
* Create a ‘Review’ content item for one of the Albums (be kind to The Joshua Tree)
* Create an alternative UmbracoVirtualNodeRouteHandler to associate review content with the custom route.
* Update our details route config to use the new handler
* Update the ‘details’ view to display a review if one exists for the Album.

First Let’s Create a new ‘Review’ Document Type, (without a template):

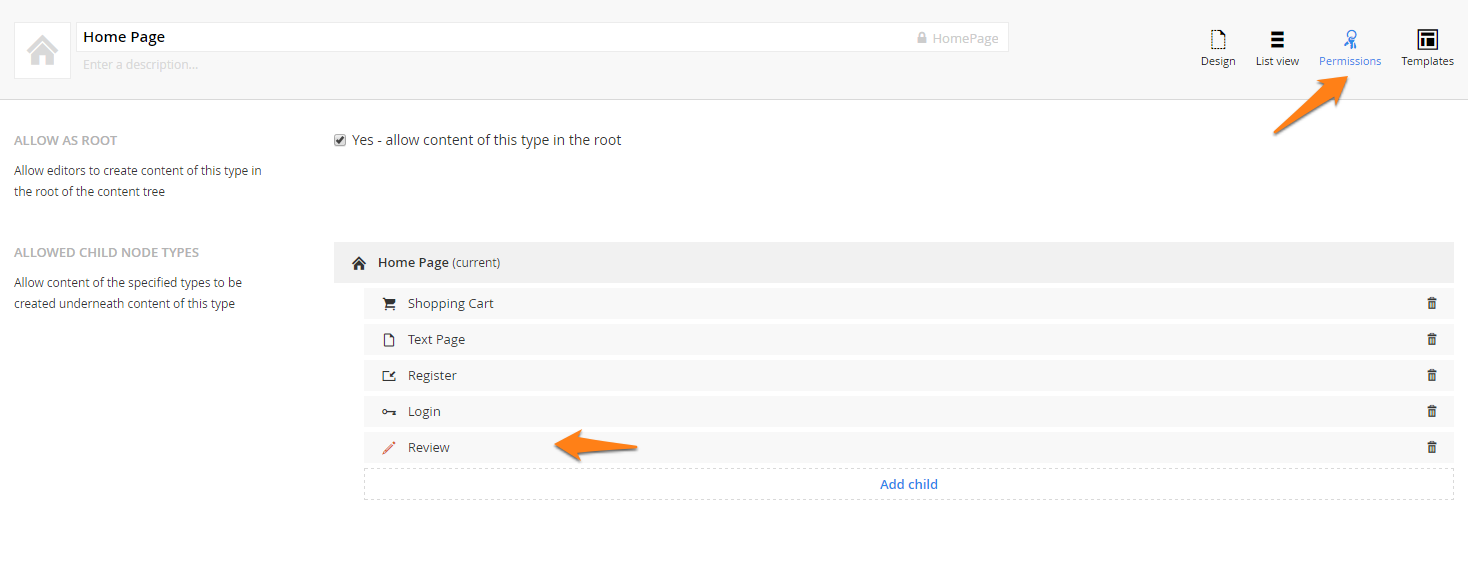
With the following properties:

* Title (title)- Textstring
* Review text (reviewText) - Richtext Editor
* Product Id (productId) - Textstring



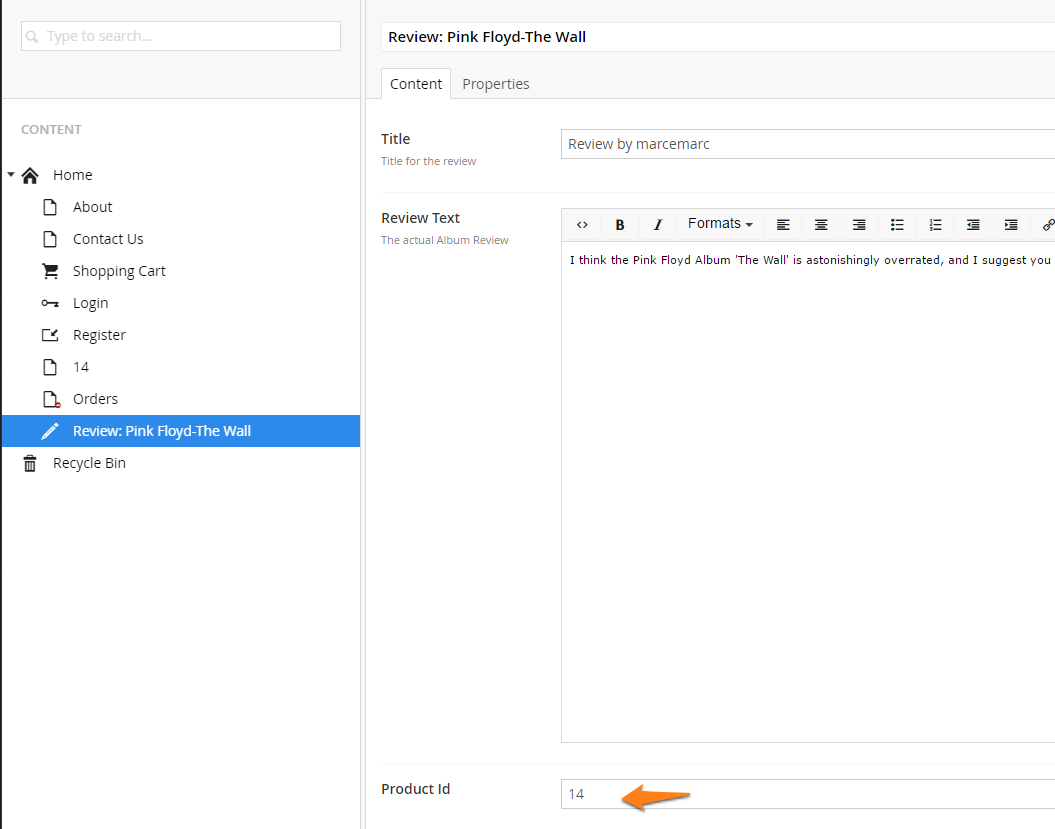
Now permit this Review document type to be created underneath the Homepage.

(we appreciate that’s not very real world!!)



**Notice**: As we use the ModelsBuilder to generate a dll - remember to generate and save your models - this is a manual task

And now create a review for one of the albums



Creating the VirtualNodeRouteHandler

Now in the App\_Start/UmbracoRouteHandlers folder add a new class:

**FindProductReviewRouteHandler.cs**

To create a custom route handler we need to inherit from the class UmbracoVirtualNodeRouteHandler, and we need to implement a single abstract method called FindContent, that is responsible for returning an associated IPublishedContent object for the route.

Our FindProductReviewRouteHandler will need to read the Id of the album from the RequestContext’s RouteData, and then use this Id to find an associated Review in the Umbraco content tree based on the value entered in the ProductId field in your review.

