Introduction to Merchello

Building a Merchello-powered online store
using Ditto and the DitFlo pattern

About me

Freelance Umbraco Developer

Bath

@barryfogarty

www.barryfogarty.com

Workshop Overview

- Part 1: Build an online store using Merchello
- Explore the basic features and configuration options of Merchello
- Discover the new product content type and collection features
- Use Ditto with DitFlo to render out product content
- Part 2: Demo of a full-featured Merchello site based on Ditto and DitFlo

NB: We will not be installing Bazaar, customising the basket / checkout or order processing parts of Merchello.

What is Merchello?

- OSS eCommerce extension for Umbraco, by Rusty Swayne
- Install as an Umbraco package or Nuget
- Reaching a point of maturity in terms of features
- Closely tied to Umbraco: concepts, ideology
- Optional starter kit (Bazaar) with standard eCommerce features implemented, and theme engine
- Today we are taking a different approach
  (Bazaar has been covered in other places)

What is Ditto?

Ditto is a lightweight model mapper for Umbraco. It offers a generic solution to the problem of using **strongly-typed models in our MVC views**.

Created by Lee Kelleher

https://github.com/leekelleher/umbraco-ditto

http://umbraco-ditto.readthedocs.org/

What is DitFlo?

A simple workflow for Umbraco using Ditto to create strongly typed models.

Prevents 'ceremony' code cluttering up our controllers and views.

Created by **Matt Brailsford**

https://github.com/mattbrailsford/umbraco-ditflo

Let's crack on

We'll be building an online surf shop

- Create products and collections in Merchello
- Hook up product detail and list templates
- Learn how we can use Ditto/DitFlo to keep our views clean/strongly typed
- Bonus: Create a category-based navigation system

https://github.com/BarryFogarty/Merchello.UkFest.Workshop

Check out **workshop** branch

Step 1. Wire up a text page

Create a **Text Page** doctype (from Base)

1. NB: Do not create a template with it - there is one to attach already
2. **Title** (textstring)
3. **Text** (RTE)
4. Allow Text Page as child node of Home
5. Create an About page
   (sample content in _*Workshop Pastebin* folder at root of repo)

Create a TextPage POCO

Create a class called TextPage in Merchello.UkFest.Web.Models

1. Our DocType properties will be mapped to our class by Ditto
2. **Title** (string)
3. **Text** (string)

TextPage code

```
using System.Web;

namespace Merchello.UkFest.Web.Models
{
    public class TextPage
    {
        /// <summary>
        /// Gets or sets the page title.
        /// </summary>
        public string Title { get; set; }

        /// <summary>
        /// Gets or sets the body text
        /// </summary>
        public IHtmlString Text { get; set; }

    }
}
```

Pass our TextPage Model into our View

This is where DitFlo comes in. It is responsible for populating our Model with Umbraco data and returning it to the view.

It is akin to creating a strongly typed model in a controller and declaring *@inherits UmbracoViewPage<TextPage>* in your view.

However, with DitFlo, no 'ceremony code' is needed (you don't need to create a boilerplate controller simply to map data to your model)

The DitFlo magic:

```
@inherits Merchello.UkFest.Web.Mvc.DitFloView<TextPage>
```

Render model properties in the view

Using DitFlo, properties are stored in the *Model.View* collection.

```
<h1>@Model.View.Title</h1>
@Model.View.Text
<hr/>
```

How did this happen?

Ditto maps properties with the same names as your doctype property aliases automatically.

You can utilise attributes to change this default behaviour and even resolve values using your own code via custom Value Resolvers.

2. Wire up the Header panel

There is a shared **_Header** partial which renders the Base doctype properties (Header tab). Let's Ditto-ise it!

1. Create a POCO class called **Header**, just like TextPage
2. Inject your model to the view using DitFlo, just as before.

MODEL

```
[UmbracoProperties(Prefix = "Head")]
public class Header
{
    /// <summary>
    /// Gets or sets the title.
    /// </summary>
    [UmbracoProperty("HeadTitle", "Name")]
    public string Title { get; set; }

    /// <summary>
    /// Gets or sets the brief.
    /// </summary>
    public IHtmlString Description { get; set; }
}
```

VIEW

```
<h1>@Model.View.Title</h1>
<div class="text-muted">@Model.View.Description</div>
```

What have we done here?

Through the use of class and property-based attributes, we can name our model properties as we like and still instruct Ditto to map to specific property aliases.

Also, we can provide a fallback property (or string value) in case the given value returns null or an empty object.

3. Wire up Meta tags on Base layout

We are going to re-use the header panel properties to populate our meta tags on the Base layout using Ditto.

