Mastering

Orchard Layouts

Mastering Orchard Layouts takes the reader on a journey through Orchard Layouts, uncovering the ins and outs of this powerful module.

Mastering Orchard Series

IDeliverable { }

2016

Mastering Orchard Layouts

Written by Sipke Schoorstra – 2016

1st edition

Contents

Introduction

Welcome to *Mastering Orchard Layouts*, a book that will take us on a journey through the wonderful and exciting world of the Orchard Layouts module, which was first released as part of Orchard 1.9.

The book is divided in three parts.

Part 1 introduces the Layouts module and looks at it from a user's perspective.

Part 2 goes a step further and looks at the various shapes and templates from a theme developer's perspective.

Part 3 takes a deep dive and looks at the module from a developer's point of view. Here we'll learn about extensibility and APIs, and how to create custom elements and element harvesters.

I hope you will enjoy reading this book as much as I had writing it, and that the knowledge you gain is useful in your Orchard Projects.

1. Introduction to Layouts

With the release of Orchard 1.9 came a new module called *Orchard.Layouts*. Before we try it out to see what we can do with it, let's first give it some context to get a better understanding of why it was created in the first place, what problems it solves, and, equally important, what problems it *won't* solve.

Defining Layouts

Layouts are everywhere. You find them not only in newspapers, magazines, and the book you're currently reading, but you find them just about anywhere, like in the office and your living room. Cities, planets and the universe, they all have a layout.

Elements, objects and shapes that are placed in a particular position relative to each other, are said to be part of a layout. In other words, *a layout is an arrangement of elements*. That's a great definition for sure, but how does this relate to Orchard you ask? Let's find out.

Orchard and Layouts

Orchard, unsurprisingly enough, is about building and managing web sites that consist of web pages. Within the context of a webpage, a layout is the arrangement of visual elements on a page. Those visual elements can include things such the site's navigation, side bar and the site's content. The site's content itself, too, can have a layout. For example, two blocks of text that appear next to one another are said to be laid out horizontally.

And that is where the Layouts module comes in: it enables content editors to create layouts of contents.

Now, technically speaking, a theme developer can choose to set up their theme in such a way that the entire layout is controlled by the Layouts module. But, practically speaking, this is probably not the best use of the module in its current form. One reason is the fact that layouts are provided through a content part (the Layout Part), which means layouts created by the Layouts module can only be applied to content items. So although you could very well add a Menu element to a page content item, not all pages in Orchard are provided by content items.

For example, the Login screen is provided by a controller. If the site's main navigation is implemented as a layout element, the main navigation would disappear as soon as a page is displayed by something other than a content item.

As the Layouts module evolves over time, new site editing paradigms may using the Layouts module emerge, but until then, we will focus on how to use the Layouts module from a content editor's perspective. Which, as you'll see, is quite impressive.

Layout and Elements

When you enable the Layouts feature provided by *Orchard.Layouts*, a new content part called *Layout Part* is added to the list of available content parts, and is attached to the *Page* content type by default. It is this Layout Part that enables content editors to visually arrange elements on a canvas, effectively enabling them to create layouts of contents.

These elements are a new type of entity in Orchard and represent the objects you can place on a canvas. An example of these elements is the Html element, which enables the user to add content. Another example is the Grid element, which enables the user to create a layout by adding Row and Column elements to it. The Column element is a container element into which you can add other elements, such as Html and Image elements. You can imagine that using these elements, it is easy to create a layout of contents. It is this capability that gave Orchard.Layouts its name.

When to use Orchard.Layouts?

From what you have read so far, the answer to the question of when to use the Layouts module may seem obvious: whenever you need to create a layout of content, use the Layouts module. But you may be wondering that surely, this was possible before we had this module? Well, yes, but that was a very hard thing to do. Let me explain.

Let's say we have a web page with content that consists of two paragraphs as seen in figure 1.1.

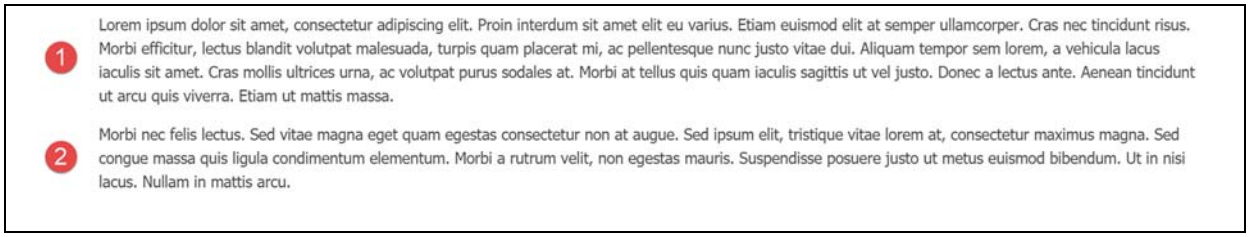

Figure 1-1 – Two paragraphs, vertically stacked.

