Ektron to Episerver Content Creation Lab Instructions

James Stout (@egandalf)

Exec. Director, Technology & Research

Brightfind

Rob Folan

Senior Solution Consultant

Episerver

Overview

Welcome to the Ektron to Episerver Lab, where I hope to show you how to remodel your Ektron content to be Episerver content, using Epi best practices and setting you up for long-term success.

This lab will help you understand how Smart Form content can be adapted into Episerver content. In some cases, it will be very simple and the Smart Form will be replicated almost exactly. In others, we can do the same amount with fewer fields in Epi. Or we may have to invent new functionality, or we may expand on what was done in Ektron (because we can).

If we have time, I will cover migration strategy for the move from Ektron to Episerver.

The goal of this lab is not to make you an amazing Episerver developer, but to help you understand how content can be recreated within Episerver and how the related pieces work together.

We'll also be working in MVC, which may be unfamiliar to some of you.

MVC is very convention-oriented, and Episerver is no exception to that. Each rendering needs a model, a view, and a controller. This Episerver site adds to that convention in a way similar to that of Alloy.

This application has set up some default paths it will use to discover views for Episerver pages and Blocks. In general, views for Pages can be found at **~/Views/{page type name}/index.cshtml**.

Similarly, views for Blocks can be found at **~/Views/Shared/Blocks/{block type name}.cshtml**.

This is noted so you're aware that the application *knows where to look for these files*. From there it shouldn't be a concern.

As you work through the lab materials, I will attempt to minimize the number of times you will need to create a new file. Most of the files are simply stubs where you will be filling in the necessary code.

NOTE

Type names for some Pages and Blocks are tied into other classes and functions on the site. It is therefore HIGHLY RECOMMENDED that for this lab you do not change these class names.

Pages

Services Page

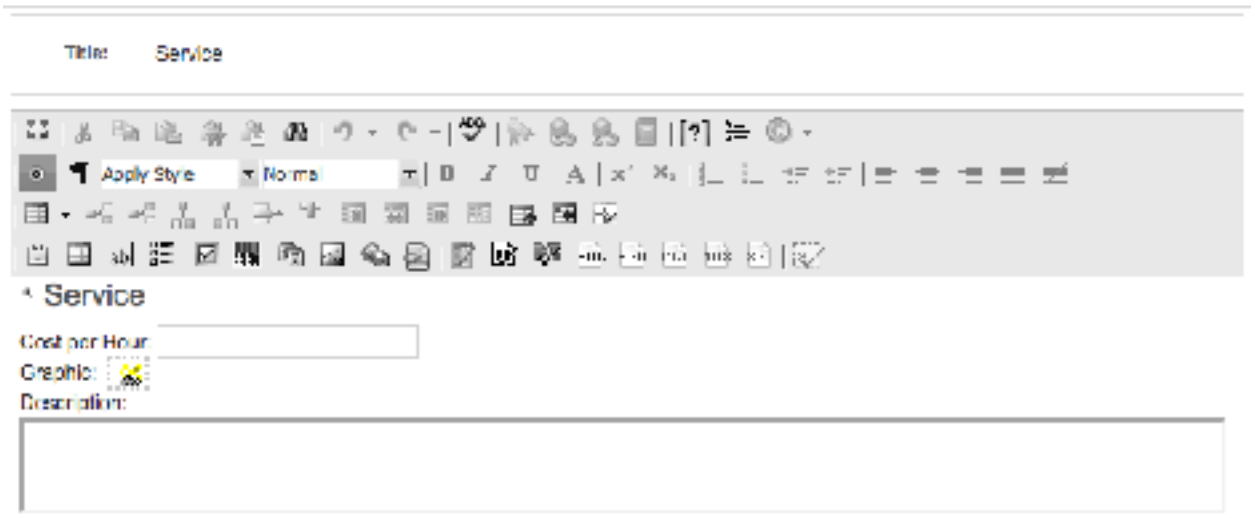
For reference, I'll show you the Ektron definition of the page first.

The Service page is a Smart Form with the following fields:

- Cost Per Hour (Integer)
- Graphic (Image selector which generates an <img /> tag
- Description (Rich Text)

Here we assume the Content Title to be the name of the service and the <h1> of the page.

Ektron Definition

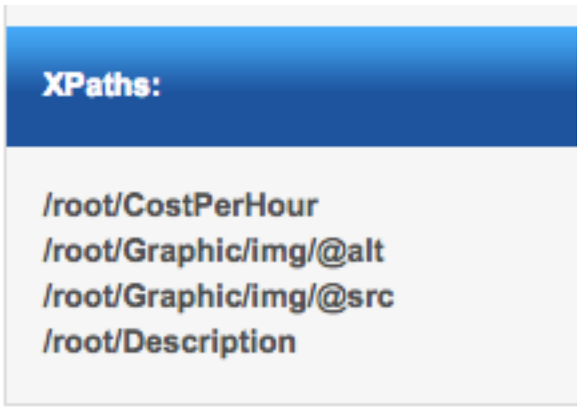


Service

Cost per Hour:

Graphic:

Description:



XPaths:

/root/CostPerHour
/root/Graphic/img/@alt
/root/Graphic/img/@src
/root/Description

Episerver Code

Please recreate this class under the [~/**Models/Pages]** folder of the project. You'll need to add or update the sections **bolded and highlighted**.

```
using System.ComponentModel.DataAnnotations;
using EPiServer.Core;
using EPiServer.DataAbstraction;
using EPiServer.DataAnnotations;

namespace Brightfind.EktronToEpiserverLab.Models.Pages
{
    [ContentType(DisplayName = "Service Page",
        GUID = "a09690a1-71cd-488b-956b-9a2db48cc34b",
        Description = "",
        GroupName = Global.GroupNames.Standard)]
    public class ServicePage : SpockPageBase
    {
        [Display(GroupName = SystemTabNames.Content, Order = 100)]
        public virtual int Price { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 200)]
        public virtual XhtmlString Description { get; set; }
    }
}
```

NOTE

Don't forget to set the **SpockPageBase** type. Our "Base" type has a field for an image, we don't need to replicate that field here. Content inheritance is AWESOME.

