

Contents

1. Introduction to Layouts.....	12
Defining Layouts	12
Orchard and Layouts.....	12
Layout and Elements	13
When to use Orchard.Layouts?	14
Option 1 - Direct HTML manipulation	15
Option 2 - Widgets and Zones	15
Option 3 - Content Fields and Placement.info	16
Enter Orchard.Layouts.....	17
Where did the Body Part go?.....	18
What happens to my existing site and its contents when upgrading to Orchard 1.9?	18
The Nature of Elements.....	19
Does Orchard.Layouts work with grid systems such as Bootstrap?.....	19
Summary.....	20
2. First Look	21
The Main Players.....	21

The Layouts Feature	21
The Layout Part.....	21
Elements	21
The Layout Editor	22
Working with the Layout Editor	24
Element Editor Controls	24
Keyboard Support	27
Moving Elements within its Container	30
Moving Elements across Containers	30
Re-sizing Columns	30
Layouts on the Front-end	30
Summary.....	31
3. Meet the Elements.....	32
Element Categories	32
Layout.....	34
Grid.....	34
Row.....	35
Column.....	36
Canvas	37

Content.....	38
Break.....	38
Content Item	38
Heading.....	39
Html	39
Markdown.....	39
Paragraph	40
Projection.....	40
Text	40
Media	41
Image	41
Media Item	41
Vector Image	42
Parts.....	42
Placeable Parts	42
Fields	43
Snippets.....	43
Shape.....	43
Snippet Elements.....	44

UI.....	44
Widgets.....	46
Summary.....	46
4. Layout Templates.....	47
Sealed Elements and Placeholder Containers	48
Summary.....	48
5. Element Blueprints.....	50
When to use Element Blueprints.....	50
Creating Element Blueprints	50
Trying it out: Creating an Element Blueprint	51
Summary.....	54
6. Elements as Widgets.....	55
Why Elements as Widgets?	55
Using Elements as Widgets.....	56
Element Wrapper Part.....	56
Trying it out: Creating a Widget based on an Element	56
Existing Widgets based on Elements	58
Summary.....	58
7. Element Tokens	59

The Element.Display Token	59
Trying it out: Using Element.Display	60
Summary	62
8. Element Rules	63
Available Functions	63
Applying Element Rules	64
Summary	65
9. Theming	66
A Primer on Shapes	66
Anatomy of a Shape	67
Shape Templates	68
Elements and Shapes	70
Overriding Element Shape Templates	72
Custom Alternates	73
Trying it out: Creating Custom Alternates for Element Shapes ..	73
Content Part and Field Elements	75
Updating Placement.info	76
Summary	77
10. Bootstrap	78

Overriding Element Shape Templates	78
Responsive Layouts.....	81
Trying it out: Creating Responsive Layouts with Bootstrap	82
Summary.....	84
11. Snippets	85
Parameterized Snippets.....	85
Trying it out: Parameterized Snippets	86
Custom Field Editors	88
Summary.....	89
12. Writing Custom Elements	90
The Element Class.....	90
Element Drivers.....	91
Element Data Storage	96
Trying it out: Creating a Map Element.....	98
The Map Element.....	99
The Map Element Driver.....	99
The Map Element Editor Template.....	100
The Map Element Template	101
Element Editors & the Forms API	102

Trying it out: Using the Forms API	103
Descriptions and Toolbox Icons	106
Summary	107
13. Writing Container Elements	108
Steps to Create a Container Element.....	108
Layout Model Mappers	109
Client Side Support	112
Trying it out: Writing a Tile element.....	112
The Tile Element.....	113
The Tile Driver	114
The Tile Element Shape Template	122
The Client Side	123
Client-Side Assets.....	123
The Tile Model Map.....	136
Test Drive	136
Updating TileModelMap	138
Updating Tile/Model.js.....	139
Updating Tile/Directive.js	145
Updating LayoutEditor.Template.Tile.cshtml	146

Summary	148
14. Element Harvesters	149
Element Descriptors	150
Element Harvesters Out of the Box.....	150
IElementHarvester	152
Trying it out: Writing a Custom Element Harvester	154
Step 1: The UserProfilePart.....	154
Step 2: The Element Harvester	155
Improvements.....	161
Summary	165
15. Extending Existing Elements	166
Using Multiple Drivers	166
Using Multiple Handlers	167
Trying It Out: Extending Elements	167
Creating the CommonElementDriver	168
Implementing the FadeIn Behavior.....	170
An Alternative Implementation	173
Summary	177
16. Layout and Element APIs.....	178

Managing Layouts and Elements	179
The Layout Manager	179
The Element Manager	183
Element Events.....	186
IElementEventHandler.....	187
Displaying Elements	187
IElementDisplay.....	187
The Layout Editor.....	188
ILayoutEditorFactory	188
Serialization.....	189
ILayoutSerializer.....	189
IElementSerializer	190
Trying it out: Working with the APIs	191
Creating the Controller	192
Creating Elements	192
Element Serialization	197
Working with the Layout Editor	200
Summary.....	207
17. Writing a SlideShow Element	208

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 will find here 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 setup 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.

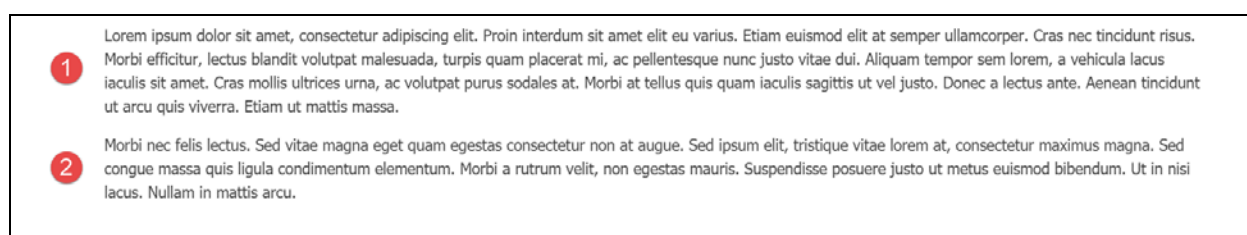


Figure 1-1 – Two paragraphs, vertically stacked.

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

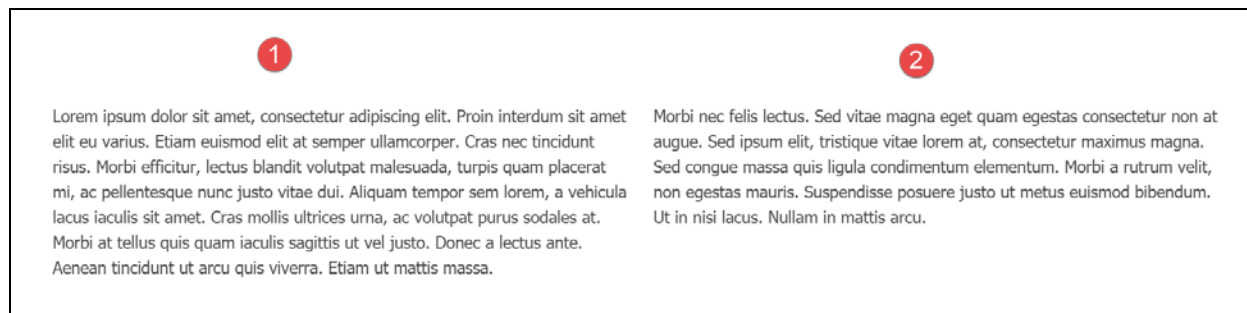


Figure 1-2 - The same two paragraphs, horizontally laid out.

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 `<div>` 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*, which allows you to associate widgets with your content items directly. 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.

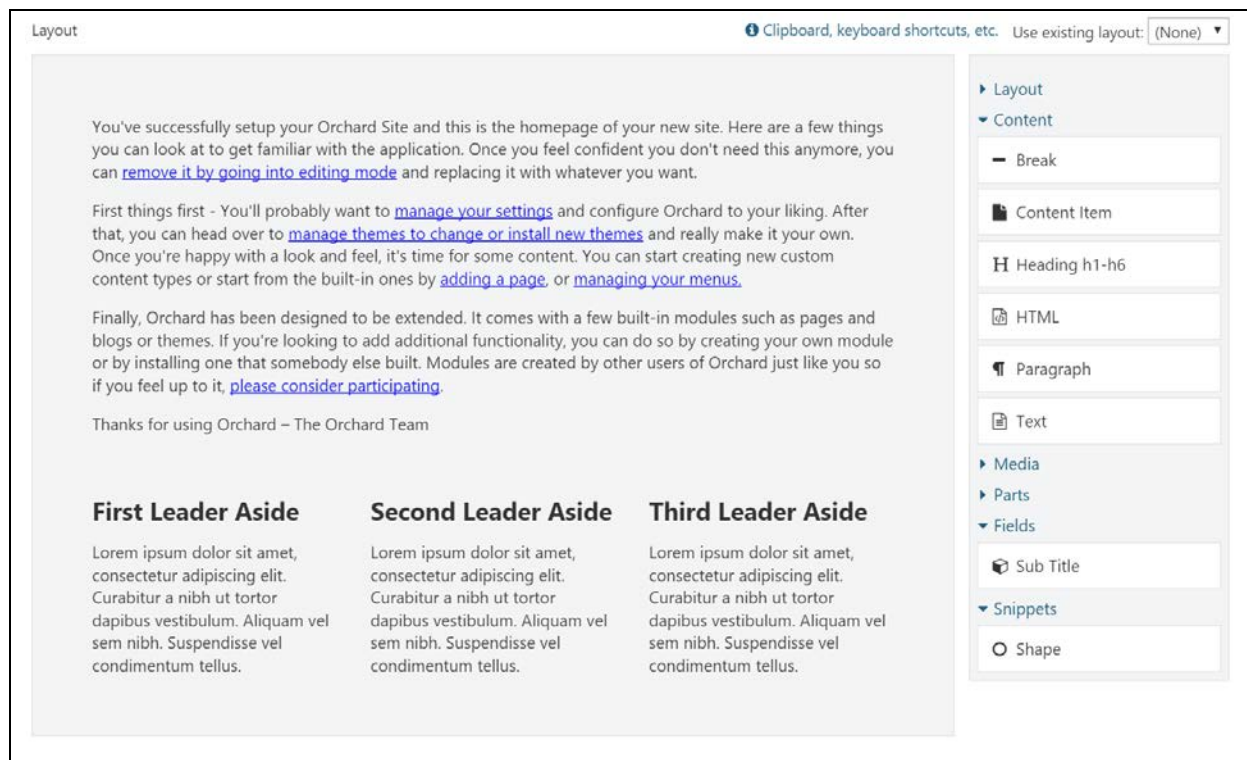


Figure 1-3- The Layout Editor.

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, we:

- No longer necessary to create page-specific layers and widgets to achieve complex layouts of contents.
- No longer necessary to create specific content types just for supporting multiple layouts.
- 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 setup 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 noticeable 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).

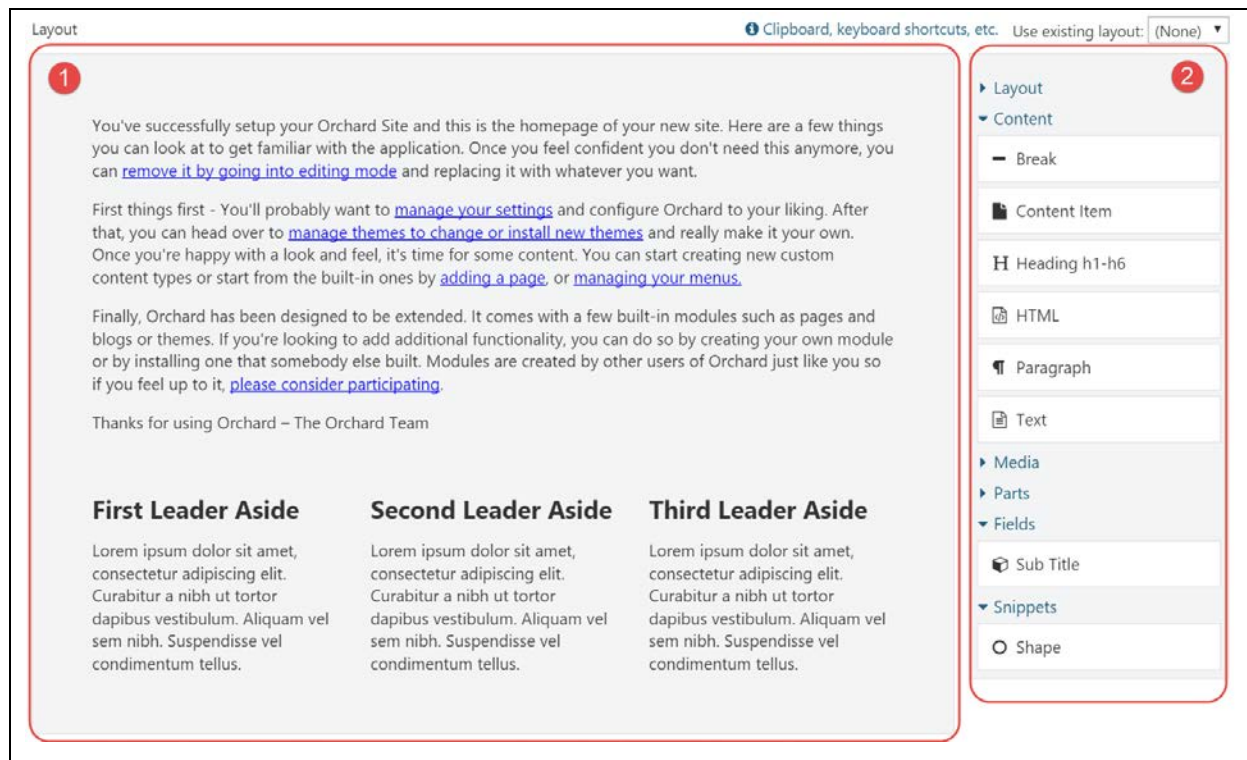


Figure 2-1 – The Layout Editor consists of the canvas (1) and the toolbox (2).

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.

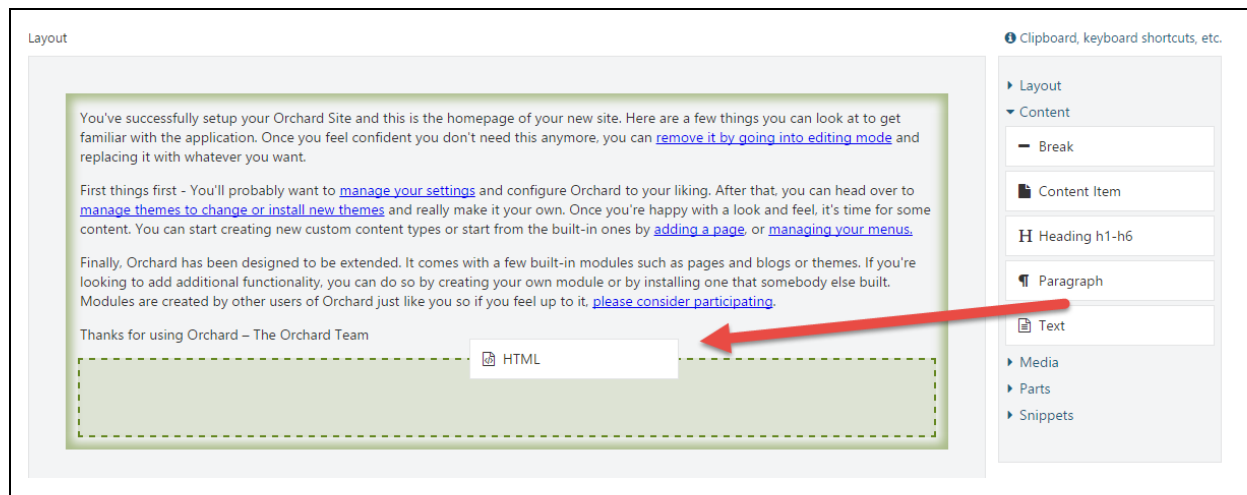


Figure 2-2 – The user drags and drops elements from the toolbox to the canvas.

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.