Now, let's say that we want to display those two paragraphs laid out horizontally instead, as seen in figure 1.2.

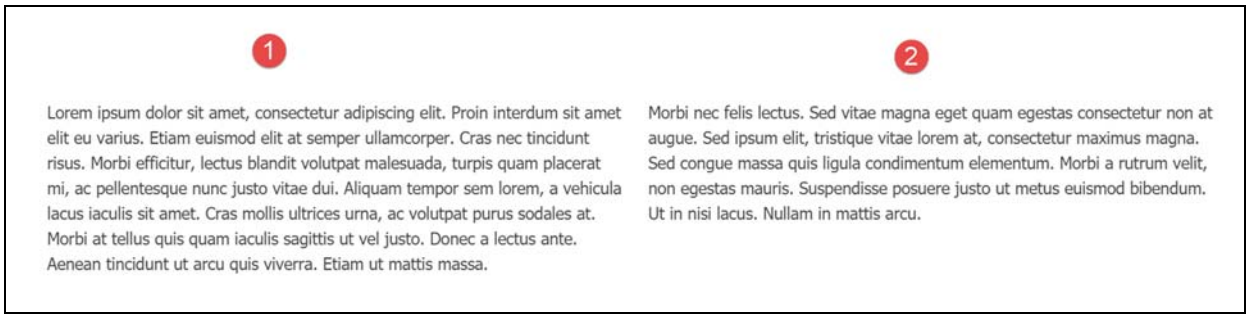

Figure 1-2 – The same two paragraphs, horizontally laid out.

Before we see how to achieve that with the layouts module, let's explore our options *before* Orchard.Layouts.

Option 1 - Direct Html manipulation

One option is to edit the Html source of the Body Part content and leverage your Html skills by adding an Html table element, or maybe even using Bootstrap's Grid CSS classes and apply them on elements. Although that would certainly work, it is far from ideal, because it would require the content editor to know about Html tables and how to work with them. For a simple two-column layout for a body of text this may not be that big of a deal, but it becomes icky real fast when working with more complex layouts.

Option 2 - Widgets and Zones

Another option is to provide two zones, let's say *AsideFirst* and *AsideSecond*. The theme's Layout.cshtml view renders these zones horizontally. You would then simply add an Html Widget to both zones, and the two Html widgets would appear next to each other. Although this approach works, a major disadvantage is that now the textual content becomes unrelated to the content item itself, since you are using widgets. To manage the content on this page, you have to go to the Widgets screen, create a page specific layer, and add two widgets. Now imagine you have to do that for 20 pages.

That means 20 widget layers, 2 Html widgets per layer, and 20 Page content items with no contents. And this is just two columns. Imagine you have other types of layouts, for example one row with two columns, another row with 4 columns, and perhaps rows with one column taking up 2/3 of the row and a second column 1/3 of the row. Crazy. Allowing this level of freedom to the content editor user would easily end up in a maintenance nightmare.

> There is a way to associate widgets with content items directly by taking advantage of a free gallery module called *IDeliverable.Widgets*. Although this is better than having to create a layer per page, it is still not ideal.

Option 3 - Content Fields and Placement.info

Yet another option is to create various content types, where a content type would have multiple content fields.

For example, we could create a new content type called TwoColumnPage with two TextField fields. The theme would use *Placement.info* to place each field into two horizontally laid out zones.

Although this option is (arguably) better than the previous option using widgets, there is still the limitation of freedom when you want to introduce additional layouts. Not to mention the fact that we're now basing the content type name on what it looks like, rather than its semantic meaning. It is not pretty.

Enter Orchard.Layouts

With the inclusion of the Layouts module, a fourth option appeared. And a much better one too!

With Orchard.Layouts, creating a two-column layout could not be simpler. Simply add a Grid element with a single Row and two Column elements to the canvas, add some content elements, and you're done. No need for Html editing, no additional zones, no widgets and layers, and no additional content types.