1. Create a Meta POCO
2. Use a Type Converter to string the HTML from 'headDescription' HTML and convert to a string
3. Wire up the model to the view

MODEL

```
public class Meta
{
    /// <summary>
    /// Gets or sets the title.
    /// </summary>
    [UmbracoProperty("HeadTitle", "Name")]
    public string Title { get; set; }

    /// <summary>
    /// Gets or sets the brief.
    /// </summary>
    [UmbracoProperty("HeadDescription")]
    [TypeConverter(typeof(MetaDescriptionConverter))]
    public string Description { get; set; }
}
```

View as of Header (declaring Meta model)

4. Merchello Settings

Let's look at the global configuration options for Merchello

1. Currency and culture
2. Units and formats
3. Tax and shipping
4. Default product settings

Merchello Products and Collections

Overview of the new Merchello Product features. Let's create a basic product in Merchello (sample content in _Workshop Pastebin)

1. Create basic Product
2. Create basic Product doctype (inherit from base, no additional properties for now)
3. Attach Product doctype to Merchello products

Wire up Product view

Let's use Ditto as before, to create a simple POCO model for our Product content and pass it in to our view.

1. Create Product class
2. Reference it as our view model
3. Render out the model properties on our product page

Product model

```
public class Product
{
    /// <summary>
    /// Gets or sets the key.
    /// </summary>
    public Guid Key { get; set; }

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

    /// <summary>
    /// Gets or sets the price.
    /// </summary>
    public decimal Price { get; set; }
}
```

Product view

```
<div id="add-basket">
    @Model.View.Price
</div>

<div id="details">
    <h4>Product details</h4>
    @Model.View.Name
</div>
```

Advanced Product Content

1. Expand our Product content type to include a multiple images picker, a text field and a 'New Product?' checkbox.

2. Populate the image with content (sample content in Pastebin)

3. Expand our Product POCO so Ditto will map these properties

4. Wire up the new properties in our view (code is in Pastebin)

```
public class Product
    {
        ...

        /// <summary>
        /// Gets or sets the description.
        /// </summary>
        public IHtmlString Text { get; set; }

        /// <summary>
        /// Gets or sets the images.
        /// </summary>
        public virtual IEnumerable<Image> Images { get; set; }

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

        /// <summary>
        /// Gets or sets a value indicating whether on sale.
        /// </summary>
        public bool OnSale { get; set; }

        /// <summary>
        /// Gets or sets a value indicating whether is new.
        /// </summary>
        public bool IsNew { get; set; }


    }
}
```

Add To Basket model

Let's look at how to resolve a complex property on or model - the AddToBasket model that we want to pass to our view and use in our Surface Controller

Note: Some of the referenced code is already included in the project to save time, we will examine what it is doing here.

Add Property to Product

```
/// <summary>
/// Gets or sets the add to basket.
/// </summary>
[CurrentContentAs]
public AddToBasket AddToBasket { get; set; }
```

Alternate: Using Value Resolver

```
/// <summary>
/// Gets or sets the add to basket.
/// </summary>
[DittoValueResolver(typeof(AddToBasketValueResolver))]
public AddToBasket AddToBasket { get; set; }
```

This is an alternative approach to the previous slide. If you need to validate data for instance, it can be done in a value resolver.

Add To Basket controller

Examine the code in the BasketController. Using out of the box Merchello techniques to add, update and remove items from the basket.

Product Listing (Category Page)

Let's create some more sample products, add them to a collection and render them out to a category page:

1. Create 2 more example products. Use the Copy Product function of Merchello
2. Sample content in Pastebin if you want
3. Create a 'Boards' product collection and add products
4. Create a matching category under Store and pick the collection

Wire up the Category view

1. Create a 'ProductList' POCO model
2. Attach it to our _ProductList partial
3. Render out the model properties of each item in the collection in the _ProductBox partial

MODEL

```
public class ProductListing
{
    /// <summary>
    /// Gets or sets the products.
    /// </summary>
    [DittoValueResolver(typeof(ProductListItemsValueResolver))]
    public virtual IEnumerable<ProductListItem> Products { get; set; }
}
```

VIEW

```
@foreach (var product in Model.View.Products)
{
    @Html.Partial("_ProductBox", product)
}
```

_ProductBox nested partial is available in Pastebin

What have we just done?

Using a custom value resolver, we get the value of the 'products' property in the normal way, then we build a collection of **ProductListItems** from the returned collection.

Using a custom value resolver allows us to handle things like null / incorrect data, as well as converting the raw data into something more useful for our application.

The **AsProductListItem** extension simply performs some checks and formatting on the raw data before returning a new instance of our **ProductListItem** type.

Bonus: Category Tree

Wire up the sidebar nav as a tree of category nodes.

1. Create POCO models for the CategoryTree and CategoryTreeItem
2. Create a value resolver to return a list of categories
3. Wire up the CategoryTree partial with our new model
4. Create some nested categories under Store

Code fragments are in Pastebin