*(NB: In the real world you are more likely to load your review content asynchronously or at least via a cached Macro/Surface Controller action - The key thing here we are trying to demonstrate is that you can write your own rules to find the content that is used with a custom route)*

The full implementation should look like this:

public class FindProductReviewRouteHandler :UmbracoVirtualNodeRouteHandler

{

protected override IPublishedContent FindContent(RequestContext requestContext, UmbracoContext umbracoContext)

{

var productId = requestContext.RouteData.Values["id"].ToString();

var umbracoHelper = new UmbracoHelper(umbracoContext);  
  
 //GetHomePage is a helper in this project - you could use  
 //GetTypedContentAtRoot

var homePage = umbracoHelper.GetHomePage();

//look for pages with a matching productId

var productPages = homePage.Children(f =>  
 f.HasProperty("productId") && f.HasValue("productId") &&  
 f.GetPropertyValue<string>("productId") == productId);

var productPage = productPages.FirstOrDefault();

if (productPage != null)

{

return productPage;

}

return homePage;

}

}

Update the route mapping

Let’s update our route mapping in UmbracoApplicationEventHandler.cs for the Product Details route to use the new handler:

RouteTable.Routes.MapUmbracoRoute(

"Product details",

"Product/details/{id}",

new

{

controller = "Product",

action = "Details",

id = UrlParameter.Optional

}, new FindProductReviewRouteHandler());

Update the Product Details View

Finally Let’s update our Product details view - /views/product/details.cshtml , to display the review if one should happen to exist:

<div class="col-lg-7">

<h2>@Model.Name</h2>

<h4>@Model.Artist</h4>

<h3>@Model.Content.Name</h3>

**@Html.Partial**("\_Rates", Model.Rates)

<p class="lead">

@Model.Description

</p>

<!-- Review Markup below -->

@if (  
 Model.Content.HasProperty("reviewText") &&  
 Model.Content.HasValue("reviewText")  
 )

{

<h2>Reviews</h2>

<blockquote>  
 @Model.Content.GetPropertyValue("reviewText")  
 <cite>@Model.Content.GetPropertyValue("title")</cite>  
 </blockquote>

}  
<!-- Review Markup end →

<h3>$@Model.Price</h3>

</div>

And now when we view the album details page, if there is no ‘review’ to show, the homepage is still the IPublishedContent item associated with the route, however if a matching review does exist then that content item will become associated to the route and the review text can be displayed.



Notice the name of the Model.Content.Name that we are writing out when a review exists….

‘Homepage’ has become ‘Review: Pink Floyd-The Wall’

# 

# 

# Exercise 6: Finding Content by its Former Url

Our application contains some controllers and views for displaying static content, eg for an ‘About Us’ page. We will want to move this content inside Umbraco to enable it to be easily edited, if the Url changes we do not want to trigger a 404 on the former Url.

We've looked in the previous example how to map a specific route through Umbraco, and associate an IPublishedContent item with the route. This is perfect for the scenario of pulling external content from a large data store to display within our Umbraco templates, without having to import that content into Umbraco; but it’s not quite so well suited if the content HAS moved into Umbraco.

Instead here we can create an IContentFinder to find the content in its new location, as the request is processed by the Umbraco Pipeline.

*(We might find that the old url format perhaps following an MVC /Controller/Action pattern is not desirable moving forward, we’ve probably all seen urls in applications like:* **/about/about-us** or **/info/terms-and-conditions** or **/content/about-us** etc*)*

So in this exercise we will:

* Create a new property on our **TextPage** document type to store the ‘former Url’ of the content for when this has changed.
* Create an IContentFinder to find the content by the former Url.
* Register the IContentFinder in the Umbraco request pipeline.

Consider the About Us Page

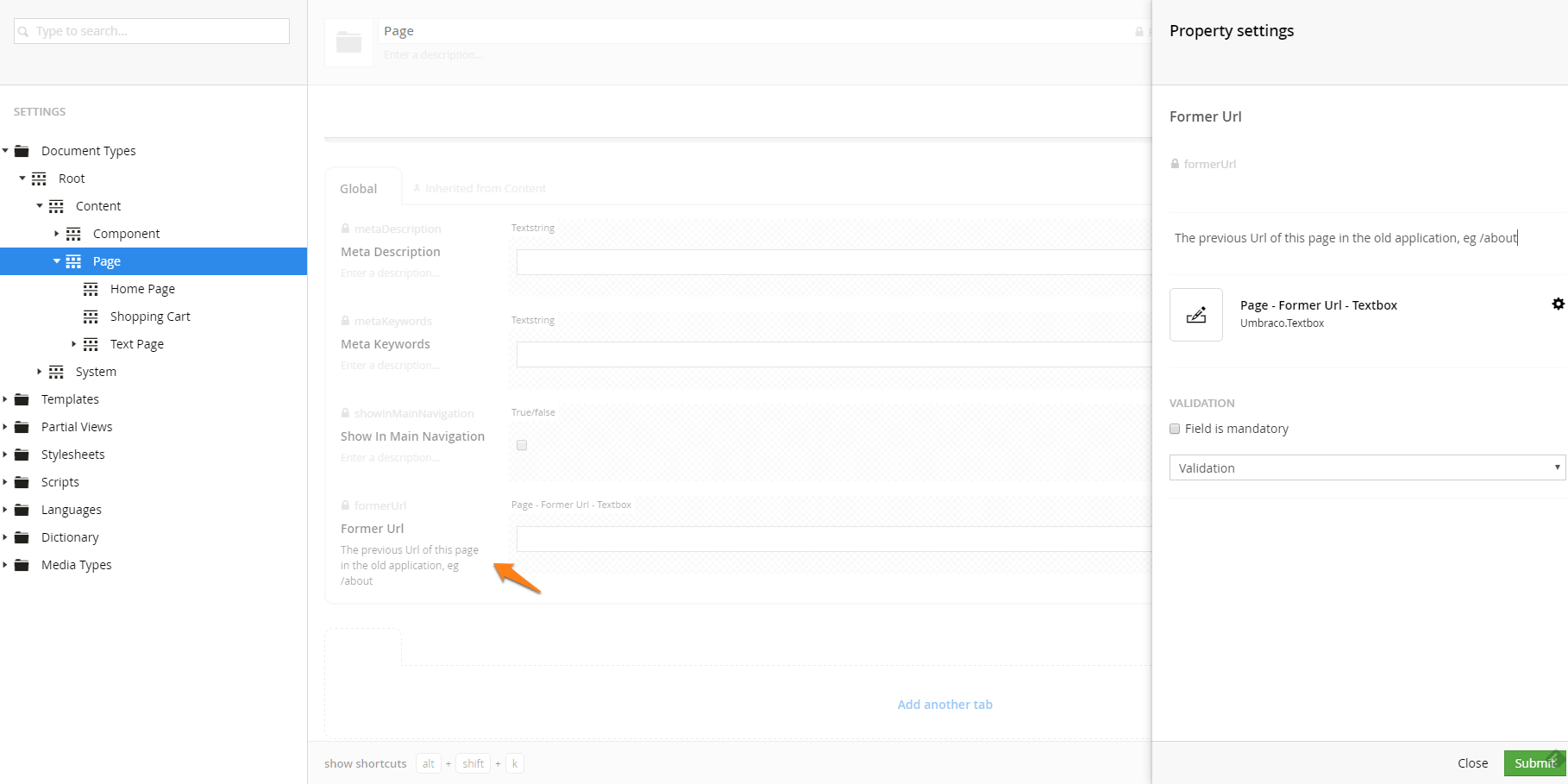
In the Umbraco Store Application we have an AboutController and the url for the About Us page was **/about** - but in our new Umbraco site, the editor has named this page to be the SEO friendly 'About Us' and the url is now **/about-us**.