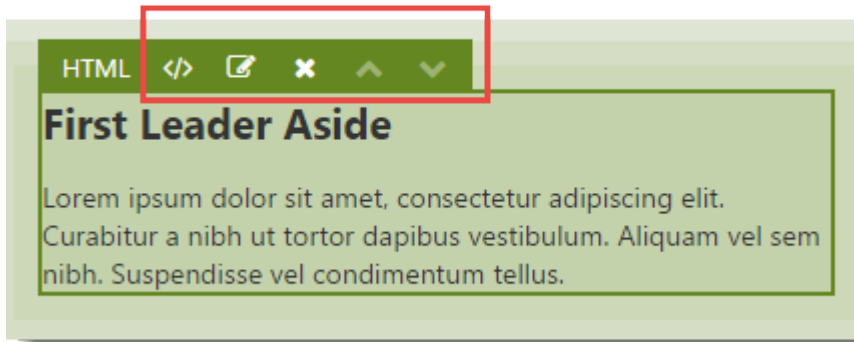













Figure 2-3 - Each element has a toolbar to control properties of and perform operations to the element.

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 dropdown 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.

Re-sizing Columns

Column elements can be re-sized 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 that, 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. Which, at its core, is about the user being able to add elements to a canvas.

Although that may sounds pretty mundane, it is actually a very powerful feature that unlocks a host of new possibilities to the user which we will explore 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 list of 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:

- Layout
 - Grid
 - Row
 - Column
 - Canvas
- Content
 - Break
 - Content Item
 - Heading
 - HTML
 - Markdown (requires the *Markdown Element* feature to be enabled)
 - Paragraph
 - Projection (requires the *Projection Element* feature to be enabled)
 - Text
- Media
 - Image
 - Media Item
 - Vector Image
- Parts
 - BodyPart
 - CommonPart
 - TagsPart
 - TitlePart
- Fields
 - Any content field attached to the content type will be made available as an element
- Snippets

- Shape
- Any Razor file ending in *Snippet.cshtml* in the current theme or any module will be made available as an element
- UI (New as of Orchard 1.10)
 - Breadcrumbs (requires the *UI Elements* feature to be enabled)
 - Menu (requires the *UI Elements* feature to be enabled)
 - Notifications (requires the *UI Elements* feature to be enabled)
- Widgets (New as of Orchard 1.10)
 - All widgets are available as elements

DynamicForms is another module introduced with Orchard 1.9 and provides its own set of elements.

Let's go over each category and their elements.

Layout

Elements in this category are typically container elements that layout their child elements in a particular way. Some container elements only support a specific set of child element types. For example, the Grid element can only contain Row elements, which in turn can only contain Column elements.

Grid

The Grid element is a container element that can hold only Row elements.

Use the Grid element whenever you want to create a layout of elements. As mentioned, Grids contain Rows, and Rows in turn contain Columns. As you can imagine, these three types of elements are fundamental to create layouts.

When you add a new Grid to the canvas, it will not contain any Row elements initially. You will need to add Row elements to the Grid yourself. Easy enough.

Row

The Row element, like the Grid element, is a container element. However, this element can only contain Column elements.

The Row element is represented in the toolbox as 7 pre-configured Row elements. The first Row toolbox item will add a single Row element with 1 Column element, the second toolbox item will add a single row with 2 columns, and so forth.



Figure 3-1 - The Row element is represented in the toolbox as a set of pre-configured rows.

The Row element has a specific toolbar command called *Distribute columns evenly*. As its name implies, this command will evenly distribute the width of the columns of the row based on a *maximum size of 12* columns. For example, if a Row element has 4 columns with varying sizes, and the maximum size is 12, each column will be re-sized to be exactly 3 units in size.

Column

Unlike the Grid and Row elements, the Column element can hold any type of child elements, except for Row and Column elements. Columns themselves can only be contained by Row elements.

A column has two specialized properties: *Width* and *Offset*. Together, these values make up the *total size* of the column.

Adding Columns

There are various ways to add columns to a row. If the row doesn't contain any column, you need to add a Column element to it. In addition, if the Row element contains at least one Column, you can split that column into two. You can repeat this until you have the desired amount of columns.

Offsetting

Another property of the Column element is its *Offset*. By default, the offset for any column is 0. Increasing the offset increases the overall size of the column. You typically use an offset if you want the contents of a column to appear more to the right.

Combined with Row and Grid, the Column element is one of the key ingredients for creating layouts.

Canvas

The Canvas element, like the Column element, can contain any type of element, except for Row and Column elements. Unlike the Column element, the Canvas element can be added to any other container element, except for Grid and Row, since they have an exclusive white-list of allowed children.

Whenever you create a new layout, that layout always start with a root element of type Canvas.

The Canvas element is great if you ever need a generic type of container element. Although in most cases you probably won't need it since the Column element already is a container, there are occasions where a Canvas is very useful. The prime example being when working with *Layout Templates*. A layout template enables the reuse of pre-created layouts. When a layout template is applied to a layout, the elements from the template are *sealed*, which means the user cannot make any modifications to these elements. If a sealed container element is empty however, it will accept elements as its children. But if a sealed container element contains at least one element, no elements are allowed to be added to that container. To work around that, you need to add an empty container element that acts as a placeholder. The Canvas is quite suitable for that purpose.

See chapter 4 for more information on Layout Templates.

Content

As you have probably guessed, the Content category contains all elements concerning content.

Break

The Break element is probably one of the simplest elements available. It has no specialized properties. All it does is render the `<hr>` HTML element.

Content Item

The Content Item element is very similar to the *Content Picker Field* and enables the user to select one or more content items to render inline on the canvas.

When adding or editing a Content Item element, the user is presented with a dialog window displaying the content properties of the element. The element has two specialized properties:

- A list of content items to render
- The *DisplayType* to use when rendering the selected content items

This element enables you for example to render the same content item at various locations.

Heading

The Heading element maps directly to the `<h1>` to `<h6>` HTML elements, and has two specialized properties:

- Level
- Text

The Level indicates the size of the heading and ranges from 1 to 6. For example, if you specify level 3, the `<h3>` tag will be rendered. The following is an example of HTML output when specifying level 6 and the text *"Hello Layouts!"*:

```
<h6>Hello Layouts!</h6>
```

Html

The Html element is probably the most commonly used one when it comes to placing content onto the canvas. It has a single property called *HTML*, which stores the HTML markup. Use this element whenever you want to display textual content anywhere on the canvas.

The HTML editor used by default is *TinyMCE*, but you can change this by enabling other features that provide another editor for the *html* flavor. For example, if you enable the *CKEditor* feature, that's the editor you'll see when editing Html elements.

Markdown

The Markdown element lets the user use Markdown syntax which gets transformed into HTML when being rendered on the front-end.

You'll need to enable the *Markdown Element* feature to enable this element.

Paragraph

The Paragraph element maps directly to the `<p>` HTML element and enables the user to add individual paragraphs to the page.

You may be wondering why you would want to use this element over the Html element. Well, here's the idea: all elements have common properties such as HTML ID, HTML Class and Html Style. These property values are rendered as HTML attributes on the HTML tag output of the element. Now, when the Html element is rendered and has a value for at least one of the three common properties, a surrounding `<div>` element is rendered onto which the common properties are rendered as attributes. However, there may be occasions where you actually want these common property values to be rendered as attributes on a `<p>` tag directly instead of a surrounding `<div>` element. That's when you use the Paragraph element.

Projection

The Projection element is the little brother of the Projection Part and allows the user to select a Query to project a list of content items.

In order to make this element available from the toolbox, enable the *Projection Element* feature.

Text

The Text element provides a simple *textarea* input control for its content input and renders that input as raw HTML.

I'm not sure when you would ever want to use this element, but it is there if you prefer to hand-code the HTML directly, instead of using a WYSIWYG editor such as TinyMCE.

Media

The Media category contains all elements that display some form of media such as images, documents and videos.

Image

The Image element allows the user to pick a single image content item from the Media Library. When rendered on the front-end, the element renders the `` HTML element. Use the Image element when:

- You only need to display a single image per element.
- You want the common properties to be rendered as part of the `` HTML tag rather than the `<div>` element that is rendered when using the Media Item element.

Media Item

The Media Item element allows the user to pick more than one media item. The user can control what display type to use when rendering the selected media items. Use the Media Item element when:

- You want to display a list of various types of media items.
- You want to control the display type being used to render each media item.

Vector Image

The Vector Image element is similar to the Image element, but only supports vector graphics formats such as .svg. In addition to the selected media item, the Vector Image element has two additional properties: *Width* and *Height*, both expressed in number of pixels. These values will be rendered as width and height attributes on the HTML tag.

Parts

The Parts category contains elements for content parts that:

- Have their *Placeable* property set to *true*.
- Are attached to the current content item's type.

Part elements are interesting. They basically enable the user to place the content parts attached to the content type anywhere within the layout of the content item. You will need to configure Placement.info however to prevent the content parts from being displayed at their default locations. You can read ore on that in chapter 9.

Placeable Parts

Content parts aren't placeable by default. The Placeable property is a new part property that controls whether the content part is harvested as an element. By default, the Placeable property is set to true for the following content parts:

- BodyPart
- CommonPart
- TagsPart

- TitlePart

Fields

The Fields category is similar to the way the Part elements work, but with a few differences:

- Only content fields attached to the current content item's type are displayed as elements
- There is no Placeable property for content fields, which means that all content fields are placeable when attached to a content type.

Snippets

The Snippets category holds two types of elements by default: One is a generic Shape element, and the others are based on a convention where Razor files are named in a certain way.

Shape

The Shape element is a very simple element that has just one content property called *Shape Type*. If you provide the shape type of an existing shape here, then that shape will be rendered wherever you place the Shape element.

You could even enable the *Templates* feature, create a template, and use that as the shape type for the Shape element.

Snippet Elements

The second type of elements in the Snippets category are called *snippets*.

Snippets are quite similar to the Shape element, but the key difference is that instead of you providing the shape type name, the *Snippet element harvester* provides elements based on the existence of Razor files in the current theme whose file names end in *Snippet.cshtml*.

Snippets are provided by the *Layout Snippets* feature, so be sure to enable it when you want to try it out.

For example, a Razor file called *LogoSnippet.cshtml* in the Views folder of the current theme (or any module for that matter) would yield an element called *Logo*.

Snippets are a great tool for theme and module developers, as they provide a quick way of providing elements without having to write element classes and drivers.

What's more, Snippets can be made configurable. We will how this works in detail in chapter 11.

UI

This feature is new as part of Orchard 1.10

The idea behind the UI elements is that they provide the user with elements that make up the UI of their site. Think menus, breadcrumbs and notifications.

The UI elements are provided by the **UI Elements** feature.

However, I'm afraid that these elements will not be very useful before Orchard has support for adding elements to zones and layers. Let me try and explain why.

Imagine you wanted to display the main menu using the Menu element. There are two major limitations with this:

1. There is no way to add elements to global zones. Although you could work around this by simply designing your theme in such a way that your entire website consists of a single Content zone and leveraging Layout Templates, this won't solve the next problem, which is:

2. The layout part is associated with content items. This means that their layouts only appear when you request the content item. This means that if you navigated to the Login screen for example, you would no longer see the main menu if you implemented that as a Menu element.

So, unless you want to display a UI element on specific pages, my advice is to not use them until Orchard unifies its Widgets and Elements story.

Widgets

This category provides all widgets as elements, enabling the user to add widgets to layouts.

When you place a widget element onto a layout, the widget element harvester will create an actual widget content item for you and render that one on the front end. The created widget is not linked with any zone or layer, since it is linked with the content item containing the layout. If you delete the content item or the widget element, the widget itself is deleted as well, since it's managed by the element.

The ability to add widgets to layouts is new since Orchard 1.10.

Summary

In this chapter, we got to meet all of the elements that ship with Orchard out of the box.

We went over each element in detail to get a better understanding of each element's intended use.

The set of elements can be extended by custom *element harvesters*, which are responsible for providing elements to the system. Module developers can create custom harvesters, into which we will look in great detail in *Part 3 - Extensibility*.

4. Layout Templates

Layout *templates* are a way to reuse layouts that you created earlier. For example, if you have a lot of content items that use a two-column layout, instead of re-creating that layout from scratch each time, you can create a layout once and reuse it on your pages.

Layout templates are implemented as content items that have the Layout Part attached. The Layout Part has a setting called *Use as Template*, which is used by the Layout Editor to populate the templates dropdown list.

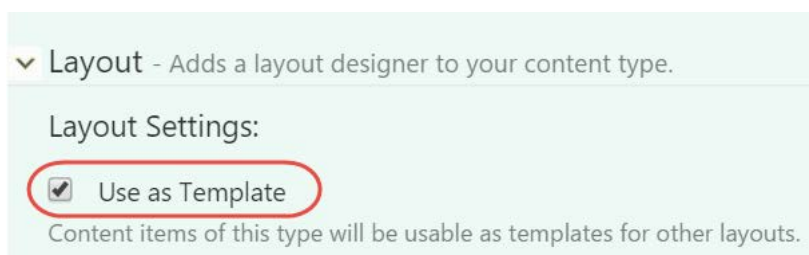


Figure 4-1 - The *Use as Template* setting on the *Layout Part* turns the content type into a layout template.

The templates dropdown list is only present when there's at least one template in the system.

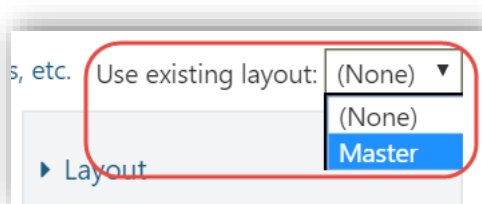


Figure 4-2 - The *Layout Editor* displays a dropdown list with available templates.

After you applied a layout, you'll notice that the elements inherited from that template are not selectable or editable, with the exception of the column elements – they are still selectable so that you can still add elements to them.

Whenever you make a change to a template, that change gets applied to all content items using that template.

Sealed Elements and Placeholder Containers

When applying a layout template, the elements inherited from that template are marked as sealed. Sealed elements cannot be modified by the user. If the sealed element is an empty container, the user can still add elements to it. However, if this container contains at least one element added via the template, then the user cannot add additional elements.

If you want to allow the user to add elements to a templated container element that already contains other elements, you will have to add an empty container to it, such as the Canvas element. The Canvas element would act as a placeholder container into which the user can add elements.

Summary

Layout templates are templates for content items that use the Layout Part. Elements inherited from a template are sealed, which means they cannot be modified from child layouts. In order to be able to add elements to a container inherited from a template, that container must be empty, otherwise it becomes sealed as well.

In conclusion, Layout templates are a powerful way to reuse commonly used layouts across many pages. Changing a template will affect all of its child layouts.

5. Element Blueprints

In this chapter we'll take a look at *Element Blueprints*, which is a feature that enables the user to create pre-configured elements that become available from the toolbox.

When to use Element Blueprints

Whenever you find that you are duplicating elements in various layouts, those elements are a candidate to be turned into an element blueprint. For example, if you need to display contact information on various pages but at different locations, you could create a pre-configured Html element with the relevant content, name the element “Contact Details”, and reuse that element anywhere you like.

Creating Element Blueprints

Creating element blueprints is easy. To create one, go to the **Elements** admin menu item right under the **Layouts** menu item. This will take you to the index screen of all of your element blueprints, which is empty by default. On this screen, click the **Create** button to the top right of the window. The screen that appears next presents you with all of the available elements to use for your pre-configured element. When one is selected, the next screen will prompt the user for the following information:

- Element Display Name
- Element Type Name
- Element Description

- Element Category

The *Element Display Name* is the user friendly name used when displaying the element in the toolbox and on the canvas.

The *Element Type Name* is the technical name of the element, and is used when serializing and de-serializing the element. No two elements can have the same name.

The *Element Description* is an optional field that gives you the opportunity to describe what the element represents. The description helps users get a better understanding of what this element represents.

The *Element Category* is an optional field that lets you control in what category of the toolbox this element appears. If no category is specified, *Blueprints* is used as the category.

Trying it out: Creating an Element Blueprint

In the example that follows, we will create a blueprint element called *Contact Details*. The purpose of this element is for the user to be able to manage their contact details from a single place, while being able to place this element on various pages. Whenever the user changes their contact details, the changes are reflected everywhere.

Step 1

Click on the **Elements** admin menu item.

Step 2

Click the **Create** button on the top right side of the window and select the *Html* element as the base element for our element blueprint.