GroupName - The GroupName value determines how this page will be grouped for Authors. Best to have the value as something easily reusable (hence the Global class).

SpockPageBase - This is our primary base class for all pages. It's abstract, so can't be used as a page itself, but it allows all pages to inherit the default properties we want them each to have.

Display - The GroupName becomes the "Tab" in the Author UI where this field is found. The Order should be obvious. ;)

Add all of this code to **~/Views/ServicePage/index.cshtml**

```
    @Html.FullRefreshPropertiesMetaData(new[] { "Price" })
<div class="row">
    <div class="col-md-12">
        <h1 @Html.EditAttributes(m =>
m.CurrentPage.Name)>@Model.CurrentPage.Name</h1>
    </div>
</div>
<div class="row">
    <div class="col-md-8">
        <div class="price" @Html.EditAttributes(m => m.CurrentPage.Price)>
            $@Model.CurrentPage.Price/hr
        </div>
        <div class="body">
            @Html.PropertyFor(m => m.CurrentPage.Description)
        </div>
    </div>
    <div class="col-md-4">
        @Html.PropertyFor(m => m.CurrentPage.PageImage)
    </div>
</div>
```

@Html.FullRefreshPropertiesMetadata - Forces all of the content of the page to refresh whenever this value is updated. This will be used again for Events, where it's even more obvious that it's taking effect. Use this whenever updating a field isn't giving you the desired output. In this case, omitting the line causes the Price to refresh without the text around it.

@Html.EditAttributes - Allows us to designate the parent tag as the Editing tag. This will inject the necessary editing code only when the page is in edit view. Keeping the markup clean for the end users.

CurrentPage - You'll see this used throughout. We're using an intermediary ViewModel instead of passing the page directly to the View. This helps us control the overall layout, set the copyright in the footer, and several other things.

You may notice there is no ServicePageController in the **~/Controllers** folder. This is great! Because the content is straightforward, we can simply rely on the DefaultPageController to catch it, generate the ViewModel and pass the data along! NEAT!

Congratulations! This page is done!

Landing Page

The equivalent of a Landing Page in Ektron would be a wide-open PageBuilder Layout. Since there's no Smart Form or other content model to port over, we can dive right in!

```
using System.ComponentModel.DataAnnotations;
using EPiBootstrapArea;
using EPiServer.Core;
using EPiServer.DataAbstraction;
using EPiServer.DataAnnotations;

namespace Brightfind.EktronToEpiserverLab.Models.Pages
{
    [ContentType(DisplayName = "Landing Page",
        GUID = "ff07239b-fe5c-4679-8bfa-4e1fd4797e95",
        Description = "",
        GroupName = Global.GroupNames.Standard)]
    public class LandingPage : SpockPageBase
    {
        [Display(Name = "Main Content Area", GroupName =
SystemTabNames.Content)]
        [DefaultDisplayOption(ContentAreaTags.HalfWidth)]
        public virtual ContentArea MainContentArea { get; set; }
    }
}
```

ContentArea - Can be thought of as similar to a PageBuilder DropZone. This provides a field for a designated area of the page where the author is allowed to make both content and layout decisions.

[DefaultDisplayOption(…)] - Indicates that, unless otherwise specified, any Blocks dragged into this ContentArea should render at half the width of the container.

Insert the following at **~/Views/LandingPage/index.cshtml**

```
@Html.PropertyFor(m => m.CurrentPage.MainContentArea, new { CssClass
= "row equal-height" })
```

@Html.PropertyFor - Will render the more complicated ContentArea field along with Editing capabilities when in that mode. If you don't want editing, use DisplayFor instead!

CssClass - The value entered here will be applied to the ContentArea wrapper div. "row" is a Bootstrap class, while "equal-height" is used specifically for this project.

No Controller necessary since this will be picked up by the DefaultPageController!