So we need a way to make the old url map to the new url with Umbraco, (and this may not be an isolated incident) so we will create a generic content finding rule within the Umbraco pipeline, which will only run after all other Umbraco content finding rules have run, to handle any remaining ad hoc static urls that may have changed. We’ll achieve this by storing the old path as a property on the new content item and using this value to match the incoming request.

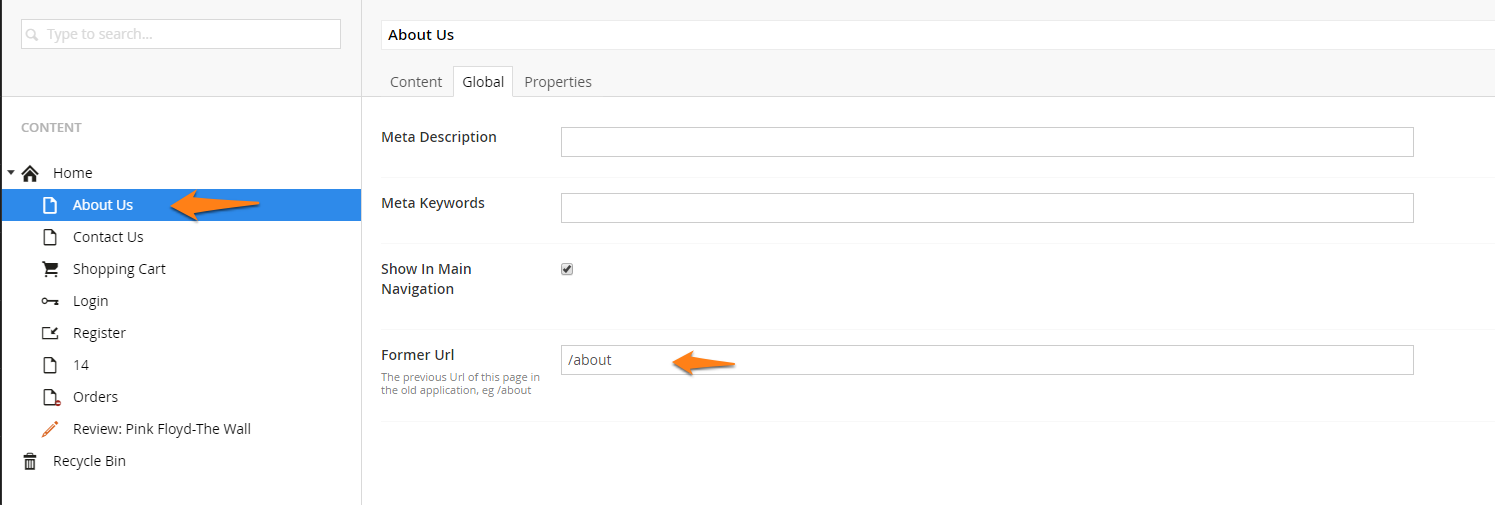
## Create the Former Url property

First let's add a property to the TextPage document type called

FormerUrl (formerUrl) Textbox



And as we know the Url for our About Us page has changed let's fill the value of its Former Url to be /about



## Creating the IContentFinder

Create a folder called **ContentFinders** in our Training.Umbraco.Website.UI project

Create a new class called **FindContentByFormerUrl.cs**

Make our class inherit IContentFinder

We need to implement a single method called **TryFindContent** that accepts the current PublishedContentRequest, and enables you to attempt to find an appropriate item of content matching your custom rules.

If you have found a matching item, your IContentFinder should set the **PublishedContent** property of the PublishedContentRequest to be the located item and return True.

Returning True tells the pipeline not to process any further IContentFinders, in the pipeline - the content for the request has been found.

Returning False from the IContentFinder says "I can't find anything" and lets the pipeline immediately move onto the next IContentFinder in the queue, to see if that one can locate something and so on until a 404 is returned.

Our rule will read the current path, and then look for any node in Umbraco that has a

matching **formerUrl** value: then we will set this to be the **PublishedContent** item of the PublishedContentRequest.

The code will look like this:

public class FindContentByFormerUrl : IContentFinder

{

public bool TryFindContent(PublishedContentRequest contentRequest)

{

// read in the path from the PublishedContentRequest.

var path = contentRequest.Uri.AbsolutePath;

// find any published item with a formerUrl matching the path

var umbracoHelper = new UmbracoHelper(UmbracoContext.Current);

var matchingContent = umbracoHelper.TypedContentSingleAtXPath("//TextPage[formerUrl/text() = '" + path + "']");

if (matchingContent == null)

{

// return false, there is no matching content

return false;

}

contentRequest.PublishedContent = matchingContent;

return true;

}

}

## Register the finder with the pipeline

We have an IContentFinder! - we now have to register it with the Umbraco Pipeline and decide its position within the queue of existing IContentFinders - we only want this finder to execute after Umbraco has tried its best to find content with its built in set of ContentFinders. We use the ContentFinderResolver when Umbraco ‘Starts up’ to add our IContentFinder to the queue.

Open up the **UmbracoApplicationEventHandler** file in the App\_Start folder

find the **OnApplicationStarting** method and add the following line:

ContentFinderResolver.Current.InsertTypeBefore<ContentFinderByNotFoundHandlers, FindContentByFormerUrl>();

We’re using the **ContentFinderResolver** to add our finder just before the position of the core ContentFinderByNotFoundHandlers - Which is usually the final IContentFinder in the pipeline queue.

*(There is a special* ***ContentLastChanceFinderResolver*** *that you can use to register a special IContentFinder that returns a custom 404 page - which will set the correct 404 status code - think multi-lingual site translated 404 pages!)*

## Has it worked ?

Set a breakpoint in the **FindContentByFormerUrl** IContentFinder and visit the old **/about** url in our Umbraco site.

Hopefully now instead of a 404, the content from the **/About-us** page in Umbraco is found and displayed.

**Woot!** - but if you check the url of the page, it still says **/about,** and the status returned is 200 - ContentFound - effectively we now have two urls that find the same **About Us** page, which as any SEO Consultant will tell you is dreadful for SEO!- if we switch back to our **FindContentByFormerUrl** implementation and add:

contentRequest.SetRedirectPermanent(matchingContent.Url);

just before we return true, then Umbraco will instead generate a 301 - Moved Permanently status code, which will earn you the respect of your SEO Consultant and let any search engines know that the page has moved permanently.

So you could use this technique in conjunction with a database table of thousands of old urls from your legacy system, that map to corresponding new urls, and use an IContentFinder to handle the 301 redirects for the old requests as they come in.