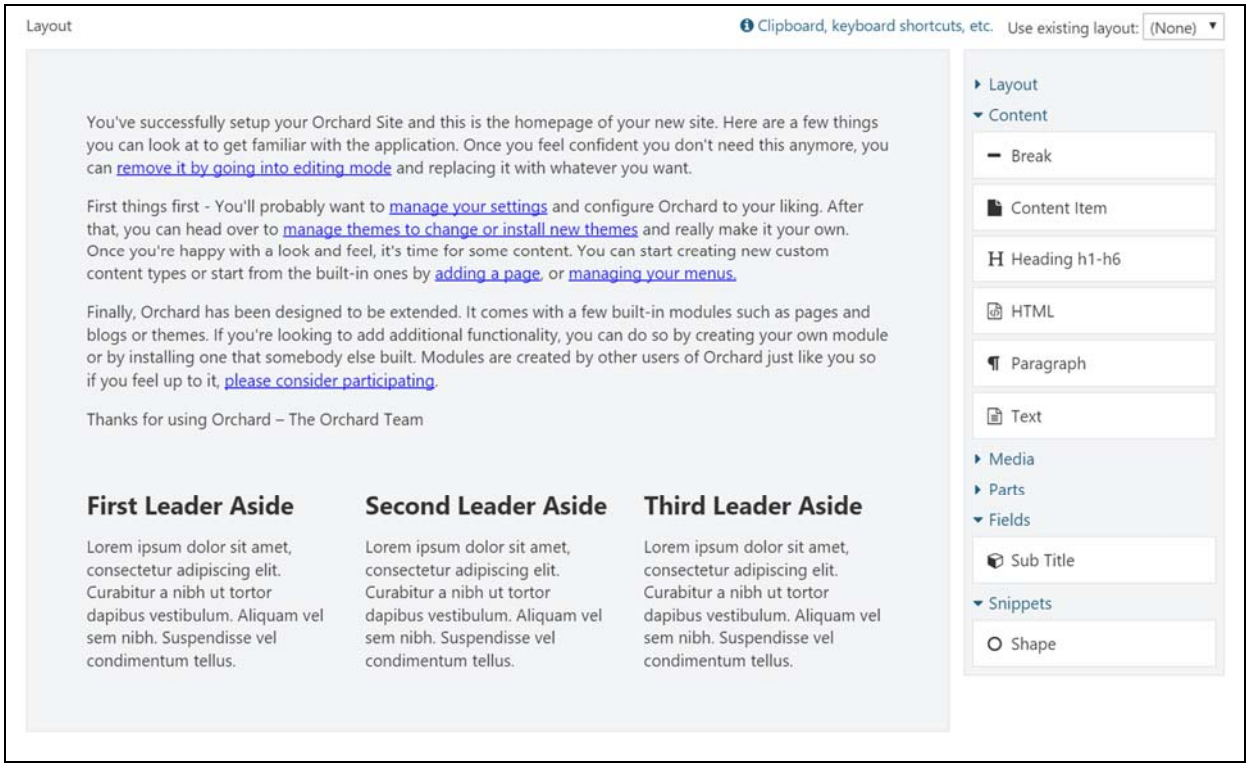

Figure 1-3- The Layout Editor.

The layout editor consists of a design surface called the *canvas* and a toolbox containing elements that the user can drag and drop onto the canvas.

To sum it up, thanks to the Layouts module,

- It is no longer necessary to create page-specific layers and widgets to achieve complex layouts of contents.
- It is no longer necessary to create specific content types just for supporting multiple layouts.
- We have an easy way to create various layouts of content.

Where did the Body Part go?

When you install Orchard 1.9 or later for the first time and have a look at the Page content type, you will notice that it doesn't have the Body Part anymore. Instead, you will see the new Layout Part attached. However, the Body Part is still a happy citizen within the Orchard, and will remain as such. The Layout Part simply serves a different purpose, namely to enable the user to layout pieces of contents. The Body Part is great when all you need is an editable body of text. Blog Posts are a great example where I would rather use the Body Part instead of the Layout Part, because all I want to do there is simply start writing content without having to first add an Html element to the canvas. In the end, it's all about choice and being able to pick the right tool for the job.

What happens to my existing site and its contents when upgrading to Orchard 1.9?

If you're worried about your existing content, fear not. When you upgrade your site to 1.9 or beyond, the Layouts feature will not be automatically enabled. And even when you enable the feature yourself, it will not change

your content type definitions. If you *do* want to use Layouts on existing Orchard installations that have been upgraded to the latest codebase, you will simply have to enable the Layouts feature and attach the Layout Part manually.