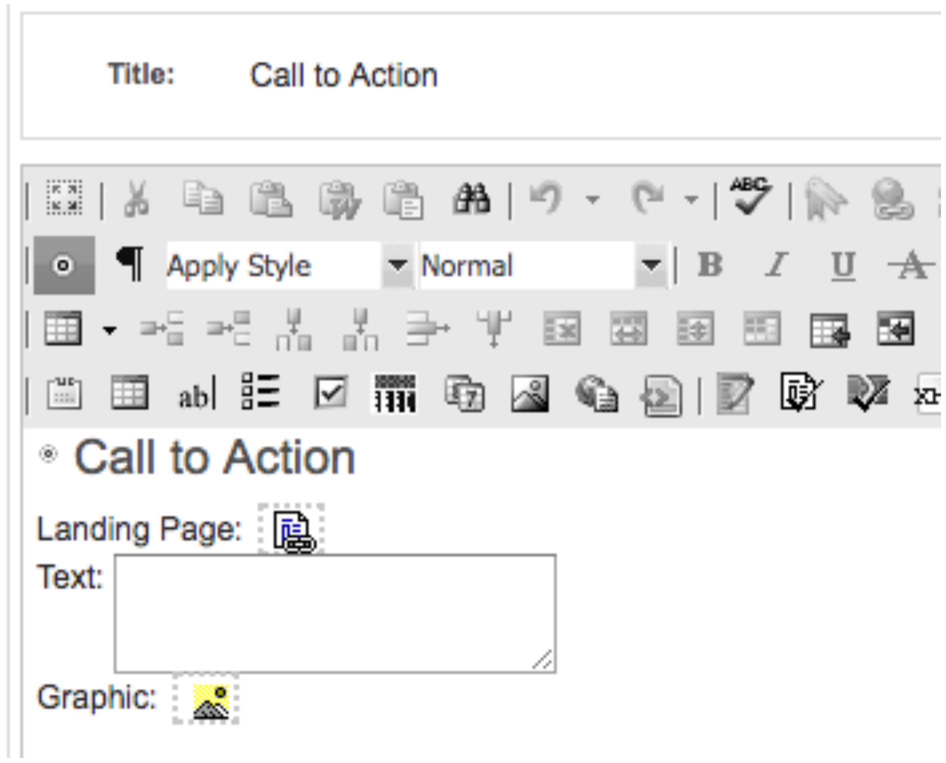
Blocks

Blocks are essentially the same as pages, except they have no direct Route (URL). Though they're often compared to Widgets, this is only due to the drag-and-drop nature of placing Blocks on a page. Where a Widget is nothing more than a rendering, a Block is a content item that can be displayed using one or several different renderings.

Call To Action Block

A Call To Action is a relatively simple Smart Form that's used in conjunction with an Ektron Widget to put CTAs onto the Home page.

Here's the Ektron definition:



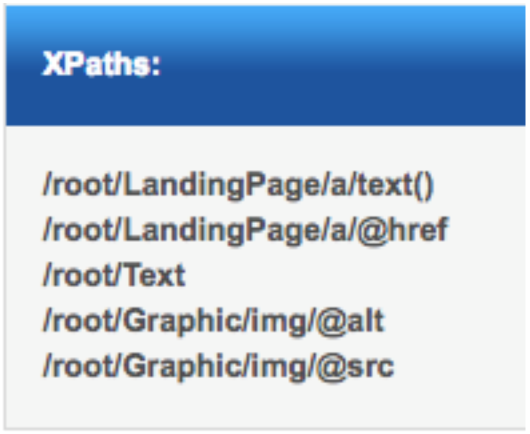
Title: Call to Action

Call to Action

Landing Page:

Text:

Graphic:



XPaths:

/root/LandingPage/a/text()
/root/LandingPage/a/@href
/root/Text
/root/Graphic/img/@alt
/root/Graphic/img/@src

The Call to Action Smart Form consists of:

- A URL field to designate a destination page for the CTA
- A TextArea field for a small bit of text
- An Image field

In Ektron, the Button text is whatever is configured for the Landing Page link field. Episerver splits the link destination and the text into two fields, which is less confusing, though it means creating an extra property in our model (two lines of code).

```
using System.ComponentModel.DataAnnotations;
using EPiBootstrapArea;
using EPiServer;
using EPiServer.Core;
using EPiServer.DataAbstraction;
using EPiServer.DataAnnotations;
using EPiServer.Web;

namespace Brightfind.EktronToEpiserverLab.Models.Blocks
{
    [ContentType(DisplayName = "CallToActionBlock",
        GUID = "5093dddf-b882-498e-8bb8-1732348bbae0",
        Description = "",
        GroupName = Global.GroupNames.Specialized)]
    [DefaultDisplayOption(ContentAreaTags.OneQuarterWidth)]
    public class CallToActionBlock : SpockBlockBase
    {
        [Display(GroupName = SystemTabNames.Content, Order = 100)]
        [UIHint(UIHint.Image)]
        public virtual ContentReference Image { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 200)]
        public virtual string Proverb { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 300, Name = "Button
Text")]
        public virtual string ButtonText { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 400, Name = "Button
Link")]
        public virtual Url ButtonLink { get; set; }
    }
}
```

[DefaultDisplayOption(…)] - *Optional*. This project uses a community add-on called EpiBootstrapArea, which provides a ready-made integration between Episerver ContentArea fields and Bootstrap 3.x. This tells the site that, unless otherwise specified, this Block should render at 1/4 width of the container.

[UIHint(UIHint.Image)] - UIHint provides you with ways of modifying the behavior of the field. In this case, this limits input to only Episerver-managed images and lets the system know that it should render an Image on output (instead of just a hyperlink).

Name - Camel-case labels aren't very friendly for most authors. When you have a multi-word name for a property or field, it's best to use the Name value in the Display attribute to give it a friendly name. You also can provide multilingual options, which is beyond the scope of this LAB.

SpockBlockBase - The Base class created for all Blocks on the site. Though it's just an empty class for now, this allows us to create global properties or functionality for all of our blocks in a single location, should we ever need to.

Place the following code into the View for the Block at
~/Views/Shared/Blocks/CallToActionBlockNarrow.cshtml

```
<div class="hero-feature text-center">
    <div class="thumbnail">
        @Html.PropertyFor(m => m.Image)
        <div class="caption">
            <p @Html.EditAttributes(m => m.Proverb)>@Model.Proverb</p>
        </div>
        <div class="anchor-bottom absolute-center">
            <a class="btn btn-primary" href="@Url.ContentUrl(Model.ButtonLink)"
@Html.EditAttributes(m => m.ButtonText)>@Model.ButtonText</a>
        </div>
    </div>
</div>
```

@Html.PropertyFor - When you've got a more complicated object to render, such as an Image or a ContentArea, which require their own markup (either native or configured by you), you can use a PropertyFor helper to render the field. This includes the Editing capabilities. If you don't want Editing, use DisplayFor instead!