*An IContentFinder operates on the Inbound Umbraco Request pipeline and enables you to define your own logic for mapping incoming requests to content. It is also possible to add custom logic to the Outbound pipeline to generate the Url that editors see for a particular content item in the Umbraco backoffice. You would achieve this by creating a class that implements an IUrlProvider, registering it with the pipeline using the* ***UrlProviderResolver*** *on application start up.*

**Documentation**:   
<https://our.umbraco.org/documentation/reference/routing/request-pipeline/outbound-pipeline>

# Exercise 7: Member API

### In this exercise we will:

* Modify the RegisterController to allow registration
* Use the member API to create new members
* Modify the Login method to allow authentication
* Add the current member to an order on checkout

## Setting up Member types and groups

Before we can create anything, we need to set up a Member group to use during registration and authentication.

Member types are very similar to Document and Media types. Member types contain the definition of the information that you want to store about your website members.

Member Groups are used for logical grouping of members. They are also used when securing our content.

First, go to the **Members** section, expand the **Member Types** folder and open the included MemberType called *“****Store Member****”*.

On the member type, add two properties to the Member Address tab:

* **Address**, with the type ***“Textarea”***
* **Country**, with the type ***“Textstring”***

Then create a Member group called *“****Customer****”* by expanding the **Member Groups** folder menu.

We now have the parts needed to begin the work to enable authentication on our website.

## 

## Building a registration form

Open the StoreRegisterModel.cs*,* RegisterController.csand RegisterPartial.cshtml file. As you can see, a form has already been added, and the form has been wired to the register action in the controller. When the form is submitted, the Registermethod is executed, and it is here we will place our Member creation code:

To create a Member, we’ll need to use the MemberService(similar to content and media).

First we check if the new Member will be unique since we don’t want duplicates, so we’ll check by using the GetByEmail method.

var memberService = Services.MemberService;

if (memberService.GetByEmail(model.Email) != null)  
{

ModelState.AddModelError("", "A Member with that email already exists");

return CurrentUmbracoPage();

}

If no member with the submitted email exists, we’ll create a new one with the CreateMemberWithIdentity method on the memberService providing it the username, email, name and the type of member as method parameters.

var member = memberService.CreateMemberWithIdentity(  
 model.Email,   
 model.Email,   
 model.Name,   
 "StoreMember");

We then assign our custom properties as specified on the Member type:

member.SetValue("address", model.Address);  
member.SetValue("country", model.Country);

Don’t forget to save the member by calling the **Save** method on the memberService passing it in the member object:

memberService.Save(member);

We then add the **StoreMember** group to the member using the **AssignRole** method, providing the member id and the group you want to add:

memberService.AssignRole(member.Id, "Customer");

Finally, we’ll need to set the member password and log the member in:

memberService.SavePassword(member, model.Password);

Members.Login(model.Email, model.Password);

**Notice**: how we use MemberService to interact with the Database to perform crud operations and the Members helper to perform session related tasks like authentication - 2 different things and handled with 2 different APIs.

After the member is logged in we’ll redirect them to the homepage:

return Redirect("/");

We should now have a form which enables you to create new members. To place the Registration form on your site add the following code to the register View.

@{  
 Html  
 .RenderPartial("~/Views/Partials/RegisterPartialView.cshtml",  
 new StoreRegisterModel());  
}

# 

# 

# Exercise 8: Member Login & Logout

Besides member registration, we’ll also need the ability for our customers to log in and out of our store.

## Login

In the Training.Umbraco.WebSite.UI project the LoginViewModel, view and controller have already been created, edit the **Login** method in the **LoginController** to perform member authentication using the code below:

if (!ModelState.IsValid)  
 return CurrentUmbracoPage();

if (Members.Login(model.Username, model.Password))  
 return Redirect("/");

//if Members.Login does not succeed - then return login form:  
ModelState.AddModelError("", "Invalid Login");  
return CurrentUmbracoPage();

Save and build your solution. Render the login partial in your solution on the login View in the same way that you rendered the registration form in the previous exercise.

## 

## 

## Logout

For log out, we’ll add the following code to the **Logout** method on the **LoginController**

Members.Logout();

return Redirect("/");

In the solution, the Logout link in the top navigation bar links to:

/umbraco/surface/Login/Logout

This shows how you can call Surface Controller actions directly, which is useful when they don’t render any output. Logout is a good example, as it doesn’t display any output, but simply redirects.

There’s a predictable structure in the URL: /umbraco/surface/**[ControllerName]**/**[MethodName]**

You can use Url.Action(“actionname”,”controllername”)

Eg:

<li><a href="@Url.Action("Logout","Login")">Log off</a></li>

The controller name is just the controller’s class name without the “Controller” suffix (so LoginController turns into Login, RegisterController turns into Register etc.).

# 

# 

# Optional Exercise 9: Checkout

Now that we have products in our Cart let’s make a few updates to allow us to checkout. The store app we’re integrating doesn’t have its own Order implementation as that is commonly a feature that is passed off to a third-party provider. For our simple store we’ll send the current Member a confirmation of the Order on Checkout, and that’s as far as we’ll go.

### In this exercise we will:

* Add a CartHelper to centralize Cart methods and move the code from the ShoppingCartController
* Update the CheckoutController to handle creating an order
* Send an email confirmation
* Clear the Cart

We’ll start by creating a new class in the Helpers folder called CartHelper and add some basic code to allow access to the ProductService consistent with our approach with the rest of the project:

public class CartHelper

{

protected readonly IProductService ProductService;

public CartHelper()

{

ProductService = new PetaPocoProductService("onlineStoreDb");

}

...

Then we’ll copy some of the methods from the current ShoppingCartController to this new class. Copy these methods:

* GetCurrentShoppingCartItems
* GetProductsInCart

As we’ll be accessing these methods from other code we need to update their signature to public like so:

public IEnumerable<T> GetCurrentShoppingCartItems<T>() ....

We can now update the ShoppingCartController methods we have copied to instead return their functionality from the methods on the CartHelper and remove the duplicate code from the controller.

private IEnumerable<T> GetCurrentShoppingCartItems<T>()

{

var cart = new CartHelper();

return cart.GetCurrentShoppingCartItems<T>();

}

private IEnumerable<Product> GetProductsInCart()

{

var cart = new CartHelper();

return cart.GetProductsInCart();

}

We’ll also update the CheckoutController to handle creating an order and sending an email confirmation.

// get current Cart

var cart = new CartHelper();

var products = cart.GetCurrentShoppingCartItems<ShoppingCartProductPreviewViewModel>();

// get current Member

var memberEmail = "storemember@local.com";

if (Members.IsLoggedIn())

{

var member = (MemberPublishedContent)Members.GetCurrentMember();

memberEmail = member.Email;

}

// send an email confirmation

SendOrderConfirmation(products, memberEmail);

// clear Cart

ShoppingCartContext.Current.ClearCart();

return Redirect("/");

We’ve already added the SendOrderConfirmation method to the controller so you’ll just need to update the smtp.PickupDirectoryLocation property to point to the physical location for your local Umbraco instance.

This makes the final connection between our store, and our member implementation. There are of course multiple questions to ask yourself at this point.

* Does it make sense to have my customers in Umbraco or in a external membership provider?
* What kind of data and logic is required to pass from the store API to umbraco and back again?
* How do I manage shop data and customers inside Umbraco?