The Nature of Elements

With the Layouts module came a new type of entity called *Element*. Unlike Widgets, Elements are not content items, but are, quite simply, instances of the *Element* class. Elements can contain other elements, and this is how layouts emerge.

The hierarchy of elements are stored using the *infoset* storage part of the content item, implemented via the Layout Part. This means that whenever a content item is loaded with the Layout Part attached, the elements are loaded all at once, unlike Widgets, where each widget is loaded individually.

Similar to content items, or more accurately, content parts, Elements have their own drivers, which decouples element data from element behavior. This pattern is borrowed from the content part and content field system that also leverage drivers.

Does Orchard.Layouts work with grid systems such as Bootstrap?

Many websites today use CSS grid frameworks such as Bootstrap. These grid systems enable web designers to layout visual components onto a grid

that is made up of rows and columns. So, you may be wondering whether the Layouts module plays nice with such grid systems. As it turns out, this scenario is well-supported. The Grid, Row and Column elements map nicely to Bootstrap's **container**, **row** and **col-md-*** CSS classes. You will have to override the shape templates for these elements in your theme so you can modify the CSS classes to use. We'll look into this in detail in chapter 10.

Summary

In this chapter, I introduced you to the new Layouts module, what it is for and why we need it.

Orchard.Layouts enables users to arrange elements of various types onto a canvas.

We explored what problem the Layouts module solves and when to use it. Where we had to resort to rather cumbersome solutions before, Layouts makes it a breeze to create all sorts of content layouts.

In the next chapter, we'll have a closer look at Orchard.Layouts from a user's perspective, and see how to actually use it.

2. First Look

In this chapter, we'll take a tour through the Layouts module and see how it works from a user's perspective.

The Main Players

First off, let’s go over some of the main concepts that are provided by the Layouts module and what their role is.

The Layouts Feature

When you set up a new Orchard 1.9 or later installation with the *Default* recipe, the Layouts feature will be enabled by default. Enabling this feature will cause a new part called Layout Part to be made available.

As mentioned before, one notable difference between Orchard 1.9 and previous versions is that the Page content type will have the Layout Part attached instead of the Body Part.

The Layout Part

It is this Layout Part that we are interested in. It provides a layout editor consisting of a canvas and a toolbar with available elements that the user can add to the canvas.

Elements

Elements are a new concept in Orchard. They are visual components that contain data and provide behavior. Elements can contain other elements, which is how you can create layouts, as we'll see shortly.

Out of the box, there are currently seven categories of elements:

- Layout
- Content
- Media
- Parts
- Fields
- Snippets
- UI

It's all lovely stuff, and we'll get to know all of the available elements in the next chapter.

The Layout Editor

The Layout Editor is the component that enables the user to add elements to a canvas, using the Grid, Row and Column elements to create layouts.

The editor consists of two main sections: the *canvas* (1) and the *toolbox* (2).