Note: This rendering will not allow you to edit the ButtonLink field using the in-context or preview view of the content. Instead, click the [☰] button in to the top-right of the page to enter the "all properties" view and edit this field. You can simply drag a page from the content tree into the Button Link field!

Because we're passing the Block data directly to the View without any modification - as is the case with most Blocks - we don't need to set up any Controller. Episerver has that one built-in!

BUILD AND RUN

Run the site and add a Landing page and at least two child Services pages to the site. Create a Call To Action Block for each Service and place them on the home page. Set the Call to Action blocks to render at One Quarter width.

Page Partial for the Service Page Type

Pages and Blocks are both just instances of content. It stands to reason, then, that Page content could be rendered like a Block, right?

Yep! And these small steps will eliminate the need for an entire Ektron widget.

Doing this will save us the effort of creating redundant instances of content when creating our Services Landing Page. (Yes, it's redundant with our Call To Action Block Type as well, and we *could* create yet another Service Page View for that rendering. However, the Call To Action could link to *anywhere* - internal or external to the site. So there's still an argument for maintaining that type.)

Add the following to **~/Views/Shared/PagePartials/ServicePageTeaser.cshtml**

```
<a href="@Url.ContentUrl(Model.ContentLink)" title="@Model.Name">
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3>@Model.Name</h3>
        </div>
        <div class="panel-body">
            <div class="row">
                <div class="col-md-8">
                    <div>
                        @Model.TeaserText
                    </div>
                    <p class="pull-right">
                        <span class="label label-default">$@Model.Price/hr</
span>
                    </p>
                </div>
                <div class="col-md-4">
                    @Html.DisplayFor(m => m.PageImage)
                </div>
            </div>
        </div>
    </div>
</a>
```

Note the **PagePartials** in the path. This organization decision lets us know that this rendering is for a Page rendered as a Partial and not a Block.

This will not be found automatically, however. In this application, we're using a class called TemplateCoordinator to configure additional rendering options for our content.

Open **~/Business/Rendering/TemplateCoordinator.cs** and add the following code at **line 35**

```
viewTemplateModelRegistrator.Add(typeof(ServicePage), new TemplateModel
            {
                Name = "ServicePagePartial",
                AvailableWithoutTag = true,
                Inherit = true,
                Path = PagePartialPath("ServicePageTeaser")
            });
```

With this code, Episerver will register a new View option for content of type ServicePage. This will be the default rendering for any ServicePage dragged into any ContentArea. We could continue to add TemplateModels for various DisplayOptions and other content types. In fact, you'll see two in here already!

AvailableWithoutTag - Means that this rendering is globally available. You can specify tags and set this value to false, which would mean that a Block cannot be rendered inside a ContentArea unless it has that tag. It's an interesting way to add restrictions to what content can be placed where within the site. For an example, see the TemplateCoordinator entry for the JumbotronBlock type, which can only be rendered in a FullWidth ContentArea.

PagePartialPath - This is calling a private method at the bottom of the file. Allowing us to specify only the name of the view and the method will fill in the rest. A great example of DRY development (Don't Repeat Yourself).

BUILD AND RUN

Run the site and go to your Landing page in edit mode. Drag services pages directly into the Landing page content area. Set each to render at Half width. The result should be similar to: http://lab.brightfind.com/Services/

Splitting Ektron Content

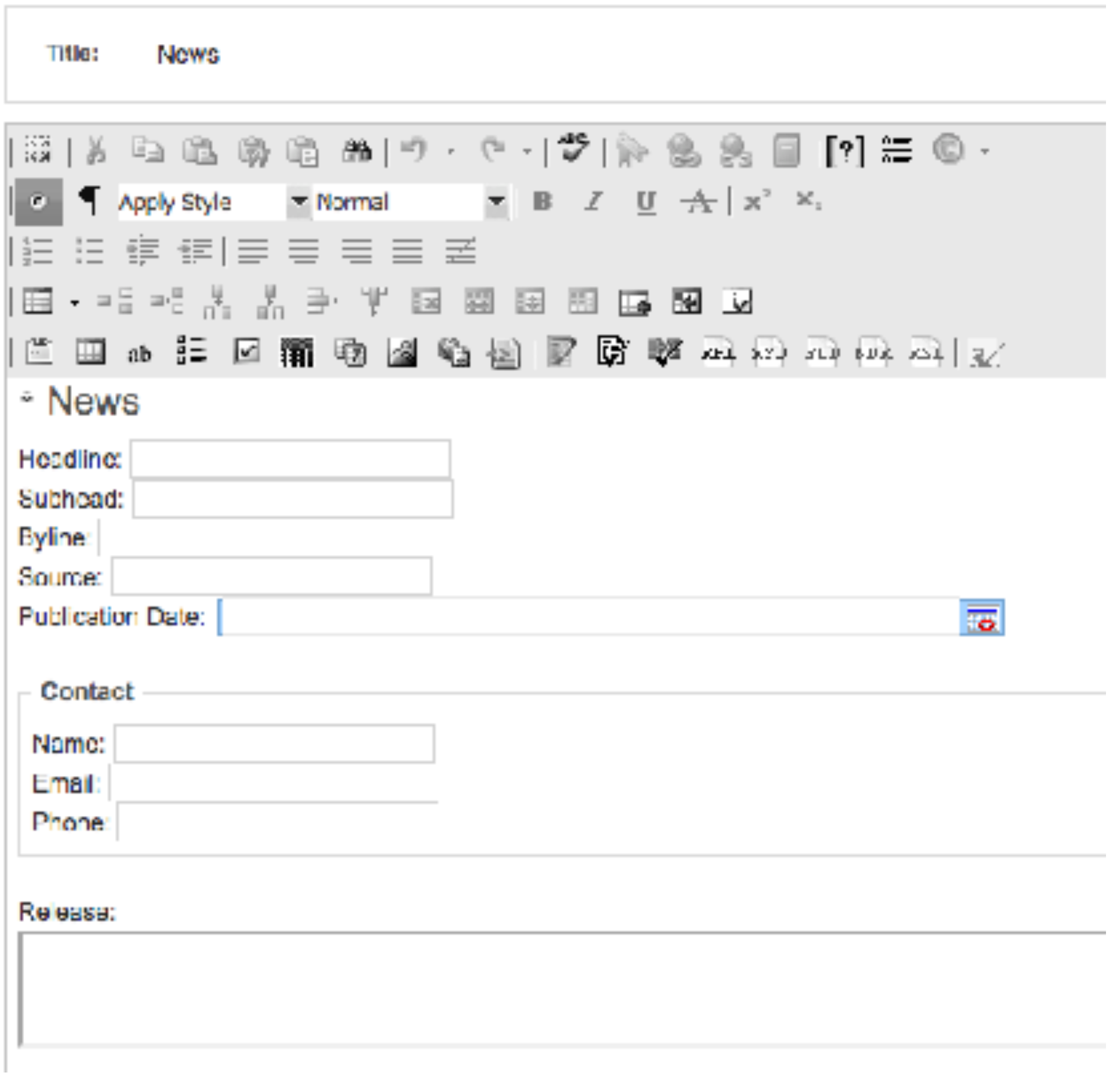
The News content in Ektron is a more robust Smart Form with a “Group Box” where the author would enter contact information for each release.