Step 3

On the screen that appears next, provide the following values:

Field	Value
Element Display Name	Contact Details
Element Type Name	ContactDetails
Element Description	My contact details
Element Category	Demo

Which should look like this:

Create Element based on HTML User: admin | Change password | Logout

Element Display Name
contact Details
The friendly name of the specialized element. Example: Special HTML.

Element Type Name
ContactDetails
The technical type name of the specialized element. Example: Acme.Elements.SpecialHTML.

Element Description
My contact details
Optionally enter a description for this element.

Element Category
Demo
Optionally specify a category for this element.

Create Cancel

Figure 5-1 – Creating a new Element Blueprint.

Hit **Create** to continue to the next screen, where the *Html* element editor will appear.

Step 4

Enter the following HTML code into the Html editor (using the HTML view of TinyMCE):

```
<p>John van Dyke</p>
<p>Cell: +18723456</p>
<p>Email: <a href="mailto:j.vandyke@acme.com">j.vandyke@acme.com</a></p>
<p>Skype: johnvandyke</p>
```

And hit **Save**.

Step 5

Now that the blueprint has been created, we can start using it. Create or edit a Page content item and notice the new category called *Demo* and the new *Contact Details* element.

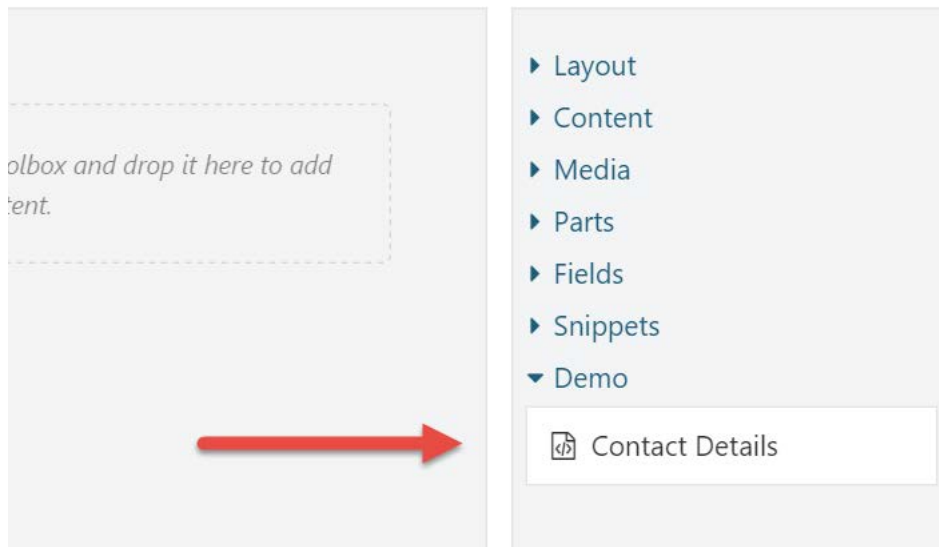


Figure 5-2 - The newly created element is available from the toolbox.

You can now add this element to as many pages as you like. When the time comes that your contact details change, all you have to do is update the blueprint element once, and the changes get reflected everywhere.

Summary

Blueprint elements enable you to create pre-configured elements and use them as normal elements. Blueprint elements are useful when you find that you use the same elements with the same configuration in multiple places. Whenever you change the blueprint element itself, the changes are reflected everywhere the element appears.

6. Elements as Widgets

So far we have seen how to work with elements and the layout editor. But elements can be used for more things than that. In this chapter, we'll see how we can use elements as widgets without the need for writing any code.

Why Elements as Widgets?

So when would you use elements as widgets? Strictly speaking, you don't have to. Let's say that you want to display some useful element in some zone. You could simply add the Layout Widget to that zone, and then add the element of course. But if you find yourself doing this often, it could make sense to simply turn the element into a widget so that you can add that widget directly without the need for the Layout Widget.

With the advent of the Layouts module, we now have the worlds of Widgets and the world of Elements. The long-term goal is to unify the two worlds. But until then, you can choose to implement custom elements and reuse them as widgets. This prevents you from having to implement both an element and a widget if you wanted to enable your users to use them as an element as well as a widget. You could of course choose to implement a widget, since widgets can be added to layouts all the same. However, a widget is a content item, which means for each widget displayed on a layout, a content item has to be loaded. Unless your page is output-cached, elements are faster to load and initialize than widgets, so that's something to keep in mind when making a decision.

So, let's talk about how to turn an element into a widget.

Using Elements as Widgets

Before you can use an element as a widget, you need to define a Widget content type that has the *Element Wrapper Part* attached.

Element Wrapper Part

The Element Wrapper Part has a single setting called *Element Type Name*. The element type name is technical name of the element that you are wrapping as a widget.

The way the Element Wrapper Part works is that it instantiates the configured element type by name and takes care of invoking the editor and display methods of that element and then returns the created shapes.

Trying it out: Creating a Widget based on an Element

In this example, we'll see how the *Contact Details* element created in chapter 5 can be reused as a widget.

Step 1

Create a new content type called *Contact Details* with the following parts:

- WidgetPart
- CommonPart
- IdentityPart
- ElementWrapperPart

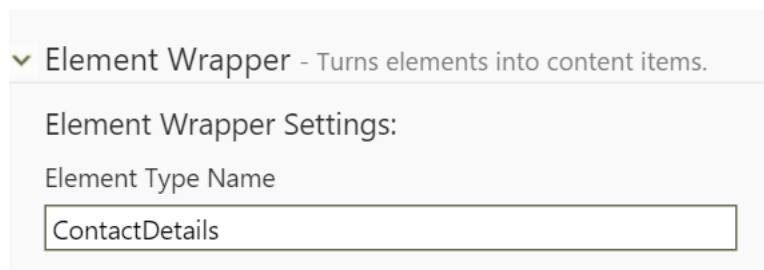
Make sure that the *stereotype* is set to *Widget* and that the type is not *creatable* or *listable*.

Step 2

Expand the Element Wrapper Part and provide the following value for the *Element Type Name* property:

ContactDetails

Hit **Save** to save your changes.



▼ Element Wrapper - Turns elements into content items.

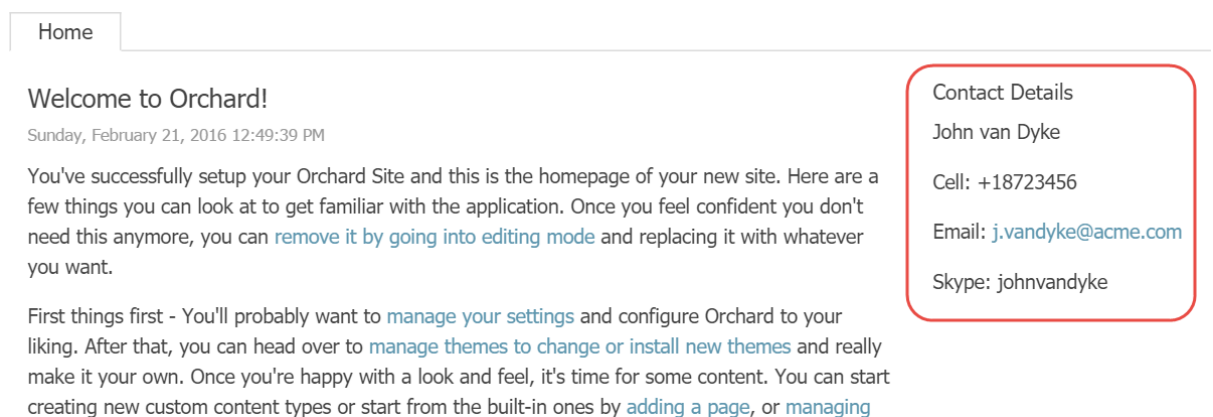
Element Wrapper Settings:

Element Type Name

ContactDetails

Figure 6-1 - The Element Wrapper Part settings,

With that in place, we can add *Contact Detail* widgets to any zone and layer.



Home

Welcome to Orchard!

Sunday, February 21, 2016 12:49:39 PM

You've successfully setup your Orchard Site and this is the homepage of your new site. Here are a few things you can look at to get familiar with the application. Once you feel confident you don't need this anymore, you can [remove it by going into editing mode](#) and replacing it with whatever you want.

First things first - You'll probably want to [manage your settings](#) and configure Orchard to your liking. After that, you can head over to [manage themes to change or install new themes](#) and really make it your own. Once you're happy with a look and feel, it's time for some content. You can start creating new custom content types or start from the built-in ones by [adding a page](#), or [managing](#)

Contact Details

John van Dyke

Cell: +18723456

Email: j.vandyke@acme.com

Skype: johnvandyke

Figure 6-2 - The Contact Details widget as it appears on the front-end in the *AsideSecond* zone.

Existing Widgets based on Elements

Being able to use elements as widget can be quite useful. For example, the Layouts module comes with a *Content Item* element. Although Orchard doesn't come with a content picker widget, it's very simple to create one with the Element Wrapper Part.

In fact, when the Layouts feature is enabled, the following additional widget types based on elements are added to the system:

- Text Widget
- Media Widget
- Content Widget

In all fairness, the Text Widget (as is the case with the Text element), is arguably not all that useful, since we already have the Html Widget. The Media Widget lets the user select one or more Media Items to be displayed, and the Content Widget lets the user select one or more Content Items to be displayed using a configurable display type.

Summary

Elements can be used in many different ways other than being added to a canvas. One such way is turning them into widgets using the *Element Wrapper Part*.

7. Element Tokens

If you've worked with Orchard for a while, you are probably already familiar with tokens. They basically provide a way to insert variables into content. These variables are processed at runtime, and it is up to the token providers to provide the result. A few examples of tokens are:

- **Content.Author** – Renders the author name of the content item.
- **Content.DisplayUrl** – Renders the display URL of the content item.
- **Content.DisplayUrl.Absolute** – Renders the fully qualified URL of the content item.
- **Request.QueryString** – Renders the specified querystring value, e.g. {Request.QueryString:MyQueryKey}.
- **Site.SiteName** – Renders the site name as configured in the **Settings** section.

There are various places where you can use tokens. For example, Orchard.Autoroute uses tokens for its configurable route patterns, and many Orchard.Workflows activities support tokens as configuration values.

The Element.Display Token

The Layouts module also provides a token, which is the **Element.Display** token.

This token is bound to the *Element Tokens* feature, so you need to enable that before you can use this token.

The purpose of the `Element.Display` token is simply to render the element that is provided as its argument. For example, the following will render the *ContactDetails* element created in chapter 5:

```
#{Element.Display:ContactDetails}
```

This token is especially useful when you want to render elements in HTML contents. For example, let's say you have a content type with a Body Part. Users can input some HTML, and insert elements using the `Element.Display` token, causing the specified element type to be rendered right there inline.

Although the `Element.Display` token does not support additional arguments to provide values for the element's properties, you can create element blueprints (pre-configured elements) and for example render a Projection element. If an element does not require configuration, then you can use it directly without creating a blueprint.

Trying it out: Using `Element.Display`

In this example, we'll have a look at how the `Element.Display` token works by going through the following steps.

Step 1

Make sure the Element Tokens feature is enabled.

Step 2

Create an element blueprint called *ContactDetails* as shown in chapter 5.

Step 3

Edit the *Welcome to Orchard* content item and edit the first Html element containing the introductory text, and insert the **Element.Display:ContactDetails** token anywhere between two sentences or paragraphs.

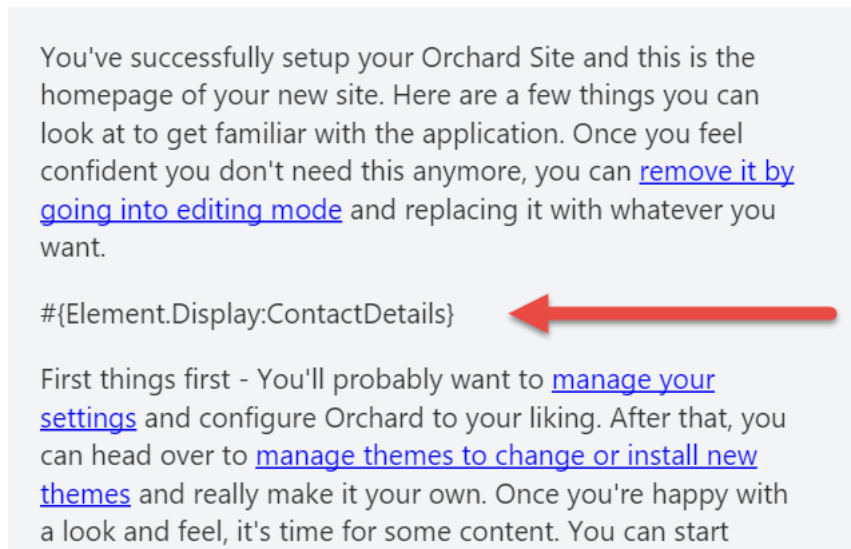


Figure 7-1 - Enter the *Element.Display* token somewhere in the Html element's contents.

And hit **Publish Now**.

Notice that I'm using the new token syntax where tokens start with the hash-tag symbol (#). Omitting this will cause your tokens not to be executed.

Now go to the homepage of your site and notice how the token has been replaced with the actual ContactDetails.

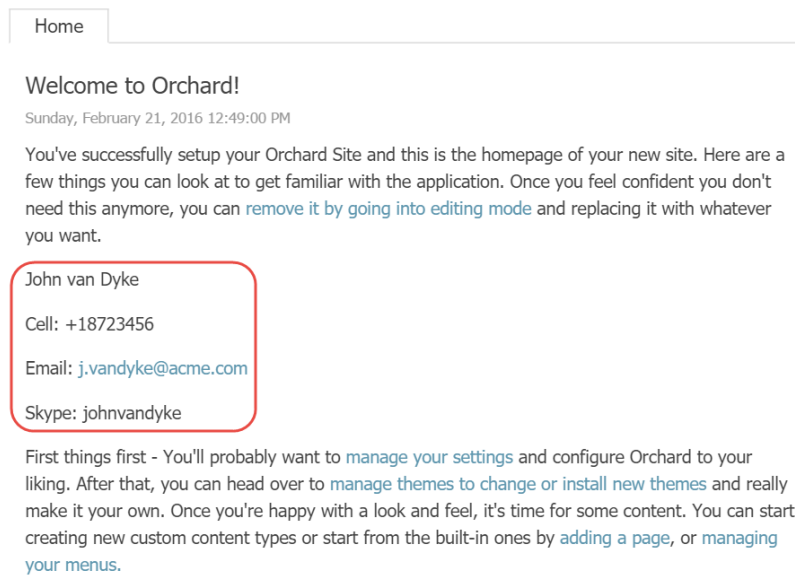


Figure 7-2 - The `Element.Display` token is dynamically replaced with its actual display.

This works pretty much anywhere you can use tokens, including the Body Part and the Email field of the Email Workflows Activity for example.

Summary

In this chapter, we've seen yet another way to display elements, this time using the `Element.Display` token.

Although this token does not support the specified element to be configured, we can create element blueprints and render them instead. The intended use of the token is to allow elements to be rendered anywhere, such as inside bodies of text.

8. Element Rules

Element Rules are very similar to Widget Layer Rules, or Layer Rules for short, but instead of controlling widget layer visibility, they control *element visibility*. Element Rules, like Layer Rules, use the rule engine provided by Orchard.Conditions.

Orchard.Conditions is a new module as of 1.10, and replaces the now deprecated rules engine provided by the ****Orchard.Widgets**** module.

This enables you for to only display elements if a certain condition evaluates to true.

If a rule applied to a container element evaluates to false, then that element, including its children, will not be rendered.

One scenario where you could use this is for example if you have elements that you want to be visible only to authenticated users.

Available Functions

To use rules, you use functions that evaluate to a **boolean** value. The following is a list of functions that are available out of the box:

Function	Description

authenticated	Evaluates to true if the current user is authenticated, false otherwise
contenttype	Evaluates to true if the current content item being displayed is of the specified content type, false otherwise. Example: <code>contenttype("Page")</code>
url	Evaluates to true if the current content item being displayed is of the specified content type, false otherwise. Example: <code>url('~/contact')</code>

Functions can be combined using the logical **and** and **or** operators and negated with the **not** operator. For example, the following rule will evaluate to true if the requested URL is either `~/contact` or `~/about` and the user is not authenticated:

```
(url('~/contact') or url('~/about')) and not authenticated
```

Applying Element Rules

To apply an element rule, click on an element's Edit toolbar icon and enter the desired rule into the Visibility Rule text area. Make sure to publish the content item to save the changes. The next time you visit the content item on the front end, the element will only be rendered if the specified rule evaluates to true.