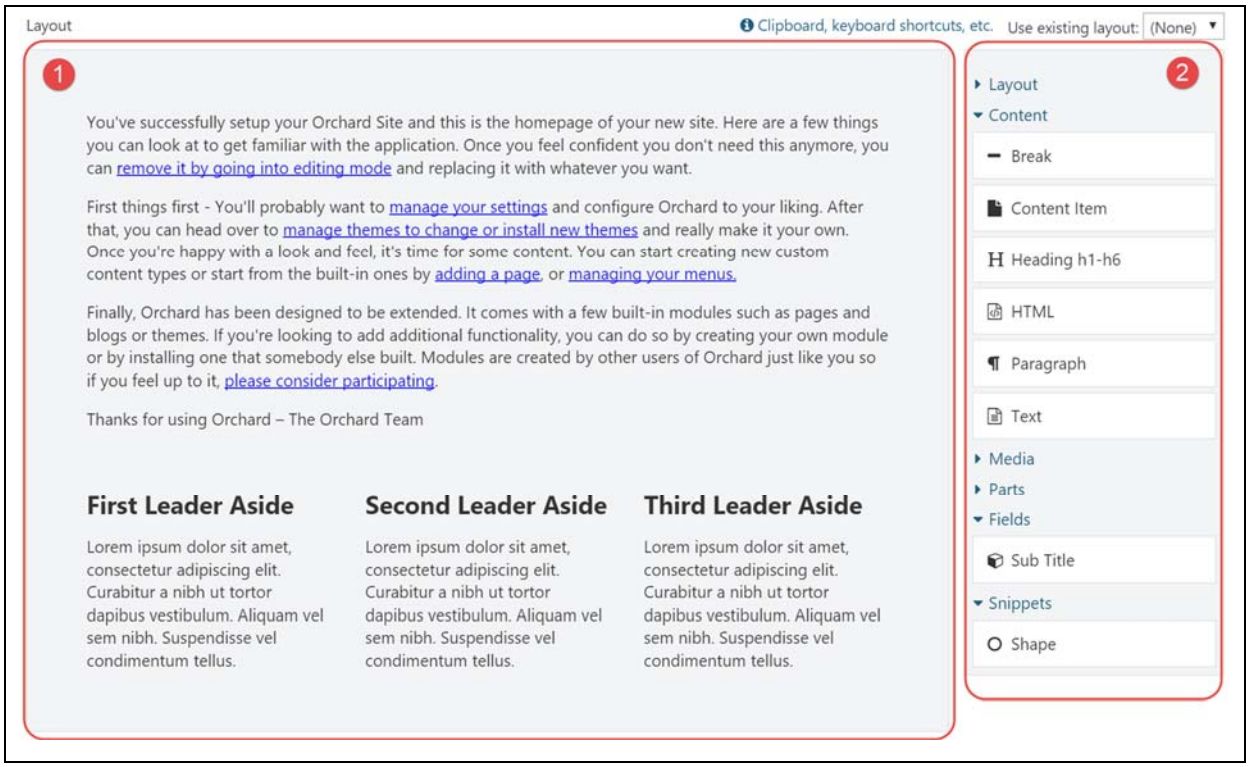

Figure 2-1 – The Layout Editor consists of the canvas (1) and the toolbox (2).

The canvas is the area onto which you place elements that are available from the toolbox.

> The canvas itself is an element of type Canvas, and is the root of the tree of elements.

The toolbox is a repository of all available elements in the system, grouped per category. Elements are bound to Orchard features, which means that other modules can provide additional element types.

The user places elements from the toolbox onto the surface by using drag & drop. If the selected element has an editor associated with it, a dialog window presenting the element's properties will appear immediately when it's dropped onto the canvas.

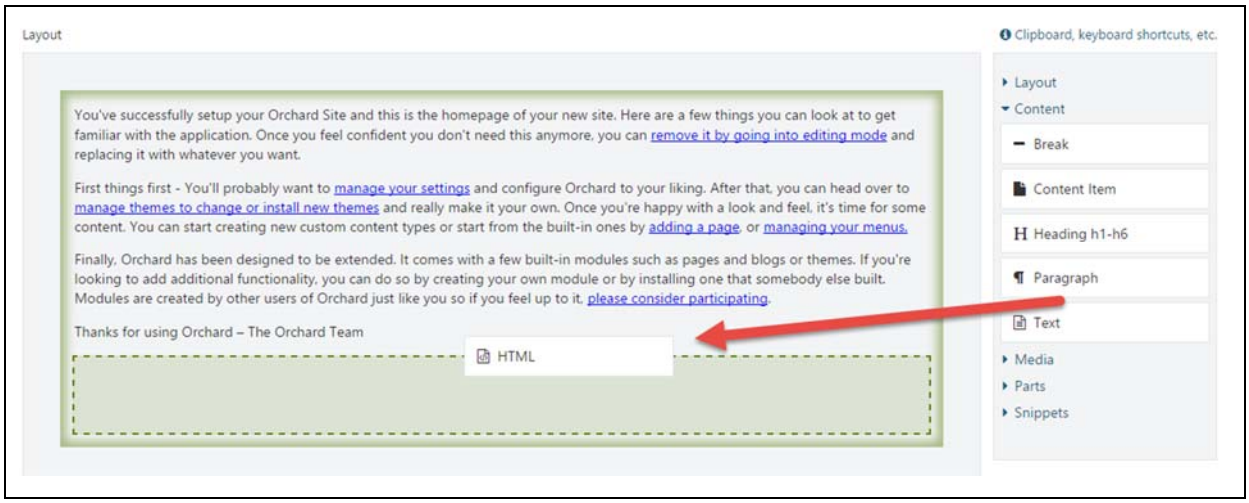

Figure 2-2 – The user drags and drops elements from the toolbox to the canvas.