One problem with this approach is that the contact information has to be entered every time. And while it is possible to directly embed one type within another type (e.g. using a Block as a property within a Page to create a similar concept to the Group Box), it’s typically not preferable. In this case, it would be better to have the Contact information be reusable, which means making it as a separate Block type from the Page.

In addition, News needs a list-style page on which to output the articles for browsing.

We’ll start with the News List page in order to create the general structure we want and we’ll come back to it to add more functionality.

We will be replacing the following Ektron Smart Form as well as a Widget for rendering the news as a list.



Title: News

News

Headline:
Subhead:
Byline:
Source:
Publication Date:

Contact

Name:
Email:
Phone:

Release:

News Page

We can immediately eliminate the Publication Date field since Episerver allows you to update the Created property of the content (YAY!). We will keep the headline, since we may want the content Name and Alias to be different from this value.

Remember we're splitting the Contact information into a separate block, which also helps simplify the final content model.

Make the following additions / updates to **~/Models/Pages/NewsPage.cs**

```
using System.ComponentModel.DataAnnotations;
using EPiServer.Core;
using EPiServer.DataAbstraction;
using EPiServer.DataAnnotations;

namespace Brightfind.EktronToEpiserverLab.Models.Pages
{
    [ContentType(DisplayName = "NewsPage", GUID = "26592a6c-
a3f9-415d-9702-39ecd9d33806", Description = "", GroupName =
Global.GroupNames.Standard)]
    [AvailableContentTypes(
        Availability = Availability.Specific,
        IncludeOn = new[] { typeof(NewsListPage) })]
    public class NewsPage : SpockPageBase
    {
        [Display(GroupName = SystemTabNames.Content, Order = 100)]
        public virtual string Headline { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 200)]
        public virtual string Subhead { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 300)]
        public virtual string Byline { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 400)]
        public virtual XhtmlString ArticleBody { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 500)]
        public virtual ContentArea Contact { get; set; }
    }
}
```

Bonus: Can you figure out how to restrict the **Contact** ContentArea to only allow ContactInformationBlock types?

Add the following to **~/Views/NewsPage/index.cshtml**

```
<div class="row">
    <div class="col-md-6 col-md-offset-3">
        <h1 @Html.EditAttributes(m =>
m.CurrentPage.Headline)>@Model.CurrentPage.Headline</h1>
        @if (PageEditing.PageIsInEditMode || !
string.IsNullOrEmpty(Model.CurrentPage.Subhead))
        {
            <h3 @Html.EditAttributes(m =>
m.CurrentPage.Subhead)>@Model.CurrentPage.Subhead</h3>
        }
        <p><i>By: <span @Html.EditAttributes(m =>
m.CurrentPage.Byline)>@Model.CurrentPage.Byline</span></i></p>
        <p><i>Published: <span @Html.EditAttributes(m =>
m.CurrentPage.Created)>@Model.CurrentPage.Created.ToLongDateString()</span></
i></p>
        <hr />
        @Html.PropertyFor(m => m.CurrentPage.ArticleBody)
        @Html.PropertyFor(m => m.CurrentPage.Contact, new { tag =
EPiBootstrapArea.ContentAreaTags.HalfWidth })
    </div>
</div>
```

@if(PageEditing.PageIsInEditMode…) - We want to hide the markup for the Subhead where a value does not exist. Yet at the same time we want to make sure it's available for in-context editing whether it has a value or not. This simple condition statement allows us to show it when only when appropriate.

Because this page requires not special logic in the Controller, we can simply allow the DefaultPageController to handle it and we're done!

Contact Information Block

The Contact Information Block will be incredibly simple. Again, since we're sending the Block data directly to the View, there's no need for a Controller. Just the model and the markup!