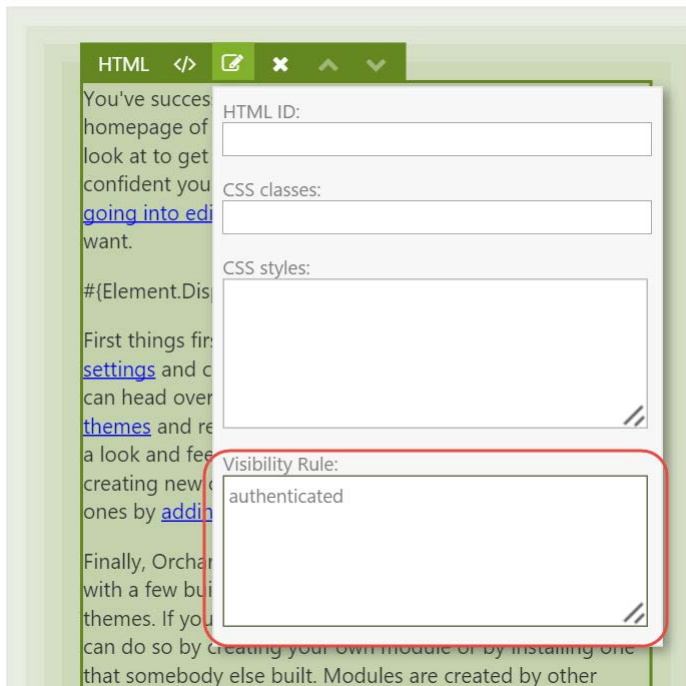


Figure 8-1 - The Element Rule editor.

Unfortunately there is currently no way to write a library of rules that you can reuse like we can today with Layer Rules, but rumor has it that a future version will have it.

Summary

That's really all there is to element rules. The rules engine used is the same one used by the Widgets module, and provides control over the visibility of a given elements.

9. Theming

Orchard has a powerful theming story that allows theme developers endless freedom to customize pretty much everything. In this chapter, we'll go over what shape templates to override when customizing the rendering of elements. To understand how theming works, it is helpful, if not important, to understand an Orchard concept called Shapes. Now, you don't need to know everything there is to know about shapes, but here's a primer.

A Primer on Shapes

Shapes are at the heart of the Orchard rendering engine, and are dynamic objects that can be rendered into HTML. Shapes act as the view model for a given shape template, which is typically (but not necessarily) a Razor view. In fact, shape templates can be implemented as:

- Razor views
- Shape methods
- Templates (a content type provided by Orchard.Templates)

Interesting fact: this list can be extended with custom implementations of **IShapeTableProvider**.

Some key characteristics of shapes are that they:

- **Can be rendered.** This means that we can render a given shape into an HTML string. This is made possible because a shape

carries information about what shape templates to use for rendering.

- **Can contain child shapes.** This means that we can build and render trees of shapes.
- **Are dynamic.** Which means that we can add properties and methods to a shape at runtime.

The following section dives a little deeper into the anatomy of shapes, however this is not something you need to understand in order to be effective with overriding element shape templates. The reason I included it is because I think one can never read enough about shapes, as they are the single most complex concept in Orchard. Once you fully grasp how they work, the world is at your feet. Or at the very least, you'll be able to create themes and modules with confidence.

Anatomy of a Shape

All shape objects are instances of the `Shape` class, which itself implements the **IShape** interface and inherits from **Composite**, which in turn inherits from the .NET **DynamicObject** class.

The `Composite` class implements the dynamic behavior when typed as **dynamic**. This is not unlike the way the .NET **ExpandoObject** works, where you can dynamically add members to a dynamic instance at runtime, using an internal dictionary to store members and their values.

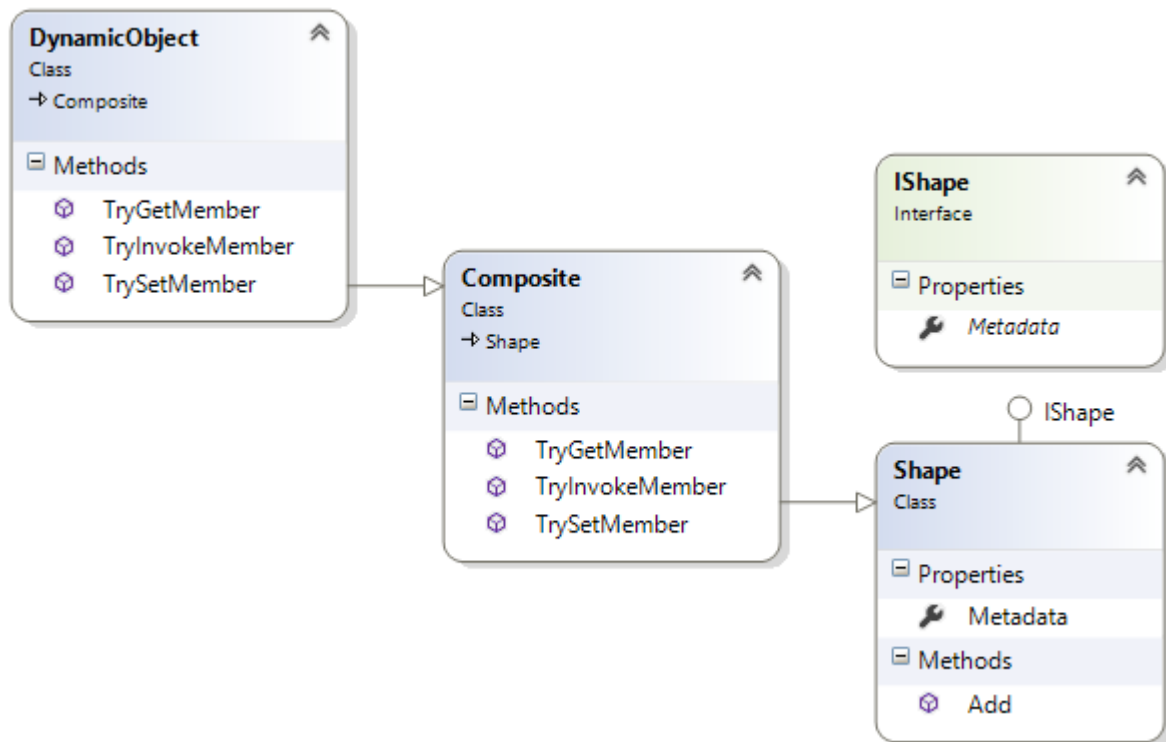


Figure 9-1 - The class hierarchy of the Shape class.

As you can see in above diagram, every shape has a Metadata property which contains information such as the name of the shape. It is this metadata that is key to rendering shapes, as we'll see next.

Shape Templates

To render a shape, Orchard relies on a service called the *display manager*. The display manager needs to know what shape template to use. In order to determine that, it uses three pieces of information to find the template to use:

1. The shape's *Name* (stored in Metadata).
2. The shape's *Alternates* (also stored in Metadata).
3. A shape table to get a *shape binding* based on either the shape name or one of the shape alternates.

The shape binding contains a delegate that will perform the actual rendering (which could be using the Razor view engine, a shape method, or potentially something else entirely).

If the shape table matches with any of the alternates, the first match is used. If no alternates match, a binding based on the shape Name is returned. If no binding was found, an exception is thrown.

If you try to render a shape for which no binding exists, an **OrchardException** with the message "Shape type {0} not found" will be thrown.

Once a binding is found, the display manager invokes its **Binding** property, which is of type **Func<DisplayContext, IHtmlString>**.

The shape rendering process essentially boils down to:

1. Given a shape object, get a function from the shape table provider to render that shape.
2. Invoke that function and render the returned HTML string.

If the previous section feels like it is above of your head, don't worry. In practice, all you need to do to render a shape is invoke **@Display** from a Razor view, passing in an instance of a shape. The **Display** property of a Razor view ultimately invokes **IDisplayManager.Execute**, which does the shape table look-up as I just described.

Ok, now let's see what shapes are created by the Layouts module and get a better understanding of what shapes and alternates we can use.

Elements and Shapes

As is the case with content items, parts and fields, Elements too are rendered using shapes. More specifically, for any given Element to render, a shape with the name **Element** is created and rendered.

To differentiate one Element shape from another, a set of alternates are included as part of the shape. One such alternate is based on the technical name of the element. That way, each element has its own shape template.

Similar to the **IContentDisplay** service that turns a **ContentItem** instance into a Content shape, the **IElementDisplay** services turns an **Element** instance into an **Element shape**. If an element has child elements, the service recursively creates shapes for them as well, adding those shapes to their parent shape.

Out of the box, the following alternates are added when creating an Element shape:

- Elements__<typeName>
- Elements__<typeName>__<displayType>
- Elements__<typeName>___<category>
- Elements__<typeName>__<displayType>___<category>

The *typeName* used in the alternates is the .NET type name (without the namespace) of the element class. For example, when an Element shape is created for the **Html** element, the *typeName* will be “Html”.

The *category* is the name of the category to which the element descriptor is assigned, “Content” for example.

The *displayType* is the value provided for the **displayType** argument when displaying a shape using the Element Display service.

If Orchard finds a shape template matching one of the alternates (where the last added alternate is considered the most specific one), it uses that template.

Orchard maps alternate name syntax to filename syntax by replacing underscores with dots and double underscores with hyphens. Check out the documentation for how this works, or my blog post at <http://www.ideliverable.com/blog/a-closer-look-at-content-types-drivers-shapes-and-placement>

For example, given a *displayType* of “Design” and a *category* of “Content”, the Element shape alternates map to the following Razor view filenames:

- Elements.Html.cshtml or Elements/Html.cshtml
- Elements.Html.Design.cshtml or Elements/Html.Design.cshtml
- Elements.Html-Content.cshtml or Elements/Html-Content.cshtml
- Elements.Html-Content.Design.cshtml or Elements/Html-Content.Design.cshtml

Notice that you can use a slash instead of a dot after “Element”. This conveniently enables you to organize your Element shape templates in an “Elements” folder.

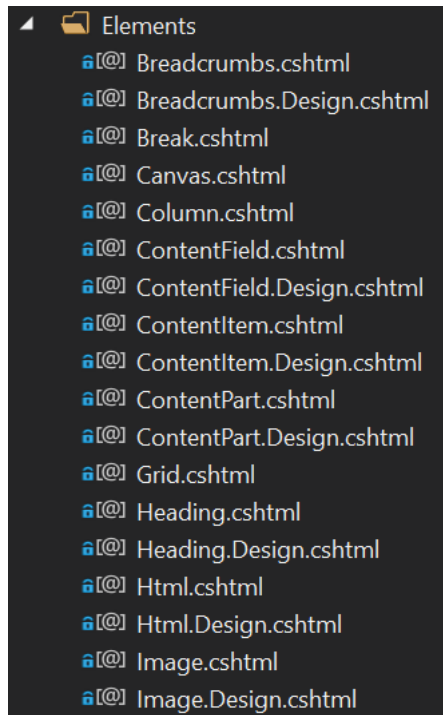


Figure 9-2 - Element shape templates can be organized in Views/Elements.

Overriding Element Shape Templates

Out of the box, Orchard takes care of rendering the Element shapes for you. Most of the times, you don't need to override these templates. Except, for example, if you're creating a theme based on a CSS Grid framework such as Bootstrap. Then you will want to override the `Grid.cshtml` and `Column.cshtml` shape templates so that you can provide the appropriate CSS classes that Bootstrap expects. Another example may be where you have a specific requirement for rendering additional HTML around a particular element.

To override a shape template, all you need to do is copy one of the existing templates from the Layouts module into the Views folder of your theme and apply your customizations. Easy, right? It is. The tricky part is discovering what shape templates to override and what shape alternates you have at your disposal. Although the Shape Tracing feature is a tremendous help when getting started, I found it even easier to just attach a debugger and set breakpoints in various Razor views and inspect the `Model` property (which is the shape object itself) and its list of Alternates.

Custom Alternates

As mentioned, the `Element` shape contains a default set of alternates. But, as is the case with any shape, you can programmatically add custom alternates yourself, giving you fine-grained control over what shape template to use based on whatever conditions you like.

For example, let's say you wanted to provide an alternate based on the `HTML ID` property of an element so that you can use a specific shape template for certain elements with certain IDs. To do so, you would implement the **`IShapeTableProvider`** interface and add an **`OnDisplaying`** event handling function from where you add the alternates.

Let's see how that works.

Trying it out: Creating Custom Alternates for Element Shapes

The following code snippet shows a shape table provider class that adds an alternate based on the element's `HTML ID` property. I excluded the

namespaces for brevity, but the complete code is provided as the complimentary sample code accompanying this book.

```
public class CustomElementShapes : IShapeTableProvider {
    public void Discover(ShapeTableBuilder builder) {
        builder.Describe("Element").OnDisplaying(context => {
            var element = (Element)context.Shape.Element;

            // Get the HtmlId value of the element.
            var htmlId = element.HtmlId;

            // Don't add custom alternates if no HTML ID value was provided.
            if (String.IsNullOrEmpty(htmlId))
                return;

            var typeName = element.GetType().Name;
            htmlId = htmlId.ToSafeName();

            // Example: Elements/Html-MyJumbotron.cshtml
            var alternate = $"Elements_{typeName}__{htmlId}";
            context.Shape.Metadata.Alternates.Add(alternate);

        });
    }
}
```

The above shape table provider describes the **Element** shape type and adds a handler for the **OnDisplaying** event of shapes of that type. When that handler executes, it gets a reference to the actual **Element** instance. This is necessary so that we can get its configured **HtmlId** property. If no such value was specified, we do nothing and return. Otherwise, we construct an alternate using the element type name and its **HtmlId** value.

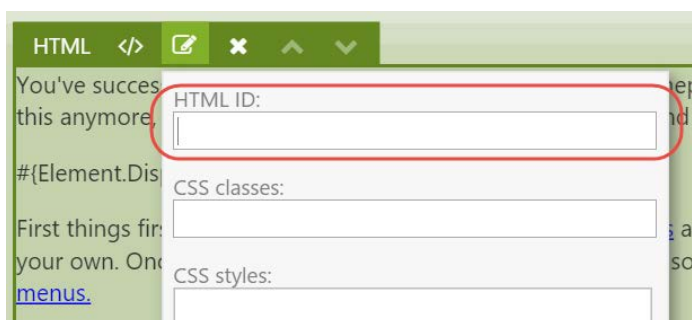


Figure 9-3 - Every element has an HTML ID property.

Given this new alternate, we can now provide specific shape templates by assigning an HTML ID to an element and providing a shape template in or theme based on our custom alternate for that HTML ID value.

For example, if you provide an HTML ID value of “MyJumbotron” to an Html element, you can create a Razor view with the following filename: **Elements/Html-MyJumbotron.cshtml**.

Content Part and Field Elements

In chapter 3 we were introduced to various element types and their categories, two of which being *Parts* and *Fields* elements. These elements allow the user to place content parts and fields attached to the current content type anywhere on the canvas.

When you first start using Part and Field elements, you'll notice that these parts and fields are rendered at least twice on the front-end, which is not typically what you want.

The reason this happens is because that out of the box, part and field shapes are configured to be placed in the Content zone using *Placement.info* files. A *Placement.info* file is an XML file that contains placement configuration for shapes created by content part and field drivers. Now, in addition to those shapes being placed, any part and field elements placed onto the canvas will also render these parts and fields.

To fix this, we need to tell Orchard to not render the shapes as configured by the default *Placement.info* files, since we are placing those things ourselves via the layout editor. The way to do this is by updating the

Placement.info file of the current theme, which is easy to do once you know how.

Updating Placement.info

To prevent a part or field shape from being rendered, all you need to do is specify an empty string (or a hyphen) as the value of the attribute representing the shape type. For example, let's say you wanted to place the *Title Part* shape onto the layout. Since the Title Part Driver returns a shape called **Parts__Title**, we can add the following XML to Placement.info in the current theme:

```
<Placement>
  <Place Parts__Title="-" />
</Placement>
```

With that change, however, neither the default shape nor the shape created by the element will be displayed. This is because the part and field element drivers use the *Content Display* service to execute the part and field drivers and process the returned shapes by applying Placement.info. So, if we configured those shapes to not be rendered, no shape will be created.