# 

# 

# Exercise 10: Admin - Property Editor with custom data

We’ll create a product picker which will be used from the Umbraco backoffice with custom data fetched from the store ProductService. The Property Editor is basic and the code is mostly in place already. We just need to wire up the store product data using an ApiController that we’ll call from an Angular Controller.

As with other Umbraco Property Editors we define our Editor using the package.manifest file in the Editor’s /App\_Plugins/StoreProductPicker folder. Our manifest is very simple, just defining the Angular view and the Angular controller that will be used, this manifest has already been added to your solution.

{

propertyEditors:[

{

name: "Product Picker",

alias: "store.productpicker",

editor: {

view: "/app\_plugins/StoreProductPicker/editor.html"

}

}

],

javascript:[

"/app\_plugins/StoreProductPicker/editor.controller.js"

]

}

We can look at the view and controller to get an idea of how the editor will work. In the Angular controller the most interesting line is where the call is made to our WebApi controller. Notice the Umbraco routing conventions for our controller.

$http.get("backoffice/api/ProductPicker/GetAllProducts").  
 then(function (response) {

$scope.overlay.products = response.data;...

Our WebApi Controller needs to have the GetAllProducts method that uses the existing store ProductService to fetch the products list and return it to the Angular controller.

There is already added a basic scaffold of the JsonController we will use to return data to the Property editor, which looks like below:

Using Training.Services.OnlineStore.Models;  
Using AutoMapper;  
   
public class ProductPickerController : UmbracoAuthorizedJsonController

{

protected readonly IProductService ProductService;

protected readonly ICategoryService CategoryService;

public ProductPickerController()

{

ProductService =

new PetaPocoProductService("onlineStoreDb");

CategoryService =

new PetaPocoCategoryService("onlineStoreDb");

}

}

This should look familiar as we’ve used a similar approach in other exercises today. Notice that the controller inherits from UmbracoAuthorizedJsonController which requires an authenticated Umbraco user to allow access and will return Json formatted data by default. This way we don’t need to worry about the method being accessed by anyone other than an already logged in Umbraco user.

To complete this controller, lets add 2 methods to fetch data from the product service:

public IEnumerable<ProductPreviewViewModel> GetAllProducts()

{

var products = ProductService.GetAll(new ProductSearchFilter());

return Mapper.Map<IEnumerable<ProductPreviewViewModel>>(products);

}

public ProductPreviewViewModel GetProduct(int id)

{

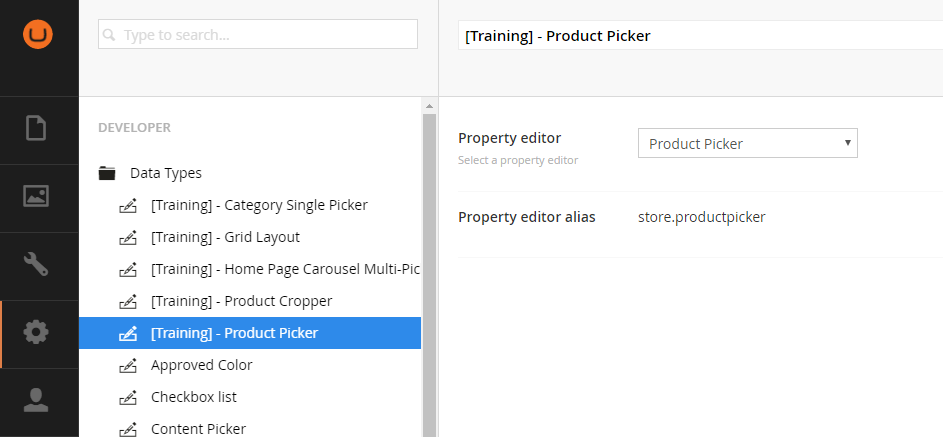
var product = ProductService.GetById(id);

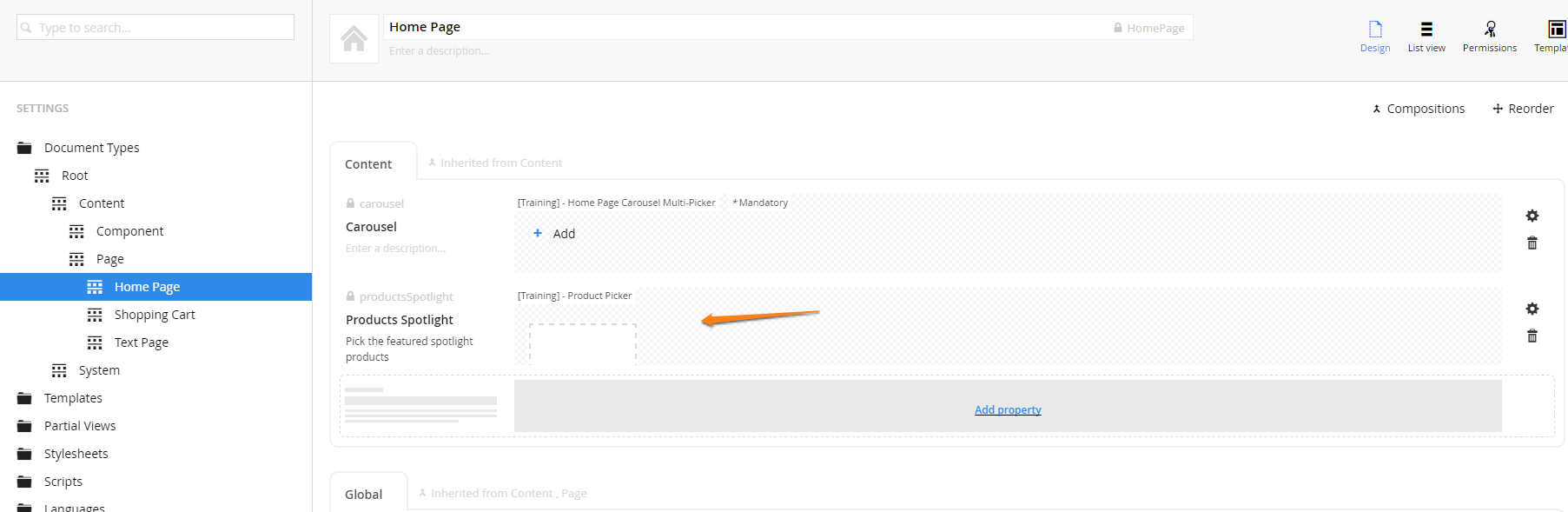
return Mapper.Map<ProductPreviewViewModel>(product);