Make the following updates to **~/Models/Blocks/ContactInformationBlock.cs**

```
using System.ComponentModel.DataAnnotations;
using EPiServer.DataAbstraction;
using EPiServer.DataAnnotations;

namespace Brightfind.EktronToEpiserverLab.Models.Blocks
{
    [ContentType(DisplayName = "Contact Information Block", GUID =
"8c163a81-1fb4-4d0b-88b0-c39654495513", Description = "", GroupName =
Global.GroupNames.Standard)]
    public class ContactInformationBlock : SpockBlockBase
    {
        [Display(GroupName = SystemTabNames.Content, Order = 100)]
        public virtual string Name { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 200)]
        public virtual string Email { get; set; }
        [Display(GroupName = SystemTabNames.Content, Order = 300)]
        public virtual string Phone { get; set; }
    }
}
```

AvailableContentTypes - By setting *Availability* to *Specific* and setting an *Include* value, we're indicating that only News types can be created as a child of our NewsList page. This is similar to configuring an Ektron folder to only allow a single Smart Form type.

It's possible to add validation and even input masks to these properties. Try adding a [Required] attribute to one of the properties above and see what happens! :)

Next, add the markup for rendering the Contact Information Block data to **~/Views/Shared/Blocks/ContactInformationBlock.cshtml**:

```
<fieldset>
    <legend>Contact</legend>
    <div>Name: <span @Html.EditAttributes(m => m.Name)>@Model.Name</span></div>
    <div>Email: <span @Html.EditAttributes(m => m.Email)>@Model.Email</span></
div>
    <div>Phone: <span @Html.EditAttributes(m => m.Phone)>@Model.Phone</span></
div>
</fieldset>
```

Done!

News List Page

Because this is a PageBuilder page in Ektron, there's no direct re-modeling to do in Episerver. In fact, even in Epi, we're going to leave it fairly simple. It's a placeholder page to engage the functionality we want as well as a means of enforcing correct structure in the page tree.

```
using EPiServer.DataAbstraction;
using EPiServer.DataAnnotations;

namespace Brightfind.EktronToEpiserverLab.Models.Pages
{
    [ContentType(DisplayName = "NewsListPage", GUID = "1d3ec6e5-ad13-47a8-
a5f2-4090518727b2", Description = "", GroupName = Global.GroupNames.Standard)]
    [AvailableContentTypes(
        Availability = Availability.Specific,
        Include = new[] { typeof(NewsPage) })]
    public class NewsListPage : SpockPageBase
    {

    }
}
```

AvailableContentTypes - By setting *Availability* to *Specific* and setting an *Include* value, we're indicating that only News types can be created as a child of our NewsList page. This is similar to configuring an Ektron folder to only allow a single Smart Form type.

News List Controller & ViewModel

Because we want special functionality with our news list page (a browsable list of news items), it will need its own controller as well as ViewModel. The ViewModel will inherit from a default (saving us some effort) and include properties necessary for pagination. The controller will populate the ViewModel with child News items and control which page of data is being shown.

View Model

Open the NewsListViewModel.cs file in **~/Models/ViewModels** and make the following changes.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Brightfind.EktronToEpiserverLab.Models.Pages;

namespace Brightfind.EktronToEpiserverLab.Models.ViewModels
{
    public class NewsListViewModel : PageViewModel<NewsListPage>
    {
        public IEnumerable<NewsPage> Articles { get; set; }

        public int TotalArticles { get; set; }
        public double TotalPages { get; set; }
        public int CurrentPageNumber { get; set; }

        public NewsListViewModel(NewsListPage currentPage) : base(currentPage)
        {
        }
    }
}
```

PageViewModel<NewsListPage> - Saves duplication of effort by building on what's already in place for a standard page.

IEnumerable<NewsPage> - We will populate this in the controller. This represents our list of Articles for rendering to the page.

TotalArticles, TotalPages, and CurrentPageNumber - Each aide in applying pagination to our list of articles.

Controller

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Brightfind.EktronToEpiserverLab.Models.Pages;
using Brightfind.EktronToEpiserverLab.Models.ViewModels;
using EPiServer;
using EPiServer.Core;
using EPiServer.Framework.DataAnnotations;
using EPiServer.Web.Mvc;
using EPiServer.ServiceLocation;

namespace Brightfind.EktronToEpiserverLab.Controllers
{
    public class NewsListPageController : PageController<NewsListPage>
    {
        public ActionResult Index(NewsListPage currentPage)
        {
            var pageSize = 5;

            var model = new NewsListViewModel(currentPage);

            var contentLoader =
ServiceLocator.Current.GetInstance<IContentLoader>();

            var newsItems =
contentLoader.GetChildren<NewsPage>(currentPage.ContentLink)
                .OrderByDescending(n => n.Created);

            model.TotalArticles = newsItems.Count();
            model.TotalPages = Math.Ceiling(model.TotalArticles /
(double)pageSize);

            var q = Request.QueryString;
            int p;
            if (string.IsNullOrEmpty(q["p"]) || !int.TryParse(q["p"], out p) ||
p <= 1)
            {
                model.CurrentPageNumber = 1;
                model.Articles = newsItems.Take(pageSize);
            }
            else
            {
                model.CurrentPageNumber = p;
                model.Articles = newsItems.Skip((p - 1) *
pageSize).Take(pageSize);
            }

            return View(model);
        }
    }
}
```

pageSize - This is currently being set in the controller, but we could readily make it configurable as part of the NewsListPage.

model - Creating an instance of our ViewModel. This prepares our data for configuring the child NewsPage items as well as gives us properties where we can set our pagination values.

contentLoader - Episerver's API for getting content. Episerver uses Dependency Injection and the ServiceLocator is the most common way of getting an instance of the IContentLoader.

newsItems - All child items of the current page that are of type NewsPage. Note the use of LINQ to order the items returned.

TotalArticles and TotalPages - Where we set the numbers for the total number of results and, based on our pageSize, the total number of pages.

The next section of code deals with pagination. Assuming a Querystring parameter of "p", we obtain the current page and simply skip and take as needed.

return View(model) - Finally, we pass the completed model to the View for rendering.