Working with the Layout Editor

Let's have a look at the various ways we can interact with the layout editor and the elements.

Element Editor Controls

Depending on the element being selected, the user can perform certain operations on that element. These operations are represented as little icons as part of a mini toolbar that becomes visible when an element is selected. Common operations are *Edit*, *Edit Properties*, *Delete*, *Move up* and *Move down*. More specific operations are *Distribute columns* evenly and *Split column*, which apply to Row and Column elements, respectively.

Figure 2-3 - Each element has a toolbar to control properties of and perform operations to the element.

The button with the </> icon is probably the most-commonly used one, as it launches a dialog window that enables the user to configure the element. The icon right next to it ( ) provides a drop down menu with a list of configurable properties common to all elements. These properties are:

- Html ID
- CSS Classes
- CSS Styles
- Visibility Rule

The first three properties are rendered onto the Html tags when an element is rendered. The Visibility Rule determines whether or not the element should be displayed at all. I will have more to say about Visibility Rules in chapter 8.

The following table lists the complete set of keyboard shortcuts.

Icon	Shortcut	Description
	Enter	Launches the element specific editor dialog.
	Space	Displays an inline popup window with properties common to all elements.
	Del	Deletes the selected element.
	Ctrl + Up	Moves the element up. Alternatively, use drag & drop to change the position within the current container.
	Ctrl + Down	Moves the element down. Alternatively, use drag & drop to change the position within the current container.
		Distributes the columns of the selected row evenly.
		Splits the selected column into two.
	Alt + Left	Decreases the column offset by one.
	Alt + Right	Increases the column offset by one.

Keyboard Support

In addition to the keyboard shortcuts listed in the table above, there is also keyboard support for doing things like *copy*, *cut*, *paste*, and navigating around the hierarchy of elements on the canvas.

Although the layout editor provides a link to a small pop-up window listing all of the available keyboard shortcuts, I included a complete reference here:

Clipboard	
Ctrl + X / ⌘ + X	Cuts the selected element.
Ctrl + C / ⌘ + C	Copies the selected element.
Ctrl + V / ⌘ + V	Pastes the copied element into the selected container element.
Resizing Columns	
Alt + Left	Moves the left edge of the focused column left.
Alt + Right	Moves the left edge of the focused column right.

Shift + left	Moves the right edge of the focused column left.
Shift + Right	Moves the right edge of the focused column right.
The Alt and Shift keys can also be combined to move both edges simultaneously.	
Focus	
Up	Moves focus to the previous element (above)
Down	Moves focus to the next element (below).
Left	Moves focus to the previous column (left).
Right	Moves focus to the next column (right).
Alt + Up	Moves focus to the parent element.
Alt + Down	Moves focus to the first child element

Editing	
Enter	Opens the content editor of the selected element.
Space	Opens the properties popup of the selected element.
Esc	Closes the properties popup of the selected element.
Del	Deletes the selected element.
Moving	
Ctrl + Up / ⌘ + Up	Moves the selected element up.
Ctrl + Down / ⌘ + Down	Moves the selected element down.
Ctrl + Left / ⌘ + Left	Moves the selected element left.
Ctrl + Right / ⌘ + Right	Moves the selected element right.

Moving Elements within its Container

Once an element is placed on the canvas, its position can be changed within its container using drag & drop or using the Ctrl + arrow keys.

Moving Elements across Containers

At the time of this writing, it is not possible to move an element to another container using drag & drop. Instead, you will have to use the Cut/Paste keyboard shortcuts (*Ctrl+X* and *Ctrl+V*) to move an element from its current container to another one.

Resizing Columns

Column elements can be resized by dragging their left and right edges. When you re-size a column, its adjacent column will be re-sized as well. If you want to re-size a column and introduce an offset (basically "detaching" the column from its neighbor), press the *Alt* key while dragging the edges. It works pretty slick, try it out.

Layouts on the Front-end

Enabling the user to create and manage layouts from the back-end is only one half of the story of course. The other half is getting that layout out on the screen on the front-end. To accomplish this, the Layout Part driver simply invokes the driver of each element to build a shape. The resulting shape is a hierarchy of element shapes, ready for display on the front-end.

Each element is responsible for providing its own shape template. Container elements' shape templates render each of their child elements.

We'll learn how to take over the default rendering of elements in chapter 10.

Summary

In this chapter, I provided a high level overview of what the Layouts module is all about. At its core, it is about the user being able to add elements to a canvas.