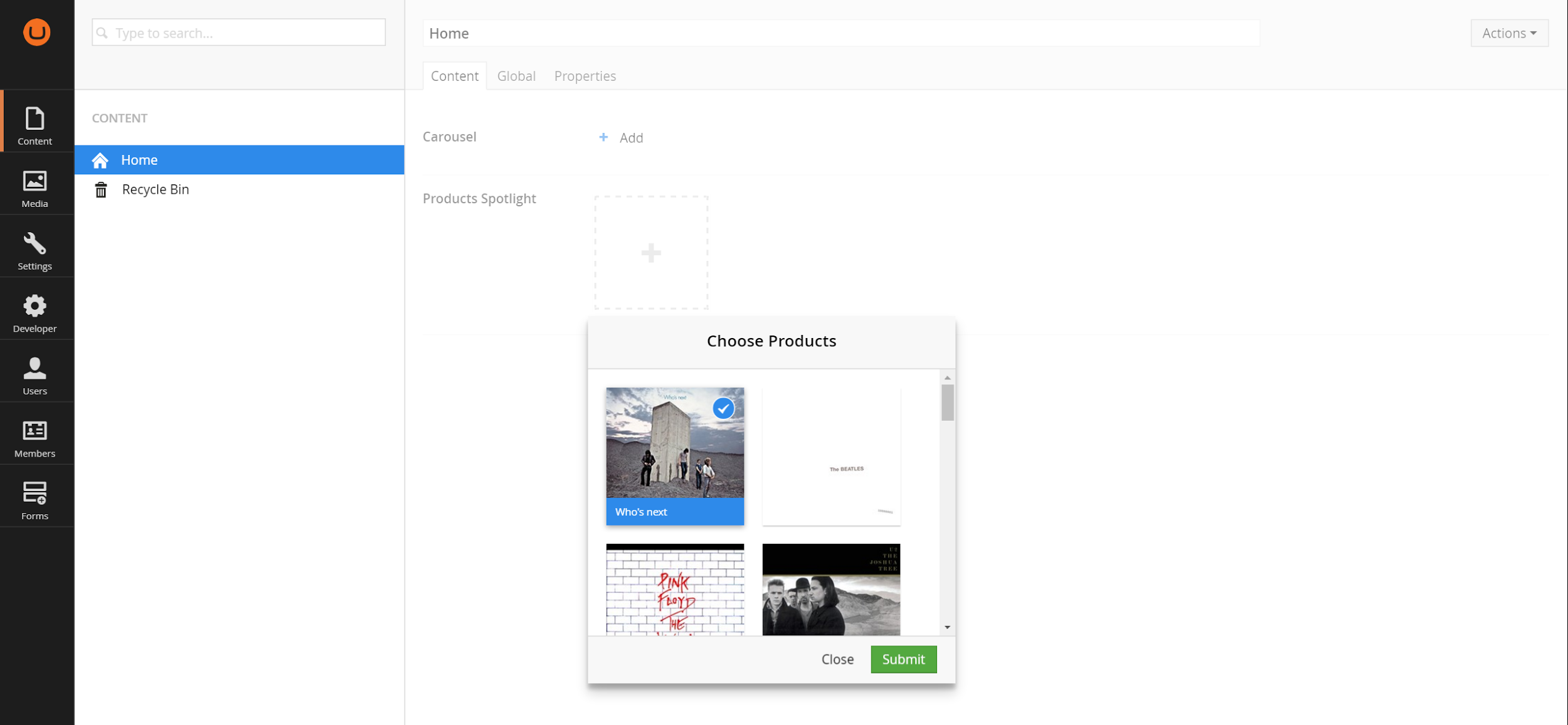
}

These are very basic methods to either query for all products or get a product by ID - this could be further optimised when the store gets a larger number of products.

All that’s left now is to create the Product Picker Data Type in the backoffice, based on this custom property editor and add it to a Document Type, and add some simple code in our view to render the property value. Name the document type property “Product Spotlight” and choose the “Product Picker” property editor.







You are picking content from outside of Umbraco, but from the back office!!! The Property editor stores the Ids of the picked products

If you add the following to your home page View:

<h1>@Model.Content.GetPropertyValue("productSpotlight")</h1>

You can see the raw product id values stored by the picker eg

**[ 7, 19, 14, 20 ]**

### Convert to a strongly typed model

While an array of product ids are fine to work with, we would like to avoid any logic in our views, (ie making requests to the product service to return the details of each product by these picked ids) - so to make the product data a native part of our model, we can introduce a *Property Value Converter*.

(A *Property Value Converter* is a class within your project that implements IPropertyValueConverter and is registered with Umbraco for a particular Property Type; its job is to convert the stored value for that property type into an appropriately typed object to use in your views.)

This converter will return a collection of Product objects instead of a collection of ids.

First, add a new folder to the project called “ValueConverters” - then add a new class to that folder, name it ProductPickerConverter.cs.

Then implement the PropertyValueConverterBase and 2 of its methods:

public class ProductPickerConverter : PropertyValueConverterBase

{

public override bool IsConverter(PublishedPropertyType propertyType)

{  
 //store.productpicker comes from the package.manifest

return propertyType.PropertyEditorAlias.Equals("store.productpicker");

}

public override object ConvertSourceToObject(PublishedPropertyType propertyType, object source, bool preview)

{

if (source == null) return null;

var array = JsonConvert.DeserializeObject<int[]>((string)source);

var ps = new PetaPocoProductService("onlineStoreDb");

return array.Select(x => ps.GetById(x)).Where(x => x != null);

}

}

The IsConverter method helps to determine if this converter should run, in this case we want it to run if the Property Editors alias is “store.productpicker”.

The ConvertSourcetoObject converts the store value, into whatever model we want to use for properties of this type.

Finally, we want to tie this converter into Umbraco’s model system, so we will first of all add 2 attributes to our converter class:

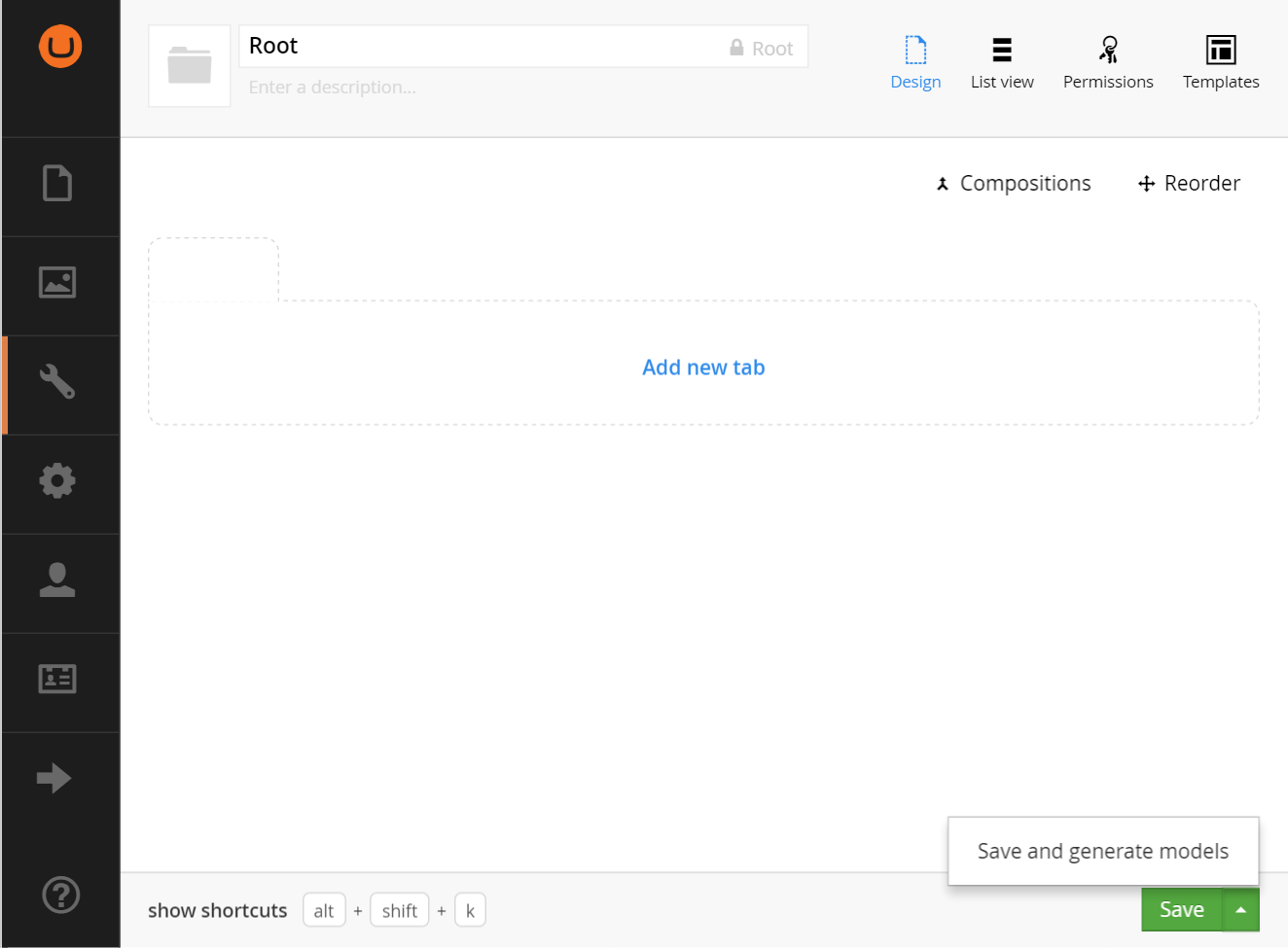
[PropertyValueType(typeof( IEnumerable<Product>))]

[PropertyValueCache(PropertyCacheValue.All, PropertyCacheLevel.Content)]

public class ProductPickerConverter : PropertyValueConverterBase

These 2 helps Umbraco determine what type of value it can expect when it retrieves it from the cache.

Secondly, we will rebuild the strongly typed models, in your project, these are set to be built to a DLL manually, so open any Document type and pick the “Save and generate” menu option.



You can now start using this property on your strongly typed models and get full intellisense like so:

@inherits StoreTemplatePage<HomePage>

<ul>

@foreach(var product in Model.Content.ProductSpotlight)

{

<li>@product.Name - @product.Artist</li>

}

</ul>

Or with the standard getPropertyValue method like so:

@Model.Content  
.GetPropertyValue<IEnumerable<onlineStoreDb.Product>>("productSpotlight")

# Exercise 11: Indexing External Content With Examine

In complex integration scenarios where some content resides in Umbraco, and some content resides in an external database, how would you manage search ?

Umbraco ships with Examine - a flexible and configurable .Net indexing and search engine powered by Lucene.Net - and all content inside Umbraco is indexed ‘out of the box’ in the Internal, External and InternalMembers indexes. But these indexes won’t contain our external content, fortunately we can create a Custom Examine Index based on our external product data.

In this exercise we will

* Create a ProductDataService to implement Examine’s ISimpleDataService
* Create a **ProductIndex** in our Umbraco site’s Examine configuration
* Configure an Examine Indexer and Examine Searcher to build and search the custom index

## Creating an Examine DataService

To create a custom index with Examine, we need to create an **Examine DataService** to be responsible for populating the index.

Examine provides an interface to enable you to create a simple data service, the ISimpleDataService, and all that is required is to implement a ‘GetAllData’ method, that as the name suggests, needs to return all of the data that we wish to index, in the format of a special Examine SimpleDataSet class that Examine knows how to work with.

## Creating a Product Data Service

To give you an quick introduction into creating a custom Examine Index, we’ll create an ISimpleDataService implementation to index our extensive data store of classic Albums.

Add a folder to the Training.Umbraco.Website.UI project called DataServices and create a new class:

**ProductDataService.cs**

That implements ISimpleDataService

Our PetaPocoProductService already has a GetAll method that will return all of our Album products (if an empty Search Filter object is passed to it) - so our custom ProductDataService implementation should resemble the following:

public class ProductDataService : ISimpleDataService

{

public IEnumerable<SimpleDataSet> GetAllData(string indexType)

{

var allData = Enumerable.Empty<SimpleDataSet>();

var productService = new  
 PetaPocoProductService("onlineStoreDb");

var productSearchFilter = new ProductSearchFilter();

var products = productService.GetAll(productSearchFilter);

if (products.Any()){

allData = products.Select(f => new SimpleDataSet()   
 {   
 NodeDefinition = new Examine.IndexedNode() {   
 Type = "content",   
 NodeId = f.Id },   
 RowData = new Dictionary<string, string>() {   
 {"title", f.Name },   
 {"artist", f.Artist },   
 {"description", f.Description },

{"category", f.Category.ToString() }   
}

});

}

return allData;

}

}

We’re passing a dictionary of **RowData** into the index from our product data, and specifying that the **IndexedNode** type is Content and we need to also ensure each indexed item has a unique nodeId identifier, for which we will use our database product Id.

## 

## Adding the Index to Umbraco

We now need to tell Examine about our new ProductDataService and use it to build a custom ProductIndex containing our external Album product content. There are two configuration files for Umbraco’s Examine Indexes and settings.

Open up the ExamineIndex.config file in the Umbraco project’s /Config folder, here we will define our IndexSet:

<!-- custom product index-->

<IndexSet SetName="ProductIndexSet" IndexPath="~/App\_Data/TEMP/ExamineIndexes/{machinename}/ProductIndex/">

<IndexUserFields>

<add Name="title" />

<add Name="artist" />

<add Name="description" />

<add Name="category" />

</IndexUserFields>

</IndexSet>

Notice it’s here we define the fields our index will use: eg, title, artist, description and category and these names match the dictionary key names we are passing into our **RowData** Dictionary in the ProductDataService we created at the start of the exercise - We set here the location on disk where Examine will build the index.

## Configuring the Indexer

In the ExamineSettings.config file (also found in the /Config folder): we can define how the ProductIndex will be created - we specify that we will be using a SimpleDataIndexer which operates in conjunction with an ISimpleDataService, and so we can configure our ProductDataService here as the dataService.