The controller needs to perform some special functions for the news list and its pagination.

News List View

Add the following code to **~/Views/NewsListPage/index.cshtml**

```
<div class="row">
    <div class="col-md-6 col-md-offset-3">
        <h1 @Html.EditAttributes(m =>
m.CurrentPage.Name)>@Model.CurrentPage.Name</h1>
        <hr />

        <div class="list-group">
            @foreach(var article in Model.Articles)
            {
                <a href="@Url.ContentUrl(article.ContentLink)"
title="@article.Name" class="list-group-item">
                    <h4 class="list-group-item-heading">@article.Headline</h4>
                    <p class="list-group-item-text">@article.TeaserText</p>
                    <p class="text-right"><span class="label label-
default">@article.Created.ToString("MM/dd/yyyy")</span></p>
                </a>
            }
        </div>
    </div>
</div>
```

@Model.CurrentPage.Name - The only value we're outputting from the NewsListPage itself is the name, which is being made editable in-context via the @Html.EditAttributes helper within the h1 tag.

@foreach - A much more convenient way of looping through a list of data than a ListView could ever hope to be! :)

News List View (part 2)

Just in case you thought we totally forgot about pagination. Ha!

Insert the following code after the <hr /> tag inside your NewsList index.cshtml (created above) to output pagination for your list.

```
<nav aria-label="page navigation">
    <ul class="pagination">
        @if(Model.CurrentPageNumber > 1)
        {
            <li>
                <a href="@Html.BuildUrl(Request.Url.AbsolutePath, new
KeyValuePair<string, object>("p", Model.CurrentPageNumber - 1))" aria-
label="Previous">
                    <span aria-hidden="true">&laquo;</span>
                </a>
            </li>
        }

        @for(int i = 1; i <= Model.TotalPages; i++)
        {
            <li class="@(Model.CurrentPageNumber == i ? "active" :
string.Empty)">
                <a href="@Html.BuildUrl(Request.Url.AbsolutePath, new
KeyValuePair<string, object>("p", i))">@i</a>
            </li>
        }

        @if (Model.CurrentPageNumber < Model.TotalPages)
        {
            <li>
                <a href="@Html.BuildUrl(Request.Url.AbsolutePath, new
KeyValuePair<string, object>("p", Model.CurrentPageNumber + 1))" aria-
label="Next">
                    <span aria-hidden="true">&raquo;</span>
                </a>
            </li>
        }
    </ul>
</nav>
```

Note: There are @if statements within the code that help us to hide the Previous or Next buttons whenever we are on the first or last page of the set, respectively. In the middle, we're looping through the available pages and designating the current page, whichever it may be, as the Active item in the list.

@Html.BuildUrl is not an Episerver function. It was created by Daved Artemik (@beenDaved on Twitter) and is included as part of this project source.

BUILD AND RUN

Run the site and create a NewsListPage under the Start page. Name it “News”. Then create a few NewsPage items under the News page. Navigate back to the News page and you should see your articles listed out and pagination in place.
http://lab.brightfind.com/News/

This process can be repeated for pages like Blog + BlogPost. The logic and code are nearly identical, so we won’t go into them as part of the LAB.

The rest of this page intentionally left blank. :-P

The…Calendar

By now you should be getting the sense that creating Page or Block types is both simple and, well, somewhat repetitive. Once you're adjusted to it, it's the sort of thing you can do very quickly without much need for reference material.

So for the Calendar section, I'd like to move on to something more interesting that may come in handy in your transition from Ektron to Episerver.

Note: The functionality in this section, **PropertyList**, is labeled as UNSTABLE, which means that we're using an internal API from Episerver. One they reserve the right to change without warning, since it's not really intended to be used by developers. *What this means* is that if you use these features, you'll need to be extra careful in your upgrades and investigate this functionality in particular.

That being said, the tools herein are being used by developers and in production sites. It's very convenient for in-content data entry for tabular data or other small, content-specific models, including dates and times for an event, for example. :)

There is no out-of-the-box equivalent in Episerver to Ektron's WebCalendar or event content. This isn't necessarily a bad thing as it means you get to implement *what you want* rather than relying on an out-of-box solution that is challenging or impossible to modify, should you need to.

In some cases, partners will recommend a 3rd party tool to support calendar events, registrations, and other related functions. However, if your needs in this area are simple enough, then you can readily accommodate Events within your Episerver site.

So instead of repeating the process of creating page types and renderings, we'll focus on the creation and configuration of a very specific property type.

Open **~/Models/Pages/EventPage.cs** and add this property for **Dates**.

```
[Display(GroupName = SystemTabNames.Content, Order = 300)]
public virtual IList<PropertyEventDate> Dates { get; set; }
```

This property is configured with a type (IList<PropertyEventDate>) that is not supported by Episerver out of the box. It will build, but you'll get runtime errors. What we need to do is add the proper code to enable Episerver to handle this type of content entry. Ready?

The Model

First, we need to create the type we want Episerver to manage as input. PropertyEventDate.

Open **~/Models/Properties/PropertyEventDate.cs** and add the following property to the class.

```
[Display(Name = "Repeats Every")]
[SelectOne(SelectionFactoryType = typeof(EnumSelectionFactory<RepeatOptions>))]
public string RepeatsEvery { get; set; }
```

Note that the properties in this model are NOT virtually declared. Moreover, pay close attention to the **[SelectOne]** attribute, which uses a SelectionFactory to inform Episerver of how to manage the values for this property. Keep that in mind - a property attribute can inform Episerver of how to present the authoring UI for this field. We'll be doing something similar for this very model.