What we need to do instead is add a **<Match DisplayType="Layout">** element. The secret is that Part and Field element drivers use the *Layout* display display type. Within that Match element we then configure the **Parts__Title** shape to be placed in the **Content** zone of the element shape. The final Placement.info configuration would look like this:

```
<Placement>
  <!-- Don't render the Parts_Title shape by default. -->
  <Place Parts_Title="-" />
  <!-- Except for Parts_Title shape being rendered using the "Layout" display
type that is used by Layouts. -->
  <Match DisplayType="Layout">
    <Place Parts_Title="Content" />
  </Match>
</Placement>
```

With this placement configuration in place, you can now place the Title Part anywhere on the canvas without that part being rendered twice.

It works exactly the same for Content Field elements.

Summary

In this chapter, we learned about shapes, which are dynamic objects that serve as view models for shape templates. Shape templates take care of generating HTML, and can be implemented either using Shape methods or Razor views.

We also learned about shape alternates, which are key to understanding how to customize shape templates in your custom theme.

We then looked at a specific category of elements: Part and Field elements. They provide the user with control over where to place them on a canvas. However, the theme's Placement.info file needs to be updated to prevent the default shapes from being rendered into their default zones.

10. Bootstrap

Many websites these days are built with CSS grid frameworks such as Bootstrap. Such frameworks make it easy to create grid-based layouts using `<div>` tags and certain CSS classes to define rows and columns. The Layouts module's Row and Column elements render markup containing CSS that is targeted by the TheThemeMachine theme, but when your own theme is based on Bootstrap, you will definitely want to take advantage of the Bootstrap CSS classes instead.

So, in this chapter we'll learn what shape templates to override and how to customize these templates to make them work with Bootstrap. Although I'm using Bootstrap as an example, the same principles apply to any other CSS grid frameworks such as Foundation for example. The main differences will be the CSS class names used by each framework.

Overriding Element Shape Templates

In order to understand what templates to override, let's analyze the HTML output as generated by the Grid, Row and Column elements, since we use those to create grids. For example, take the following layout:

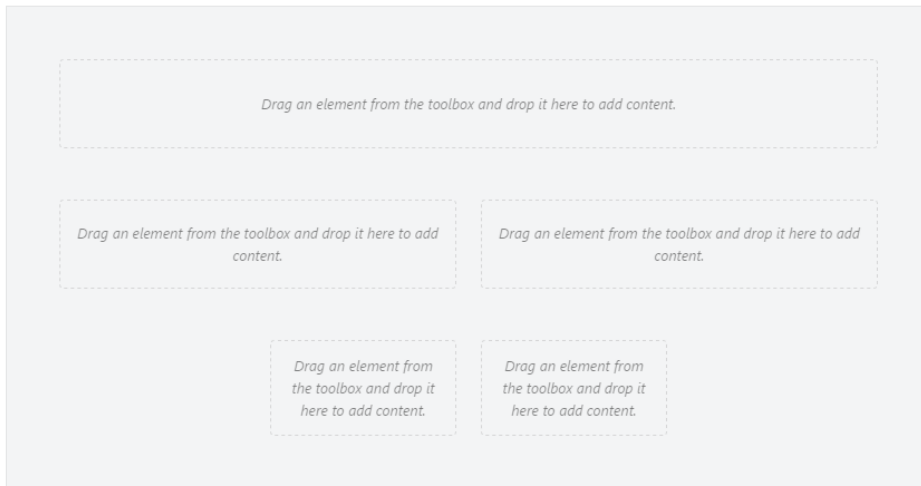


Figure 10-1 - A layout created with the Layout Editor.

When rendered on the front-end using the default shape templates, the following HTML output is generated for this grid:

```
<div class="table">
  <div class="row">
    <div class="span-12 cell"></div>
  </div>
  <div class="row">
    <div class="span-6 cell"></div>
    <div class="span-6 cell"></div></div>
  <div class="row">
    <div class="offset-3 span-3 cell"></div>
    <div class="span-3 cell"></div>
  </div>
</div>
```

Now, when we want the same grid structure implemented using Bootstrap, we would need something like the following HTML output:

```
<div class="container-fluid">
  <div class="row">
    <div class="col-xs-12"></div>
  </div>
  <div class="row">
    <div class="col-xs-6"></div>
    <div class="col-xs-6"></div></div>
  <div class="row">
    <div class="col-xs-offset-3 col-md-3"></div>
    <div class="col-xs-3"></div>
  </div>
</div>
```

Notice that the HTML structure itself can remain exactly the same. The only changes needed are the CSS classes being applied.

To change the CSS classes being used, you need to override the following two shape templates from Orchard.Layouts:

- Elements/Grid.cshtml
- Elements/Column.cshtml

Since the Row.cshtml shape template renders the **row** CSS class already, there's no need to override that one.

If you're using Orchard 1.9, you'll also need to override the Parts.Layout.cshtml shape template. This is the root shape that renders the layout and includes a default CSS grid stylesheet called default-grid.css. If you're relying on Bootstrap CSS on the front-end, you need to override that template to prevent the default grid stylesheet from being included. This has been changed in Orchard 1.10, where it's TheThemeMachine that includes default-grid.css. This way, theme developers no longer need to override Parts.Layout.cshtml just to get rid of the default grid.

To achieve the desired HTML output, copy over the listed shape templates from the *Orchard.Layouts/Views* folder to the *Views* folder of your current theme, and update each template to use the Bootstrap specific class names:

Views/Elements/Grid.cshtml

```
...
tagBuilder.AddCssClass("container"); // <-- Notice the use of the "container"
class.
...
```

Views/Elements/Column.cshtml:

```
...
var columnOffsetCss = columnOffset > 0 ? "col-xs-offset-" + columnOffset :
default(string); // <-- Notice the use of the "col-xs-offset-{n}" class.
...
tagBuilder.AddCssClass(String.Concat("col-xs-", columnSpan)); // <-- Notice the
use of the "col-xs-{n}" class.
...
```

In case you're using Orchard 1.9, also override `Parts.Layout.cshtml` and remove the inclusion of the `default-grid.css` stylesheet.

Responsive Layouts

CSS grid frameworks such as Bootstrap typically have support for *responsive design*. Responsive design in this context means handling various viewport sizes so that the layout looks good on any of them using so called *breakpoints*. A breakpoint is a certain viewport width in pixels. CSS has support for implementing breakpoints using media queries. Bootstrap implements a number of breakpoints using media queries. To target these breakpoints, it makes available the following family of column CSS class prefixes:

- col-xs- (Extra small viewports, e.g. smartphone devices)
- col-sm- (Small viewports, e.g. tablets)
- col-md- (Medium viewports, e.g. desktop computers and laptops)
- col-lg- (Large viewports, e.g. wide screens)

The idea behind these classes is that for example on larger viewports you may want to show columns horizontally, but stacked vertically on smaller viewports. Such HTML could look like this:

```
<div class="row">
  <div class="col-xs-12 col-sm-8 col-md-6"></div>
</div>
```

Now, the layout editor does not know anything about responsive design. It simply lets you create a layout. It's up to you to specify any specific CSS classes on the Column elements to provide further control on smaller and larger viewports. In the sample templates overridden earlier, we used the *col-md-* CSS class prefix, which means we are targeting medium-sized viewports by default. To have a column span 12 columns on smaller viewports, we could specify a CSS class of *col-xs-12* on a column element that spans 6 columns by default.

Trying it out: Creating Responsive Layouts with Bootstrap

In this example, we will see how to use the *CSS classes* setting on the Column elements to achieve the following responsiveness effect:

- On small viewports, a row with two columns should be equal in width (6+6),
- But for larger viewports we want the second column to be smaller than the first one (9+3).

For this demo I downloaded and installed the excellent Bootstrap theme called **PJS.Bootstrap** from the gallery.

To start, create a new Page content item with the following grid:

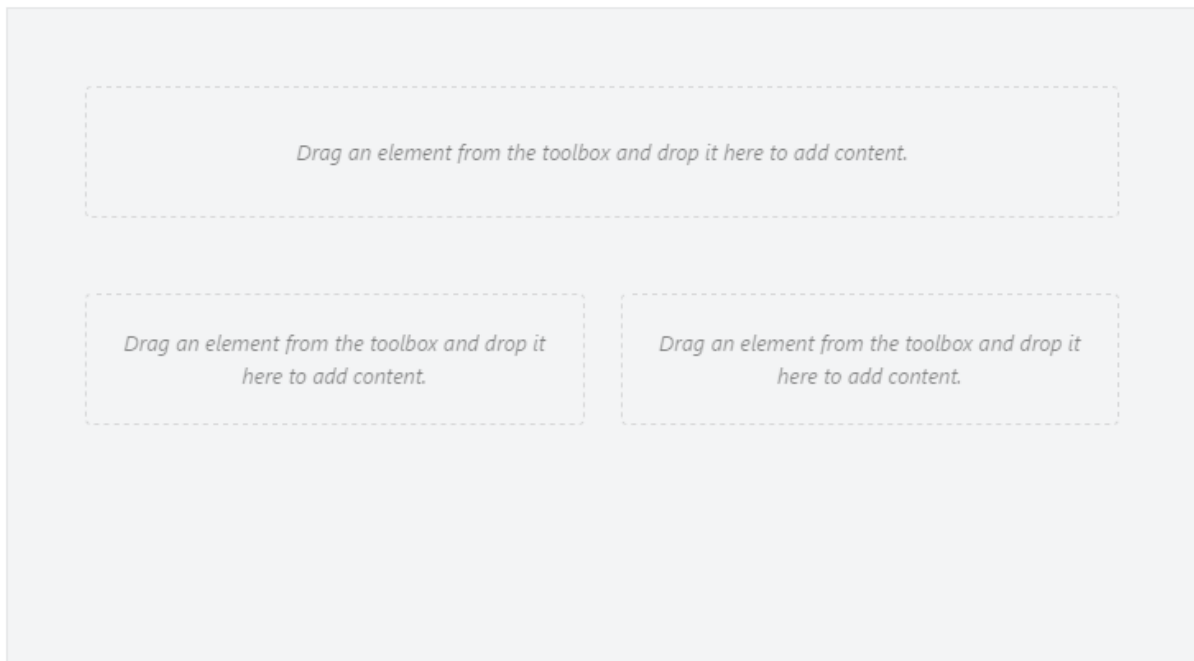


Figure 10-2

Since we used the `col-xs-` class prefix in the `Column.cshtml` template, the two columns of the second row will be sized equally, since we created the columns as 6 units each.

Next, to make the first column of the second row appear as 9 units wide and the second column as 3 units, we need to apply the following CSS classes to each column respectively:

- `col-sm-9`
- `col-sm-3`

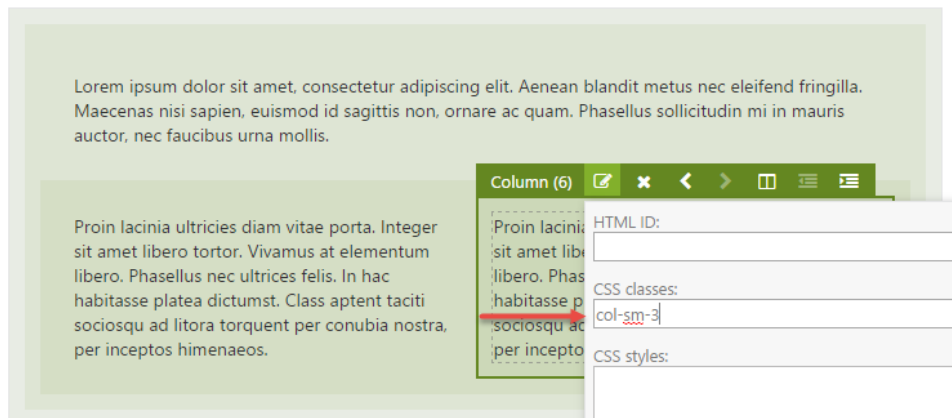


Figure 10-3 - Setting the CSS classes property to `col-sm-3`. Don't forget to also set the CSS classes of the left column to `col-sm-9`.

Now when you check out the front end and resize the browser window to simulate a very small view port, you will notice that both columns are equally sized, as designed using the Layout Editor.

When you resize the browser window to simulate a larger view-port, you'll notice that the columns are no longer equally sized. This is because the Bootstrap stylesheet defined a media query to implement a break-point for *width* > 768px, which causes the `col-sm-` prefixed classes kick in.

And there you have it: responsive design using Orchard Layouts.

Summary

In this chapter, we talked about Bootstrap and how to take advantage of its responsive features using the Layout editor to gain control over column sizes depending on various breakpoints.

To learn more about Bootstrap and responsive grids, visit <http://getbootstrap.com/css>.

11. Snippets

Snippets are another type of element that are dynamically discovered based on the mere existence of Razor views whose name follow a certain convention. When a Razor view ends in *Snippet.cshtml*, it is harvested as a Snippet element. For example, *ButtonSnippet.cshtml* would be harvested as an element called *Button Snippet*.

Parameterized snippets are a great tool for theme developers to create custom elements without the need to write element classes. All they need to do is create a Razor view and provide the necessary markup.

Parameterized Snippets

As of Orchard 1.10, the Snippets feature was enhanced to provide *parameterized snippets*. A parameterized snippet enables the user to add snippet elements and configure these snippets with custom settings.

To add settings to a snippet, all a developer needs to do is use an HTML helper called **SnippetField**. This helper does two things:

1. Provide meta information to the element editor about the configurable field, such as name, type and description.
2. Read the user-provided value when the element is being rendered.

The signature of the helper looks like this:

```
SnippetFieldDescriptorBuilder SnippetField(this HtmlHelper htmlHelper, string name, string type = null) {}
```

The following demonstrates various usages of the SnippetField HTML helper:

```
@* Render a field called "MyfieldName". *@  
@Html.SnippetField("MyFieldName")  
  
@* Render a field called "MyfieldName" with a friendly name and description. *@  
@Html  
    .SnippetField("MyFieldName")  
    .DisplayedAs(T("My Field Name"))  
    .WithDescription(T("The description of my field."))
```

The **name** argument represents the technical name of the field, while the **type** argument represents the data type of the field. At the time of this writing, the only type provided out of the box is *text*, but it is easy to add additional types as we'll see shortly.

To get a good understanding of what you can do with parameterized snippets, let's see how to create one.

Trying it out: Parameterized Snippets

In this example, we'll build a reusable, parameterized snippet called *JumbotronSnippet* with two fields: *Caption* and *Text*. The idea is that users can add any number of jumbotrons to their pages and provide a caption and a body text for every one of them.

The first thing to do is create a view called *JumbotronSnippet.cshtml* in the current theme or any custom module, and provide the following markup:

```
@using Orchard.Layouts.Helpers  
...  
<div class="jumbotron">  
    <h2>  
  
@Html.SnippetField("Caption").DisplayedAs(T("Caption")).WithDescription(T("The  
caption of this jumbotron."))
```

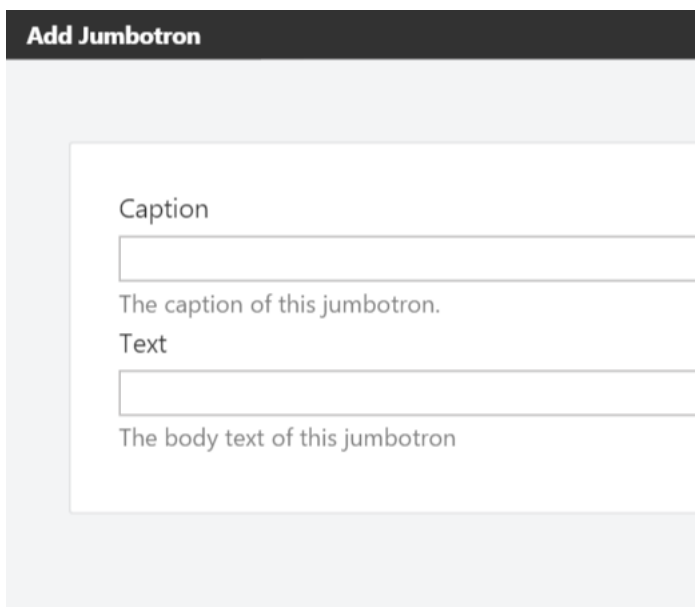
```
</h2>
```

```
@Html.Raw(Html.SnippetField("Text").DisplayedAs(T("Text")).WithDescription(T("The  
body text of this jumbotron")))  
</div>
```

Notice that I'm chaining the **DisplayedAs** and **WithDescription** methods. The fluent API enables you to do everything using a single statement, rather than having to first instantiate a snippet field, initializing it with desired values, and then render it.