Although that may sound pretty mundane, it is actually a very powerful feature that unlocks a host of new possibilities to the user. We will explore this in the rest of this book.

3. Meet the Elements

In this chapter, we'll go over all of the available elements in the default Orchard distribution. Most elements should be self-explanatory to use, but I think that some of them could use a little bit of a background to get a decent understanding on how to use them.

Elements are grouped by their category, so let’s go over them first.

Element Categories

The list of elements as well as categories are completely extensible of course, but by default Orchard comes with the following categories:

- Layout
- Content
- Media
- Parts
- Fields
- Snippets
- UI

Custom modules can provide additional categories, or associate custom elements with existing categories.

What follows next is a complete list of available elements when all features (except the features from *Orchard.DynamicForms*) are enabled:

12. Writing Custom Elements

Elements are at the heart of the Layouts module, and in this chapter, we will see how we can write our own.

Writing custom elements typically involve the following steps:

1. Create a class that derives from **Element** or any of its child classes. The only required member that needs to be implemented is the abstract **Category** property.
2. Create a class that derives from **ElementDriver<T>**, where **T** is your element's class. This class does not require any members, but needs to be there in order for your element type to be discoverable by the **TypedElementHarvester**. We'll look into element harvesters later on.
3. Although not strictly required, create a Razor shape template for your element for the *"Detail"* display type. If you don't provide a template, a default one will be used which simply displays the name of the element.
4. Also not strictly required, create a *design-time* view for your element for the *"Design"* display type. If you don't provide a design-time view, the view for the "Detail" display type will be used. This view is used when your element is rendered by the layout editor.

The Element Class

Elements are instances of .NET types that ultimately inherit from the **Element** abstract base class which lives in the

Orchard.Layouts.Framework.Elements namespace. Sub classes can add properties to their type definition, and of course override inherited properties such as **Category**.

When implementing properties on a custom element, you need implement them in a certain way so that the values will be persisted. This works much the same way like custom content parts, as you'll see shortly.

Element Drivers

While element classes provide information about the element, *element drivers* provide an element's behavior. This is similar in concept to the way content part drivers and content field drivers work. Drivers handle things such displaying and editing.

If you want, you can write *more than one element driver* for a given element type. As we'll see later in this book, this is key to extending existing elements with additional settings and behavior.

The following table lists all of the protected and virtual members of the base element driver that you can override in your own implementation:

Member	**Description**
Priority	A way for drivers to influence the order in which they are executed. This is useful in rare cases

	where you implement additional drivers for the same element type.
OnBuildEditor	Override this method when your element has an editor UI
OnUpdateEditor	Override this method to handle post-backs of your editor UI created by **OnBuildEditor**.
OnCreatingDisplay	Override this method to cancel the display of your element based on certain conditions. An example using this method is the **NotificationsElementDriver**, which prevents its element from being rendered when there are no notifications.
OnDisplaying	Override this method to provide additional information to the ElementShape that has been created for your element. Typical use cases are drivers that query or calculate additional information and add that to the ElementShape, which is provided by the context argument.
OnDisplayed	Override this method to provide additional information to the **ElementShape** after the Displaying event has been invoked. A good example of a use case for this is the

	RowElementDriver, which needs to determine whether or not it should be collapsed based on its child Column elements.
OnLayoutSaving	Override this method to perform some additional actions just before the element is being serialized. Nothing out of box is currently using this, so I have no good example of when you might want to use this. But, it's there if you need it.
OnRemoving	Override this method if you need to perform cleanup when your element is being removed. A good example are elements that create content items and are in charge of managing their lifetime. If the element is removed, you probably also want to remove the content item. The Removing event is invoked whenever an element was removed from the layout and the Layout Part is being saved, but also when a content item with the Layout Part is being removed.
OnExporting	Override this method if you need to add additional data when your element is being exported. A common example is the case where your element references a content item. In such cases, you'll need to export the identity of the content item, since you cannot rely on the content item ID itself

	remaining the same when the content item is being imported again.
OnExported	At the moment of this writing, the Exported event is triggered right after the Exporting event is triggered, so in practice there is no difference between the two. However, this may change in the future. For now, I recommend just sticking with the Exporting event when you need to provide additional information that you want to export.
OnImporting	Override this method when you have additional data to process when your element is being imported. For example, if you exported a content item's identity value, here is where you read back that information and get your hands on the content item in question, get its ID value and update your reference to that content item. A good example is the **ContentItemElementDriver**, which exports and imports a list of referenced content items.
OnImported	At the moment of this writing, the Imported event is triggered right after the Importing event is triggered, so in practice there is no difference between the two. However, this may change in the future. For now, I recommend just sticking with