The Base

I won't claim to know all of the inner workings here, but this works. We need to create some system by which Episerver knows how to interpret the object we just created. In effect, it needs some way to serialize and deserialize the data and we're going to help it along.

We're going to create a class that inherits from Episerver's own PropertyList<T> and add to it a serializer that can take our model and deserialize it from JSON back to an instance we can use in .NET code.

Open **~/Models/Properties/PropertyListBase.cs** and add the following code.

```
public class PropertyListBase<T> : PropertyList<T>
{
    private readonly IObjectSerializer _serializer;
    private Injected<ObjectSerializerFactory> _objectSerializerFactory;

    public PropertyListBase()
    {
        _serializer =
_objectSerializerFactory.Service.GetSerializer("application/json");
    }

    public override PropertyData ParseToObject(string value)
    {
        ParseToSelf(value);
        return this;
    }

    protected override T ParseItem(string value) =>
_serializer.Deserialize<T>(value);
}
```

IObjectSerializer - Is Episerver's own Serialization tool. We could as readily use something like Newtonsoft. This example keeps it pure to Episerver. This is instantiated within the constructor.

Injected<…> - This will return an instance of the requested type regardless of where we are in the application lifecycle. This is less commonly used than **ServiceLocator.Current.GetInstance**, but even more valuable in some contexts when the ServiceLocator is not yet initialized. It will feel odd, but this doesn't need any further assignment. It's ready to use as-is as long as you reference the method you want via **myobject.Service.Method**.

ParseToObject and ParseItem help to deserialize the incoming JSON and help the parent object return the expected type.

Using The Base

The Base class helps extend Episerver's functionality in a very general way, but it still doesn't know that such a base class should be used for our property.

Next we'll register our type with Episerver.

Open **~/Models/Properties/PropertyEventDateDefinition.cs** and add the following code:

```
using EPiServer.PlugIn;

namespace Brightfind.EktronToEpiserverLab.Models.Properties
{
    [PropertyDefinitionTypePlugIn]
    public class PropertyEventDateDefinition :
PropertyListBase<PropertyEventDate>
    {
    }
}
```

This merely informs Episerver that PropertyEventDate should be registered as a valid type and that it should be serialized and deserialized using the **PropertyListBase** class.

This could instead inherit **PropertyList** directly and apply the same overrides to this class. The abstraction into a Base class, however, helps us create similar functionality in other areas of the application with minimal effort.

The EditorDescriptor

Finally, we need to update our EventPage model and add an **EditorDescriptor** to the Dates property. This addition will inform Episerver that the authoring environment should treat this property as a specialized Collection of PropertyEventDate.

To do this, add the following attribute to the Dates property.

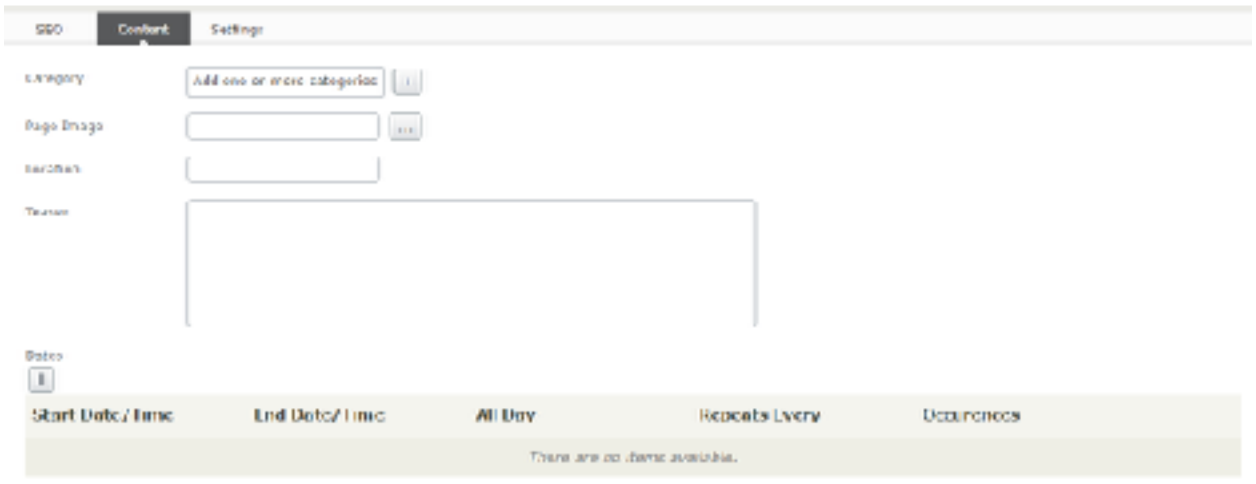
```
[Display(GroupName = SystemTabNames.Content, Order = 300)]
[EditorDescriptor(EditorDescriptorType =
typeof(CollectionEditorDescriptor<PropertyEventDate>))]
public virtual IList<PropertyEventDate> Dates { get; set; }
```

The ViewModel

In order to not have the site crash and burn for everything else we've done, I had to disable code within the ViewModel. Typically, you would not put much logic into the ViewModel, but in this case it seemed to fit. Open **~/Models/ViewModels/EventViewModel.cs** and uncomment all lines of code. These functions loop through assigned dates and generate the list of occurrences.

BUILD AND RUN

Run the site and create a CalendarPage under the Start page. Name it "Calendar". Then create a few EventPage items under the Calendar page. As you create Events, go to the All Properties view [ ] and you should see a view similar to below for entering dates. Awesome!


SEO
Content
Settings
Category
Add one or more categories
Page Image
Teaser
Dates
Start Date/Time
End Date/Time
All Day
Repeats Every
Occurrences
There are no items available.