With this snippet in place, all you need to do next is make sure that the *Layout Snippets* feature is enabled and add the Jumbotron element to the canvas.

Notice that as soon as you add a Jumbotron element to the page, the element editor dialog appears, showing the two fields as defined in the *JumbotronSnippet.cshtml* file.



The image shows a dialog box titled "Add Jumbotron". Inside the dialog, there are two input fields. The first field is labeled "Caption" and has a description "The caption of this jumbotron." below it. The second field is labeled "Text" and has a description "The body text of this jumbotron" below it. The dialog box is light gray and has a dark gray header bar.

Figuur 11-1 - The Jumbotron element editor displays the Snippet Fields as used in the Jumbotron.cshtml shape template.

The element editor takes the information from the snippet field descriptors as created in the view and used it to render appropriate input controls.

Custom Field Editors

Besides the default set of snippet field editors, you can provide your own snippet field editor types. All you need to do is create a Razor view in the *Views/EditorTemplates* folder of your theme or module, and give it a name as follows: *Elements.Snippet.Field.[YourEditorTypeName].cshtml*. For example, if you wanted to provide an editor called *Multiline*, you would create a Razor file called *Elements.Snippet.Field.Multiline.cshtml*. Then, when rendering snippet fields, you would specify *Multiline* as the snippet field type, as is demonstrated in the following sample snippet:

```
@Html.SnippetField("Paragraph", "Multiline")
```

The following code shows an example implementation of the field editor snippet:

```
@model Orchard.Layouts.ViewModels.SnippetFieldViewModel
@{
    var field = Model;
}
<div class="form-group">
    @Html.Label(field.Descriptor.Name, field.Descriptor.DisplayName.ToString())
    @Html.TextArea(field.Descriptor.Name, field.Value, 10, 50, new { @class =
"text large" })
    @if (field.Descriptor.Description != null) {
        @Html.Hint(field.Descriptor.Description)
    }
</div>
```

The only requirement for a custom snippet field type Razor view is to render some input field that has the **same name as the value provided by `SnippetFieldViewModel.Descriptor.Name`**, which wires up the user's input with the backing store of the snippet field.

Summary

In this chapter, we saw what snippets are and how they work. You can “parameterize” snippets so that users can configure them from the layout editor. Snippets enable theme developers to create custom elements without having to write C# classes. All that's required is a Razor view.

We also saw how to provide custom snippet field type editors, simply by creating another Razor view that follows a particular naming convention.

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	<p>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.</p>
Editor	<p>Use the Editor method from OnBuildEditor/OnUpdateEditor to create an EditorResult. An EditorResult provides a list of <i>Editor Shapes</i> 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.</p>

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 kinda cool if I could write a similar tutorial but for writing a custom element.

So that's exactly what will do next: writing 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.

The goal of the Map element is to allow the user to provide a latitude and a longitude, which the element will use to render a map image.

Let's do it.

The Map Element

First, create a directory in your module called *Elements* and add a class called **Map** as follows:

```
using Orchard.Layouts.Framework.Elements;
using Orchard.Layouts.Helpers;

namespace OffTheGrid.Demos.Layouts.Elements {
    public class Map : Element {

        public override string Category => "Demo";

        public double Latitude {
            get { return this.Retrieve(x => x.Latitude); }
            set { this.Store(x => x.Latitude, value); }
        }

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

The Map Element Driver

Next, we need to create a driver for the Map element. The driver will be responsible for displaying the element editor and updating the element with values provided by the user. Create a directory called *Drivers* and create the following driver class:

```

using OffTheGrid.Demos.Layouts.Elements;
using Orchard.Layouts.Framework.Drivers;

namespace OffTheGrid.Demos.Layouts.Drivers {
    public class MapDriver : ElementDriver<Map> {

        protected override EditorResult OnBuildEditor(Map element,
        ElementEditorContext context) {

            // If an Updater is specified, it means the element editor form is
            // being submitted
            // and we need to store the submitted data.
            context.Updater?.TryUpdateModel(element, context.Prefix, null, null);

            // Create the EditorTemplate shape.
            var editor = context.ShapeFactory.EditorTemplate(
                TemplateName: "Elements/Map",
                Model: element,
                Prefix: context.Prefix);

            return Editor(context, editor);
        }
    }
}

```

When the user hits **Save** on the element editor dialog screen, the driver's **OnUpdateEditor** method will be invoked. However, you don't need to implement that method yourself, because the **ElementDriver** base class does that for you by simply invoking the **OnBuildEditor** method, which we did implement. In that method we use the **Updater** provided by the **context** argument to bind the submitted form values against our model (the **Map** element). In this example we're using the element directly as the model, but in more advanced scenarios you may choose to work with view models instead. We'll see how that works later on.

The Map Element Editor Template

Since the driver returns an **EditorTemplate** shape that is configured to use a Razor view called "*Elements/Map.cshtml*", we'll need to create that file in the *Views/EditorTemplates* folder with the following contents:

```
@model OffTheGrid.Demos.Layouts.Elements.Map
<fieldset>
  <div class="form-group">
    @Html.LabelFor(m => m.Latitude, T("Latitude"))
    @Html.TextBoxFor(m => m.Latitude, new { @class = "text medium" })
    @Html.Hint(T("The Latitude of the location to show on the map."))
  </div>
  <div class="form-group">
    @Html.LabelFor(m => m.Longitude, T("Longitude"))
    @Html.TextBoxFor(m => m.Longitude, new { @class = "text medium" })
    @Html.Hint(T("The Longitude of the location to show on the map."))
  </div>
</fieldset>
```

Nothing too fancy going on there. All it does is render a few labels and input fields for the Map element.

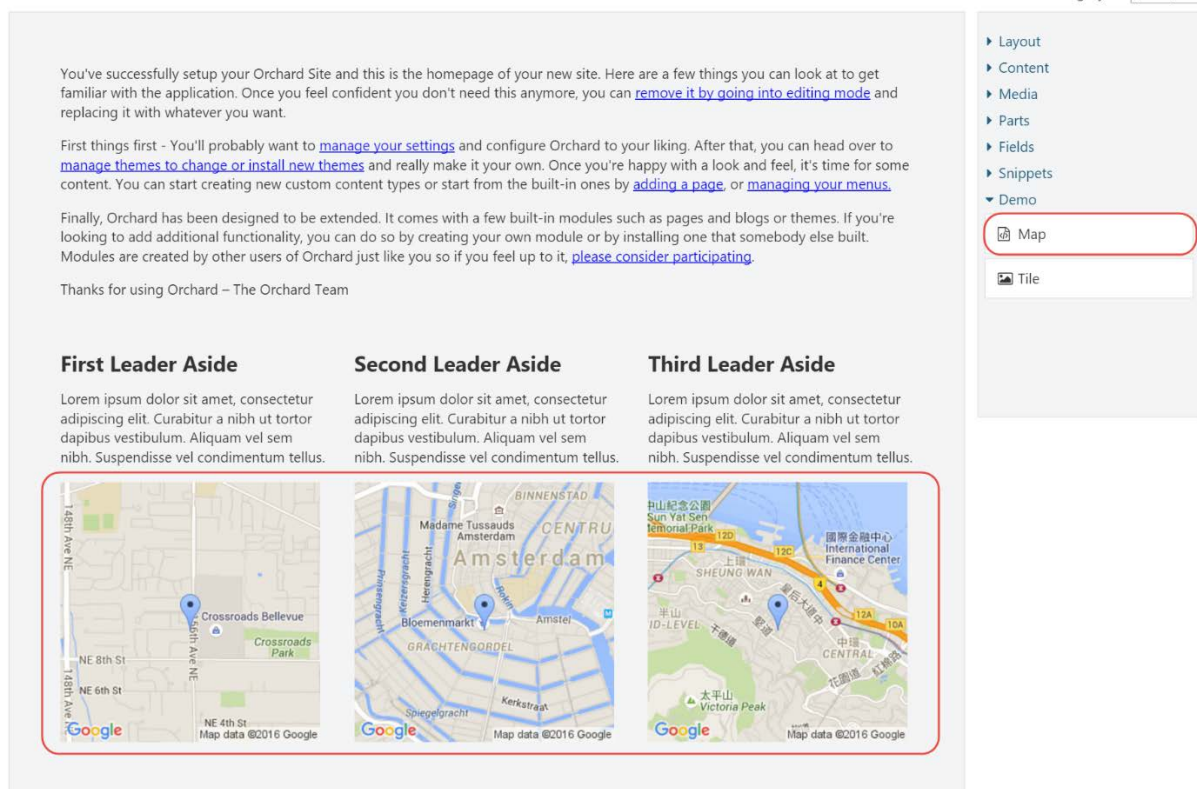
The Map Element Template

Finally, we need to actually provide a shape template for the Map element shape that will actually render the specified lat/lon location on a map. To do so, create another Razor file in the *Views/Elements* folder called “Clock.cshtml” with the following contents:

```
@using OffTheGrid.Demos.Layouts.Elements
@{
  var element = (Map) Model.Element;
}

```

With that in place, you can now add the Map element to the canvas. Contrary to the Map Part in the online documentation, you can add as many maps as you like!



Figur 12-1 - The Map element in action

Element Editors & the Forms API

In the Maps tutorial, I showed you how to implement an element editor using the **EditorTemplate** shape and a corresponding Razor template. However, there's another way to write editors that doesn't involve Razor views. The *Orchard.Forms* module provides an API to programmatically create forms.

To simplify working with this API, the Layouts module comes with a base driver class called **FormsElementDriver**. When you derive your element driver from this base class, all you have to do is provide one or more form names that you want to rendered.

Let's see what it takes to refactor the Map driver to use the forms API.

Trying it out: Using the Forms API

The first things to do is to replace the base class of the **MapDriver** class with **FormsElementDriver**, remove the **OnBuildEditor** method and the editor view template (EditorTemplates/Elements/Map.cshtml).

Next, we need to:

1. Override the `DescribeForm` method to programmatically create our editor (or create a separate class that implements the **IFormProvider** interface from Orchard.Forms).
2. Override the **FormNames** property to tell the driver which form to render when editing the Map element.

The following code listing shows the updated version of the **MapDriver** class:

```

using System.Collections.Generic;
using OffTheGrid.Demos.Layouts.Elements;
using Orchard.Forms.Services;
using Orchard.Layouts.Framework.Drivers;
using Orchard.Layouts.Services;

namespace OffTheGrid.Demos.Layouts.Drivers {
    public class MapDriver : FormsElementDriver<Map> {
        public MapDriver(IFormsBasedElementServices formsServices) :
base(formsServices) { }
        protected override IEnumerable<string> FormNames {
            get { yield return "MapEditor"; }
        }

        protected override void DescribeForm(DescribeContext context) {
            context.Form("MapEditor", shapeFactory => {
                var shape = (dynamic)shapeFactory;
                var form = shape.Fieldset(
                    Id: "Map",
                    _Latitude: shape.Textbox(
                        Id: "Latitude",
                        Name: "Latitude", // -> This name needs to match the name
of the Latitude property of the Map class.
                        Title: T("Latitude"),
                        Classes: new[] { "text", "medium" },
                        Description: T("The latitude of the location to show on
the map.")),
                    _Longitude: shape.Textbox(
                        Id: "Longitude",
                        Name: "Longitude", // -> This name needs to match the name
of the Longitude property of the Map class.
                        Title: T("Longitude"),
                        Classes: new[] { "text", "medium" },
                        Description: T("The longitude of the location to show on
the map.")));
                return form;
            });
        }
    }
}

```

There are a few things to keep in mind when using the Forms API with element editors:

1. The form name being returned by the **FormNames** property needs to match the name of the form that is provided when describing the form in the **DescribeForm** method.
2. The **FormsElementDriver** base class implementation stores the form field values in the Data dictionary of the **Element** class.

Since the **Map** class reads and writes from and to this dictionary, we need to make sure the Name values of the “__Latitude” and “__Longitude” text box elements match exactly with the keys into the Data dictionary.

3. Notice that the root shape being returned is not a **Form** shape, but a **Fieldset** shape instead. This is important, because the element editor dialog already renders a form element.

With these changes in place, the editor experience looks exactly the same, but without the need for a Razor view for the editor template.

Providing Additional Display Data

When writing custom elements, there will be times when you need to provide additional information for the display shape template. For example, data coming from an external service that you retrieve from the **OnDisplaying** method of the driver. The Map element example didn't have that requirement, since all the necessary information was provided by the element itself (Latitude and Longitude), but if you wanted to provide additional data, you can do so by setting that data directly onto the Element shape as follows:

```
protected override void OnDisplaying(Map element, ElementDisplayingContext context) {  
    context.ElementShape.MyAdditionalData = "Some additional data";  
}
```

You can then access that additional data from your view like this:

Views/Elements.Map.cshtml:

```
@{  
    Additional data: @Model.AdditionalData  
}
```

Descriptions and Toolbox Icons

When writing custom elements, it is a good practice to provide a description for your element. The description is displayed as a tooltip when hovering over the element in the toolbox. You can provide a description by overriding the **Description** property. You can also specify a custom toolbox icon by overriding the **ToolboxIcon** property on your element class and setting a valid *Font Awesome* icon identifier to be returned.

For example:

```
public override LocalizedString Description => T("Renders a map with a specified  
location.");  
public override string ToolboxIcon => "\uf041";
```

Summary

In this chapter, we have seen how we can extend the list of available elements by implementing our own element classes. We learned about the **Element** class itself as well as the **ElementDriver<T>** class to implement the element's behavior and handle events pertaining that element.

We also learned about implementing element editors using the Forms API provided by the Orchard.Forms module, which is great for basic element editors where there is no need for more advanced editor UIs.

With this knowledge in our arsenal, we have all we need to be able to write any kind of elements, whether they are simple or more complex. It all boils down to implementing an element class and driver, and implementing the various methods on the driver.

Writing custom elements is fun.

13. Writing Container Elements

In the previous chapter, we learned how to write custom elements. However, if you want to write a custom element that can contain other elements, there are quite a few more steps involved. In this chapter, we will unravel all those steps that are involved when creating custom container elements.

Steps to Create a Container Element

When writing custom container elements, you typically perform the following steps:

1. Implement a class that derives from **Container**.
2. Implement a driver for your element.
3. Implement a mapper class that derives from **LayoutModelMapBase<T>** that takes care of mapping values between the layout editor and elements back and forth.
4. Implement a *client-side model* for the element as well as an *Angular directive*.
5. Implement a resource manifest provider that provides the client script to the layout editor.
6. Implement a shape table provider that invokes the resource manager to include the element resources.

Most of the work when creating a custom container element goes into making it work with the layout editor. The primary reason for this is the fact that the layout editor has a client side model of elements, and it needs to know what the type of object is to be used on the client.

The layout editor roughly divides elements into two categories:

- Content elements (Html, Image, Heading, etc.)
- Non-content elements (Canvas, Grid, Row, Column)

The distinction is made as follows: if a given element type declares its own *Model Map*, it means it has a specific client side (JavaScript) representation of that element. If not, the client side representation is always the **Content** JavaScript class.

Layout Model Mappers

“Layout model mapping” is a process that converts a list of server-side **Element** objects into a JSON format that the layout editor can work with, and the other way around: to convert a layout editor data JSON string back into a list of **Element** objects.

The reason we need to do that is because the layout editor requires its own JSON schema to work with elements, so we need a mechanism to serialize elements into that JSON format.

Another way of thinking about layout model mapping is to think of it as a provider for the client-side representation of your custom element.

To implement a model map for your own element, you need to implement the **ILayoutModelMap** interface and provide a DTO to be serialized into JSON, or parse a **JNode** object back into an **Element** instance.

The **ILayoutModelMap** interface has the following members:

Member	Description
Priority	If multiple model maps can convert from and to a given element type, the one with the highest priority is selected to do that job. This is useful in cases such as the ContentModelMap , which can basically map any element that is not a container. So in order to allow other non-container elements to provide their own model map implementation, they need to be able to provide a higher priority.
LayoutElementType	Returns the <i>client-side</i> “class” name of the element. When the layout editor needs to instantiate an element based on its JSON data, it uses this name to instantiate an object of that class.
CanMap	Returns true if this model map implementation can map the specified element, false otherwise.
ToElement	Returns an Element object initialized with the provided JNode data. This is a way for the client-side layout editor to pass in data set by the user and into the element as stored on the server side.

	Examples of this are the HtmlId , HtmlClass and HtmlStyle properties on an Element.
FromElement	The counterpart of ToElement . This method expects implementations to populate the specified JNode with data from the specified Element.

The layout editor allows the user to manipulate element properties in two ways:

1. Via the Element Editor Dialog.
2. Directly from the Layout Editor.

The first way is by invoking the element editor dialog, where element drivers are involved to render the editor UI and handle form submissions. The resulting element data is then sent back as a JSON string to the layout editor via JavaScript and stored in a **Data** property on the client-side model that represents the layout and its individual elements.

The second way is to edit element properties *directly* from the layout editor. This set of properties is not typically the same set as shown in the element editor dialog. Instead, it shows common properties such as **HtmlId**, **HtmlClasses** and **HtmlStyles**. These property values are set on a client-side model representing the layout and its individual elements, so we need a way to include this information when everything gets submitted to the server.

That is where the layout model mappers come into play.

Client Side Support

Implementing the client side story of container elements essentially involves the following four steps:

1. Implement the client-side model of the element
2. Implement the directive (the layout editor is implemented with AngularJS)
3. Implement the template for the directive
4. Register the element with the client-side element factory

Once you got all that in place, you'll be able to add your custom container element onto the canvas, and add any other elements to it.

Let's dive in and see what it looks like when actually implementing a container element.

Trying it out: Writing a Tile element

In this walkthrough, we'll implement a custom container element called **Tile** that enables the user to specify a background image. Once you understand how to implement container elements, you'll be able to create any other type of elements.

The Tile element will have the following features:

- A Tile element can contain any other type of element.
- A Tile element can optionally have a background image.
- A Tile element can optionally have a background size.
- The user will be able to change the background size property using the *quick properties accessor*.

The *quick properties accessor* is the little pop-out window that appears when you click the "Edit [element type] properties (Space)" icon in an element's toolbar, and offer a way for users to quickly change certain properties without having to launch the element editor's dialog. Common to all elements are properties such as **HTML ID**, **CSS Classes**, **CSS Styles** and **Visibility Rule**. However, as we'll see in this walkthrough, custom elements have complete control over this set of properties.

Since the Tile element stores a reference to a content item (the background image), we'll need to make sure that we export the *content item identity*, since the content item id (a primary key value) will be useless when importing. To learn more about content identity, check out Bertrand Leroy's blog post on the matter: <http://weblogs.asp.net/bleroy/identity-in-orchard-import-export>

The Tile Element

First of all, we need to create an element class called **Tile**, which will override the **Category** and **ToolboxIcon** properties, and add two additional properties: the **BackgroundImageId**, which will store the image content item ID that the user selected, and the **BackgroundSize** property, which

will control how the background image gets applied in terms of the **background-size** CSS attribute. The following code listing shows the complete **Tile** class:

```
using Orchard.Layouts.Elements;
using Orchard.Layouts.Helpers;

namespace OffTheGrid.Demos.Layouts.Elements {
    public class Tile : Container {
        public override string Category => "Demo";
        public override string ToolboxIcon => "\uf03e";

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

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

The Tile Driver

The Tile driver will be responsible for the following things:

- Handling the Tile element editor;
- Handle the **OnDisplaying** event to prepare some data when the Tile element is being rendered. We need to load the selected background image by the ID that is stored in the **BackgroundImageId** property.
- Implement *Import/Export* for the Tile in order to export/import the background image content item identity.

The following code shows the complete implementation of the Tile driver:

```

using System.Linq;
using OffTheGrid.Demos.Layouts.Elements;
using OffTheGrid.Demos.Layouts.ViewModels;
using Orchard.ContentManagement;
using Orchard.Layouts.Framework.Display;
using Orchard.Layouts.Framework.Drivers;
using Orchard.Layouts.Helpers;
using Orchard.MediaLibrary.Models;

namespace OffTheGrid.Demos.Layouts.Drivers {
    public class TileDriver : ElementDriver<Tile> {
        private readonly IContentManager _contentManager;

        public TileDriver(IContentManager contentManager) {
            _contentManager = contentManager;
        }

        protected override EditorResult OnBuildEditor(Tile element,
            ElementEditorContext context) {
            var viewModel = new TileViewModel {
                BackgroundImageId = element.BackgroundImageId?.ToString(),
                BackgroundSize = element.BackgroundSize
            };

            // If an Updater is specified,
            // it means the element editor form is being submitted
            // and we need to read and store the submitted data.
            if(context.Updater != null) {
                if (context.Updater.TryUpdateModel
                    (viewModel, context.Prefix, null, null)) {
                    element.BackgroundImageId =
Orchard.Layouts.Elements.ContentItem.Deserialize(viewModel.BackgroundImageId).First
OrDefault();
                    element.BackgroundSize = viewModel.BackgroundSize?.Trim();
                }
            }

            viewModel.BackgroundImage = GetBackgroundImage(element,
VersionOptions.Latest);

            var editorTemplate = context.ShapeFactory.EditorTemplate(
                TemplateName: "Elements/Tile",
                Model: viewModel,
                Prefix: context.Prefix);

            return Editor(context, editorTemplate);
        }

        protected override void OnDisplaying(
            Tile element, ElementDisplayingContext context) {
            var versionOptions = context.DisplayType == "Design"
                ? VersionOptions.Latest
                : VersionOptions.Published;
            Context.ElementShape.BackgroundImage =
                GetBackgroundImage(element, versionOptions);
        }
    }
}

```

```

protected override void OnExporting(
    Tile element,
    ExportElementContext context) {

    // Load the actual background content item.
    var backgroundImage = GetBackgroundImage(element,
VersionOptions.Latest);

    if (backgroundImage == null)
        return;

    // Use the content manager to get the content identities.
    var backgroundImageIdentity =
_contentManager.GetItemMetadata(backgroundImage).Identity;

    // Add the content item identities to the ExportableData dictionary.
    context.ExportableData["BackgroundImage"] =
backgroundImageIdentity.ToString();
}

protected override void OnImporting(Tile element, ImportElementContext
context) {
    // Read the imported content identity from
// the ExportableData dictionary.
    var backgroundImageIdentity =
context.ExportableData.Get("BackgroundImage");

    // Get the imported background content item from
// the ImportContentSession store.
    var backgroundImage =
context.Session.GetItemFromSession(backgroundImageIdentity);

    if (backgroundImage == null)
        return;

    // Get the background content item id (primary key value)
// for each background image.
    var backgroundImageId = backgroundImage.Id;

    // Assign the content id to the BackgroundImageId property so
// that they contain the correct values, instead of the
// automatically imported value.
    element.BackgroundImageId = backgroundImageId;
}

private ImagePart GetBackgroundImage(Tile element, VersionOptions options)
{
    return element.BackgroundImageId != null
        ? _contentManager.Get<ImagePart>(element.BackgroundImageId.Value,
options, QueryHints.Empty.ExpandRecords<MediaPartRecord>())
        : null;
}
}

```

Let's go over that driver method-by-method.

OnBuildEditor

The `OnBuildEditor` method takes care of both displaying the element editor as well as handling post-back. The first 4 lines initialize a new view model used by the editor template:

```
var viewModel = new TileViewModel {  
    BackgroundImageId = element.BackgroundImageId?.ToString(),  
    BackgroundSize = element.BackgroundSize  
};
```

This simply constructs a new **TileViewModel** object and initializes it with existing Tile element data. The reason I introduced a view model is because I need to pass an actual Image content item object to the view, as you'll see.

The `TileViewModel` is defined as follows:

```
using Orchard.MediaLibrary.Models;  
namespace OffTheGrid.Demos.Layouts.ViewModels {  
    public class TileViewModel {  
        public string BackgroundImageId { get; set; }  
        public string BackgroundSize { get; set; }  
        public ImagePart BackgroundImage { get; set; }  
    }  
}
```

In case you're wondering about the **BackgroundImageId** being of type **string** rather than **int**, that's because of the way the **MediaLibraryPicker** shape works, which I'm re-using to render a media item picker. Basically, it stores potentially multiple selected image IDs using a *comma-separated list of ids*.

After initializing the view model, we check to see if we're in "post-back" mode by checking whether the **Updater** is null or not. The Updater is

specified by the framework when the user submitted the form on the element editor dialog, and really is just a reference to a controller. If the Updater is not null, we go ahead and use it to model bind the posted values against our view model like this:

```
// If an Updater is specified, it means the element editor form is being submitted
// and we need to read and store the submitted data.
if(context.Updater != null) {
    if (context.Updater.TryUpdateModel(viewModel, context.Prefix, null, null)) {
        element.BackgroundImageId =
Orchard.Layouts.Elements.ContentItem.Deserialize(viewModel.BackgroundImageId).FirstOrDefault();
        element.BackgroundColor = viewModel.BackgroundColor?.Trim();
    }
}
```

Notice that we're converting the **viewModel.BackgroundImageId** from **string** to **IEnumerable<int>** using the **Deserialize** static method as defined on **Orchard.Layouts.Elements.ContentItem**. The ContentItem element just happened to provide this a convenient method, so I chose to reuse it here. Since the Tile only supports a single background image, we call **FirstOrDefault** to get the first selected image, if any.

After handling the post-back scenario, the method continues by loading the selected background image using the content manager:

```
viewModel.BackgroundImage = GetBackgroundImage(element, VersionOptions.Latest);
```

We need the Image content item to initialize the **MediaLibraryPicker** shape which will be used by the Tile's editor shape template.

The **GetBackgroundImage** method is a private convenience method implemented as follows:

```
private ImagePart GetBackgroundImage(Tile element, VersionOptions options) {
    return element.BackgroundImageId != null
        ? _contentManager.Get<ImagePart>(element.BackgroundImageId.Value, options,
QueryHints.Empty.ExpandRecords<MediaPartRecord>())
    : null;
}
```

```

        : null;
    }

```

This basically uses the content manager to load the image content item using the specified version options. It also “expands” the **MediaPartRecord**, since we'll be accessing its **MediaUrl**, so it makes sense to join the **ImagePartRecord** and **MediaPartRecord** tables when loading the image.

Finally, we create a new shape of type **EditorTemplate**, where we provide all the information it needs.

The **EditorTemplate** shape is configured to use the *Elements/Tile* template, which maps to the Razor view *Views/EditorTemplates/Elements/Tile.cshtml*:

```

@using Orchard.ContentManagement
@model OffTheGrid.Demos.Layouts.ViewModels.TileViewModel

@Display.MediaLibraryPicker(
    FieldName: Html.FieldNameFor(m => m.BackgroundImageId),
    DisplayName: T("Background Images").ToString(),
    Multiple: true,
    Required: false,
    Hint: T("Optionally select one or more background images. Additional
background images may be used for specific breakpoints, depending on the current
theme's implementation.").ToString(),
    ContentItems: Model.BackgroundImage != null ? new ContentItem[] {
Model.BackgroundImage.ContentItem } : null,
    PromptOnNavigate: false,
    ShowSaveWarning: false)

<fieldset>
    <div class="form-group">
        @Html.LabelFor(m => m.BackgroundColor, T("Background Color"))
        @Html.TextBoxFor(m => m.BackgroundColor, new { @class = "text medium" })
        @Html.Hint(T("Optionally specify a background-color value. Examples: auto,
[length], cover, contain, initial and inherit. Substitute [length] with a width
and optionally a height, either in pixels or percentage."))
    </div>
</fieldset>

```

The above code listing shows the usage of the **MediaLibraryPicker** shape as provided by the Orchard.MediaLibray module. Notice especially the **ContentItems** property initialization, which expects an enumerable of **ContentItem** objects.

OnDisplaying

The **OnDisplaying** method is called whenever the element is about to be rendered, and gives us a chance to prepare some data or objects for easy consumption from the view. In our case, we need to load the selected background image, which we pass into the element shape. Since the **OnDisplaying** event is triggered for both the front-end and layout editor back-end, we get a chance to optimize the data for both display types.

Whenever the layout editor renders the elements, it uses the “Design” display type. This enables you to provide tailor-made views optimized for being displayed as part of the layout editor.

Since the background image is a content item, it can potentially be saved as a draft. This means that you'll want to get the latest version when being rendered as part of the layout editor, but only the published version when being rendered on the front-end:

```
var versionOptions = context.DisplayType == "Design" ? VersionOptions.Latest :  
VersionOptions.Published;  
context.ElementShape.BackgroundImage = GetBackgroundImage(element,  
versionOptions);
```

OnExporting and OnImporting

OnExporting and **OnImporting** are invoked when the content item is being exported or imported, and gives you an opportunity to include any additional information about the element that you need when being imported again.

The reason we need to implement these methods here is because we are referencing a content item by its primary key value (ID), which may be of a different value when content is exported and then imported into another database. Therefore we need to export the *content identity* of the image content item, which we can use during import.

The exporting code looks like this:

```
// Use the content manager to get the content identities.
var backgroundImageIdentity =
_contentManager.GetItemMetadata(backgroundImage).Identity;

// Add the content item identities to the ExportableData dictionary.
context.ExportableData["BackgroundImage"] =
backgroundImageIdentity.ToString();
```

Notice the usage of the **ExportableData** dictionary, which is where you store information to be exported and imported.

The importing code mirrors the exporting code:

```
// Read the imported content identity from the ExportableData dictionary.
var backgroundImageIdentity = context.ExportableData.Get("BackgroundImage");

// Get the imported background content item from the
// ImportContentSession store.
var backgroundImage =
context.Session.GetItemFromSession(backgroundImageIdentity);

if (backgroundImage == null)
    return;

// Get the background content item id (primary key value)
// for the background image.
var backgroundImageId = backgroundImage.Id;

// Assign the content id to the BackgroundImageId property so
// that they contain the correct values, instead of the automatically
// imported value.
element.BackgroundImageId = backgroundImageId;
}
```

The **context.Session** object maintains a dictionary of imported content items, keyed by content identity. Once we get our hands on the

referenced content item, we use its primary key value (ID) to update the element's **BackgroundImageId**.

The Tile Element Shape Template

In order for the Tile element to be able to render its child elements, It needs its own shape template, so create a file called `Tile.cshtml` in the `Views/Elements` folder. Its contents should look like this:

```
@using Orchard.ContentManagement;
@using Orchard.DisplayManagement.Shapes
@using Orchard.Layouts.Helpers
@using Orchard.MediaLibrary.Models
@using OffTheGrid.Demos.Layouts.Elements
@{
    var tagBuilder =
(OrchardTagBuilder)TagBuilderExtensions.CreateElementTagBuilder(Model);
    var element = (Tile)Model.Element;
    var backgroundImage = (ImagePart)Model.BackgroundImage;

    if(backgroundImage != null) {
        var mediaPart = backgroundImage.As<MediaPart>();
        var backgroundSize = !String.IsNullOrEmpty(element.BackgroundSize) ?
element.BackgroundSize : "cover";
        tagBuilder.Attributes["style"] = String.Format("background-image:
url('{0}'); background-size: {1};", mediaPart.MediaUrl, backgroundSize);
    }
}
@tagBuilder.StartElement
@DisplayChildren(Model)
@tagBuilder.EndElement
```

The first thing this view does is creating an **OrchardTagBuilder**. Then we check if we have a non-null **BackgroundImage** available, which is set from the **OnDisplaying** method in the driver.

If there is a **BackgroundImage** set, we add a **style** attribute to our tag with the background image URL and background image size, falling back to “cover” as a default size.

The rest of the code renders the tag, and any child elements. Child element shapes will be added to the `Element` shape automatically by the `Layouts` module, so all you have to do is render them using `@DisplayChildren`.

The Client Side

So far the process of writing a container element hasn't been that different from creating a regular element. The key differences are the fact that we derived the `Tile` class from `Container` instead of `Element` and some additional code to deal with the `MediaLibraryPicker` shape and additional import/export code because of that.

Implementing Containment

Surely enough, at this point the `Tile` element works like any other element. Except for the fact that you can't currently add elements to the `Tile`, despite the fact that it derives from `Container`. From the server-side's point of view, you could programmatically add child elements, but to enable the user to drag and drop elements using the Layout Editor, additional work needs to be done.