	the Importing event when providing additional information.
OnImportCompleted	This event is triggered after the Importing/Imported events have been invoked on all content items and elements. Override this method when you need to update your element with additional referenced data, which may be available only after all other content items have been imported. A good example is the **ProjectionElementDriver**, which relies on the Query content items to be fully imported, since Query content items need to have their Layout records imported first before they are available to projections referencing those layout records.
Editor	Use the **Editor** method from **OnBuildEditor/OnUpdateEditor** to create an **EditorResult**. An EditorResult provides a list of *Editor Shapes* to be rendered. Although you could construct an EditorResult yourself, the advantage of using the Editor method is that it takes care of setting the **Metadata.Position** property on each editor shape, which is required for your editor shapes to become visible in the element editor dialog.

Element Data Storage

When implementing properties on your custom Element class, you'll probably want the information to be persisted when the element instance itself is persisted. To do so, all you need to do is store that information into the **Data** dictionary that each element inherits from the base **Element** class.

The following is an example implementation of a property:

```
public string MyProperty {
    get { return Data.ContainsKey("MyProperty") ? Data["MyProperty"] : null; }
    set { Data["MyProperty"] = value; }
}
```

The Data dictionary only stores string values, so if your property uses any other type, you'll have to convert it to and from a string value.

Fortunately, there is a nice helper class called **XmlHelper** in the **Orchard.ContentManagement** namespace that can help with that. For example, imagine implementing a property of type **Int32**:

```
public int MyProperty {
    get { return Data.ContainsKey("MyProperty") ?
XmlConvert.Parse<int>(Data["MyProperty"]) : 0; }
    set { Data["MyProperty"] = value.ToString(); }
}
```

Although pretty straightforward, let's see if we can simplify the property implementation a bit. For example, the check for the existence of a dictionary key in each and every property getter is pretty repetitive. As it turns out, there is a nice little extension method called **Get** in the **Orchard.Layouts.Helpers** namespace, which we can use as follows:

```
public int MyProperty {
    get { return XmlConvert.Parse<int>(Data.Get("MyProperty")); }
    set { Data["MyProperty"] = value.ToString(); }
}
```

The **XmlHelper.Parse<T>** returns a **default(T)** in case we pass in a null string, we don't have to worry about null checking ourselves.

But we can do even better. Instead of working with magic string values as the dictionary keys, we can implement our properties using strongly-typed expressions using the **Retrieve** and **Store** extension methods, which also live in the **Orchard.Layouts.Helpers** namespace. This is how to use them:

```
public int MyProperty {
    get { return this.Retrieve(x => x.MyProperty); }
    set { this.Store(x => x.MyProperty, value); }
}
```

Much better! The extension methods take care of null-checking as well as the string parsing.

So far we have seen how to store primitive types such as integers and strings. But what if you wanted to store complex objects? Unfortunately, the **Store** and **Retrieve** methods don't support that.

However, since the Data property is of type **ElementDataDictionary**, we can take advantage of its **GetModel** method, which uses model binding under the covers.

For example, let's say we have the following complex type:

```
public class MyElementSettings {
    public int MyNumber { get; set; }
    public string MyAddress { get; set; }
}
```

Implementing an element property of that type would look like this:

```
public MyElementSettings MyProperty {
    get { return Data.GetModel<MyElementSettings>(""); }
    set {
        Data["MyNumber"] = value?.MyNumber;
        Data["MyAddress"] = value?.MyAddress;
    }
}
```

Unfortunately there's currently no convenient method we can use to serialize the complex type back into a string, so you'll have to do that manually as shown above.

Trying it out: Creating a Map Element

If you ever browsed through the online Orchard Documentation, you've undoubtedly run across the "Writing a content part" tutorial (http://docs.orchardproject.net/Documentation/Writing-a-content-part).

In that tutorial, the author demonstrates writing a custom content part called **MapPart**. It is the first tutorial I ever followed when learning Orchard, and I thought it would be kind of cool if I could write a similar tutorial but for writing a custom element.

So that's exactly what we will do next: write a custom element called Map. Now, I won't go through the process of generating a new module, of which I'm sure you've done that before. If not, the "Writing a content part" tutorial explains the process in detail.