<add

name="ProductIndexer"   
 type="Examine.LuceneEngine.Providers.SimpleDataIndexer, Examine"

dataService="Training.Umbraco.WebSite.UI.DataServices.ProductDataService,  
 Training.Umbraco.Website.UI"

indexSet="ProductIndexSet" runAsync="true" indexTypes="content" />

*(Notice the format of the dataService is the full path to the data service class followed by the name of the dll where the class resides)*

## Product Searcher

And finally we need to add a configuration of a searcher to determine how the index will be searched:

<add name="ProductSearcher"

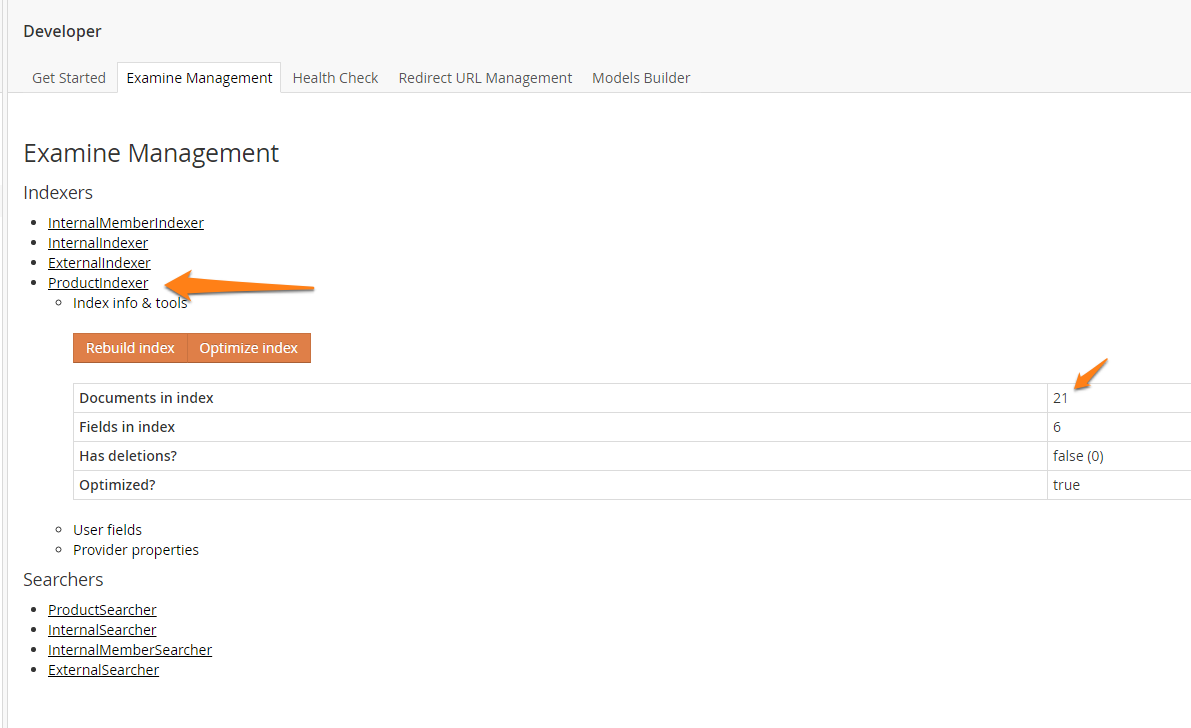
type="Examine.LuceneEngine.Providers.LuceneSearcher, Examine"

analyzer="Lucene.Net.Analysis.Standard.StandardAnalyzer, Lucene.Net"

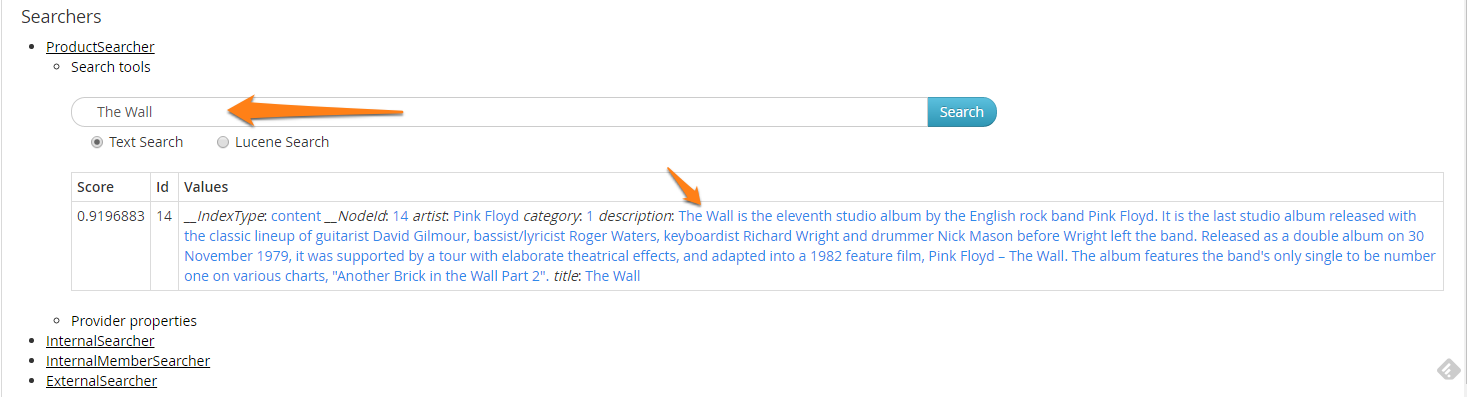
indexSet="ProductIndexSet" />

*Note: There is nothing ‘custom’ about how the index is searched!*

Now login to the Umbraco backoffice and visit the **Developer** section and the **Examine Management** tab, you should be able to see our freshly configured **ProductIndexer**, containing approximately 21 albums:



And if you expand the **ProductSearcher** you can search the indexed content by keyword



If there is time you could go further here and implement a custom search page, perhaps using Examine’s MultiIndexSearcher that enables multiple indexes to be searched by one query.

<add name="MultiIndexSearcher"

type="Examine.LuceneEngine.Providers.MultiIndexSearcher, Examine"

analyzer="Lucene.Net.Analysis.Standard.StandardAnalyzer, Lucene.Net"

indexSets="ProductIndexSet,ExternalIndexSet" />

# 

# Umbraco Support

Get access to developer tools, enterprise level support, a bug fix warranty, and architectural advice directly from the core team.

### Full freedom and dedicated support

Umbraco Confidence is the choice for anyone who implements a professional website based on Umbraco. It combines a commercial license which removes any open source restrictions, with guaranteed support from members of the core team.

### Everything you need

Umbraco Confidence is a subscription-based bundle of tools and support that makes it easier to use Umbraco in professional environments and adds confidence to your implementation process. Umbraco Confidence includes a suite of tools, as well as optional support and architectural advising, from the people who know Umbraco better than anyone, the founders and core team. Umbraco Confidence can be licensed per site or per server and can either be purchased once or as a yearly subscription. Simple, powerful, flexible, just like the software.

### Get advice and immediate help

As an optional addition to Umbraco Confidence, we offer guaranteed support that ensures you’ll receive help within 24 hours. This support is handled by people who know Umbraco by heart. The core team has experience with the inner-workings of Umbraco and more than 400 Umbraco implementations in every conceivable environment.

With support subscriptions and Umbraco Confidence, you’ll get real solutions to your real challenges. Better yet, before those challenges arise you can contact us for architectural advice to ensure that all your Umbraco implementations follow best practices.

### Guaranteed response from three time-zones

With Umbraco Confidence you can get guaranteed response within 24 hours. And with a team working across nine time-zones, the response time is usually very fast.

## http://umbraco.com/products