Client-Side Assets

When implementing custom elements with a customized client-side representation, it is important to realize that the layout editor is implemented using Angular and that it relies heavily on model binding. At the moment of this writing, there is no generic container element class that we could reuse unfortunately, so we'll have to do quite some coding ourselves.

Regular (non-container) elements do have a generic client-side representation, which is the **Content** element model. Future versions of Orchard may provide a generic **Container** representation to implement custom containers, but until then we'll have to implement our own.

Although not required, I chose to write JavaScript and Less files that take advantage of the Gulp pipeline support provided with Orchard.

Gulp and Assets.json

To see how the Gulp pipeline works, let's go ahead and create the following file & folder structure in the module:

- Assets
 - Elements
 - Tile
 - Directive.js
 - Model.js
 - Style.less
- Assets.json

These files won't be referenced by our views directly. Instead, we'll provide a configuration file called *Assets.json* that Gulp will use to compile, combine and minify the assets and output them to the *Scripts* and *Styles* folder.

The *Assets.json* file needs to live in the root of the module, and will have the following contents:

```
[
  {
    "inputs": [
      "Assets/Elements/Tile/Style.less"
    ],
    "output": "Styles/TileElement.css"
  },
  {
    "inputs": [
      "Assets/Elements/Tile/Directive.js",
      "Assets/Elements/Tile/Model.js"
    ],
    "output": "Scripts/TileElement.js"
  }
]
```

If you have worked with Gulp before, the structure of this JSON file should look quite familiar. It's basically an array with inputs/output objects, where the inputs are compiled, combined, minified and stored in the output file. The specified paths are relative to the module's root.

The above Assets.json configuration will generate four files:

- /Scripts/TileElement.js
- /Scripts/TileElement.min.js

And

- /Styles/TileElement.css
- /Styles/TileElement.min.css

The easiest way to execute Gulp is to use the Task Runner Explorer. Simply right-click on the **build** task and hit **Run**. When you do this for the first time, the resulting files will be created, but not be made part of the project automatically.

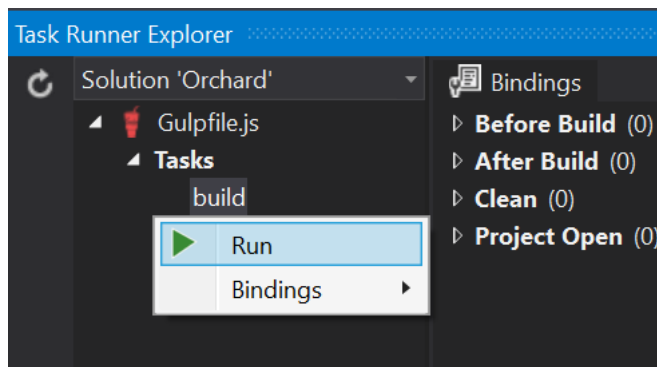


Figure 13-1 - The Task Runner Explorer in Visual Studio 2015.

When the Task Runner Explorer shows the message “Failed to load. See output window”, this is usually an indication that you haven't installed the NodeJS packages. This is easily fixed by opening the *Package.json* file in the *Solution Items/Gulp* solution folder, and simply saving that file. When you do, Visual Studio will start downloading NodeJS packages, including Gulp. Gulp is used to compile, minify and bundle files such as CSS, LESS, JavaScript and TypeScript. Installing the various NodeJS packages may take a few minutes, so keep a close eye on the status bar (which should read “Installing packages” while installing packages, and “Installing packages complete” when done. Once the packages have been installed, click the refresh icon in the Task Runner Explorer and wait a second for it to refresh. Now you should see the **Tasks** and its child nodes **build**, **rebuild** and **watch**. Right-click on **build** and then left-click **Run** to have your *Assets.json* file executed.

With that in place, we are ready to provide actual contents to each asset file. We'll start with *Assets/Elements/Tile/Model.js*.

The Tile Model

The following code shows the minimum amount of boilerplate code required to make the Tile element a container element:

```

var LayoutEditor;

(function (LayoutEditor) {

    // The constructor.
    LayoutEditor.Tile = function (data, contentType, htmlId, htmlClass, htmlStyle,
isTemplated, rule, hasEditor, children) {
        var self = this;

        // Inherit from the Element base class.
        LayoutEditor.Element.call(self, "Tile", data, htmlId, htmlClass,
htmlStyle, isTemplated, rule);

        // Inherit from the Container base class.
        LayoutEditor.Container.call(self, ["Canvas", "Grid", "Content"],
children);

        // This Tile element is containable, which means it can be added
        // to any container, including Tiles.
        self.isContainable = true;

        // Used by the layout editor to determine if it should launch
        // the element editor dialog when creating new Tile elements.
        // Also used by our "LayoutEditor.Template.Tile.cshtml" view
        // that is used as the layout-tile directive's template.
        self.hasEditor = hasEditor;

        // The element type name, which is sent back to the
        // element editor controller when being edited.
        self.contentType = contentType;

        // The "layout-common-holder" CSS class is used by the layout editor
        // to identify drop targets.
        self.dropTargetClass = "layout-common-holder";

        // Implements the toObject serialization function.
        // This is called when the layout is being serialized into JSON.
        var toObject = self.toObject; // Get a reference to the base function.
        self.toObject = function () {
            var result = toObject(); // Invoke the base implementation.
            result.children = self.childrenToObject();
            return result;
        };
    };

    // Registers the factory function with the element factory.
    LayoutEditor.registerFactory("Tile", function (value) {
        var tile = new LayoutEditor.Tile(
            value.data,
            value.contentType,
            value.htmlId,
            value.htmlClass,
            value.htmlStyle,
            value.isTemplated,
            value.rule,
            value.hasEditor,
            value.children);
    });
});

```

```

        LayoutEditor.childrenFrom(value.children));

    // Initializes the toolbox specific properties.
    tile.toolboxIcon = value.toolboxIcon;
    tile.toolboxLabel = value.toolboxLabel;
    tile.toolboxDescription = value.toolboxDescription;

    return tile;
});
})(LayoutEditor || (LayoutEditor = {}));

```

It is not complex code, but there are quite a few things that aren't obvious without proper documentation. The inline comments should be self-explanatory, but I will go over some of the more interesting aspects section by section.

```

// Inherit from the Element base class.
LayoutEditor.Element.call(self, "Tile", data, htmlId, htmlClass, htmlStyle,
isTemplated, rule);

```

As the comment indicates, this makes the **Tile** class inherit from the **Element** base class by calling into the **LayoutEditor.Element** function, which will add all the required members. The second parameter being passed in, “**Tile**”, is the client side type name of the element. Mind you, this is not the same value as the server side element type name! Instead, this value must be the same as the value returned from **LayoutElementType** property of the model map class, which we’ll look into after we’re done with the client-side story.

In addition to inheriting from **Element**, the **Tile** class also inherits from **Container**:

```

// Inherit from the Container base class.
LayoutEditor.Container.call(self, ["Canvas", "Grid", "Content"], children);

```

That call turns the **Tile** into an actual container so it can receive elements. The second argument being passed in is an array of element types that the **Tile** element can contain. However, this list does not need to

include every element type out there, since any element whose **isContainable** field is set to **true** can be added to any container. This means that the Tile element itself for example can be added to other Tile elements, since we're setting exactly that property on the next line:

```
// This Tile element is containable, which means it can be added to any container,
including Tiles.
self.isContainable = true;
```

The following line is key to turn our element into a drop target:

```
// The "layout-common-holder" CSS class is used by the layout editor to identify
drop targets.
self.dropTargetClass = "layout-common-holder";
```

I personally think this CSS class value should have been set by default, but for now we'll just have to do it ourselves.

The same goes for the next and final block of code of the constructor function:

```
// Implements the toObject serialization function.
// This is called when the layout is being serialized into JSON.
var toObject = self.toObject; // Get a reference to the default implementation
before we override it.
self.toObject = function () {
    var result = toObject(); // Invoke the original (base) implementation.
    result.children = self.childrenToObject();
    return result;
};
```

The above code overrides the **toObject** function in order to have its children be serialized as well. This code is so generic that it should probably be handled by the Container base class. But at this point in time, we'll just have to do this also ourselves.

The layout editor invokes **toObject** on all elements whenever the elements are being serialized into JSON.

The final piece of the *Model.js* file registers the Tile class with the layout editor's element factory:

```
// Registers the factory function with the element factory.
LayoutEditor.registerFactory("Tile", function (value) {
    var tile = new LayoutEditor.Tile(
        value.data,
        value.contentType,
        value.htmlId,
        value.htmlClass,
        value.htmlStyle,
        value.isTemplated,
        value.rule,
        value.hasEditor,
        LayoutEditor.childrenFrom(value.children));

    return tile;
});
```

The first argument is again the client-side element type (“Tile”) which serves as a key into an internal dictionary. The second argument takes a function that is invoked by the factory when a new Tile element needs to be instantiated. This is where we “new up” the Tile class, passing in various bits and pieces of the specified **value** argument. This value is an object that contains all the values provided from the (yet to be created) model map class.

The Tile Directive

In addition to implementing a client-side model for the tile element, we also need to implement an Angular directive so that Angular templates can render the elements.

The Tile directive *Directive.js* requires the following code in order for it to function as a container:

```
angular
    .module("LayoutEditor")
    .directive("orcLayoutTile", ["scopeConfigurator", "environment",
        function (scopeConfigurator, environment) {
```

```

        return {
            restrict: "E",
            scope: { element: "=" },
            controller: ["$scope", "$element", "$attrs",
                function ($scope, $element, $attrs) {
                    scopeConfigurator.configureForElement($scope, $element);
                    scopeConfigurator.configureForContainer($scope, $element);
                    $scope.sortableOptions["axis"] = "y";
                }
            ],
            templateUrl: environment.templateUrl("Tile"),
            replace: true
        };
    }
}
]);

```

A few important aspects worth mentioning:

For starters, it is important that the directive is defined as part of the **“LayoutEditor”** Angular module, and that the directive name starts with **“orcLayout”**, followed by the client-side element type name **“Tile”**. This is a convention used by the **orcLayoutChild** directive that is provided by the Layouts module.

Next, we need to setup the controller function such that it configures the Angular **\$scope** (which is more or less the model of the directive) using the injected **scopeConfigurator**, which adds all the necessary functions to the scope and the element. Also, since we're configuring a container, we need to configure the **sortableOptions**, which is specific to the **Sortable jQueryUI** widget used by the layout editor to implement drag & drop.

Finally, we need to configure the directive's template, which we do using the injected **environment** service and its **templateUrl** function. This function constructs a relative path to the **Template controller** of the Layouts module, which returns a view based on the specified parameter. When passing in **“Tile”**, the resulting template URL will become: **“Admin/Layouts/Template/Get/Tile”**. The **TemplateController** creates a

shape based on the element name being passed in using the following format: “LayoutEditor_Template_{0}”.

This means that you’ll have to provide a Shape template named “LayoutEditor.Template.Tile.cshtml” for the Tile’s directive.

The Tile Directive Template

After implementing the element's directive, we need to provide the directive's template, which is implemented as a shape template. Create a Razor file called “LayoutEditor.Template.Tile.cshtml” with the following markup:

```
<div class="layout-element-wrapper layout-element-background" ng-class="{ 'layout-
container-empty': getShowChildrenPlaceholder()}">
  <ul class="layout-panel layout-panel-main">
    <li class="layout-panel-item layout-panel-label">Tile</li>
    <li class="layout-panel-item layout-panel-action layout-panel-action-edit"
ng-show="{{element.hasEditor}}" title="Edit tile (Enter)" ng-click="edit()"><i
class="fa fa-code"></i></li>
    @Display(New.LayoutEditor_Template_Properties(ElementTypeName: "tile"))
    <li class="layout-panel-item layout-panel-action" title="@T("Delete tile
(Del)")" ng-click="delete(element)" ng-show="element.canDelete()"><i class="fa fa-
remove"></i></li>
    <li class="layout-panel-item layout-panel-action" title="@T("Move tile up
(Ctrl+Up)")" ng-click="element.moveUp()" ng-show="element.canMoveUp()"><i
class="fa fa-chevron-up"></i></li>
    <li class="layout-panel-item layout-panel-action" title="@T("Move tile
down (Ctrl+Down)")" ng-click="element.moveDown()" ng-
show="element.canMoveDown()"><i class="fa fa-chevron-down"></i></li>
  </ul>
  <div class="layout-container-children-placeholder"
style="{{element.getTemplateStyles()}}">
    @T("Drag an element from the toolbox and drop it here to add content.")
  </div>
  @Display(New.LayoutEditor_Template_Children())
</div>
```

All of the Angular bindings you see in this file are bindings against methods and fields of the **\$scope**, which is configured by the Tile directive code. This template takes care of the entire rendering of the element's representation in the layout editor. This means that if we were to add

additional information to our Tile model, we could bind that information with this template. This is how you can make our template more specific to the Tile element, which we'll see later on.

Registering the Tile Assets

Although we created various Tile assets and configured Gulp to generate their output files, we haven't actually included those files anywhere yet, and the Layouts module isn't doing that for us automatically. So how do we take care of that?

Your initial thought may be to simply include those files from the Tile shape templates. But here's the issue with that: those templates are rendered *after the layout editor has been rendered and using AJAX calls*. So even though the **Script.Include** and **Style.Include** calls would register the assets, it won't do you any good since the main page will already have been rendered, and the HTML being sent back is just the HTML output of a single element, not the entire HTML document. So what is one to do?

We need a way to register the assets *at the same time the layout editor is rendered*. This will cause all elements' assets to be rendered during the initial page load, which is good.

Unfortunately, there's no formal extensibility point or event that we can handle to register those resources.

So we'll just do the next best thing: implement an **IShapeTableProvider** and handle the **OnDisplaying** event for the **EditorTemplate** shape, and check if its **TemplateName** value equals "Parts.Layout", since that's the template name used by the **LayoutPartDriver** when creating the layout editor object.

> This is sadly enough a major limitation currently with the layout editor in case it's used outside the context of the **LayoutPart**. For example, if you were to manually construct the **LayoutEditor** object yourself using the **ILayoutEditorFactory** and then render that object using the **Html.Editor** HTML helper, the layout editor would be rendered, but the handler created in the shape table provider would not be triggered, since you would not be rendering any **EditorTemplate** shape. So what do we do about it? Create a GitHub issue for it of course: <https://github.com/OrchardCMS/Orchard/issues/6221>.

Let's go ahead and create the following class in the *Handlers* folder:

```
using Orchard.DisplayManagement.Descriptors;
using Orchard.Environment;
using Orchard.UI.Resources;

namespace OffTheGrid.Demos.Layouts.Handlers {
    public class TileResourceRegistrations : IShapeTableProvider {
        private readonly Work<IResourceManager> _resourceManager;
        public TileResourceRegistrations(Work<IResourceManager> resourceManager) {
            _resourceManager = resourceManager;
        }

        public void Discover(ShapeTableBuilder builder) {
            builder.Describe("EditorTemplate").OnDisplaying(context => {
                if (context.Shape.TemplateName != "Parts.Layout")
                    return;

                _resourceManager.Value.Require("stylesheet", "TileElement");
                _resourceManager.Value.Require("script", "TileElement");
            });
        }
    }
}
```

What we're doing here is adding a handler for the **OnDisplaying** event of the **EditorTemplate** shape. If the template being displayed is the "Parts.Layout" template, we register two resources with the injected resource manager:

- “TileElement” stylesheet
- “TileElement” script

We haven't defined those resources yet, but we'll get to that in a bit. You may be wondering why I didn't use the **Include** method of the resource manager instead of the **Require** method, since then I wouldn't have to create a resource manifest provider that defines these resources. Well, the reason is that I need to ensure that the layout editor scripts are included *before* the Tile scripts, and the only way to ensure that is to have those resources depend on the layout editor's resources. Since we can't declare that dependency directly, we will have to rely on a resource manifest provider to do so.

The Resource Manifest Provider

Let's go ahead and create the following class in a new folder called *ResourceManifests*:

```
using Orchard.UI.Resources;

namespace OffTheGrid.Demos.Layouts.Handlers {
    public class TileResourceManifest : IResourceManifestProvider {
        public void BuildManifests(ResourceManifestBuilder builder) {
            var manifest = builder.Add();
            manifest.DefineStyle("TileElement").SetUrl("TileElement.min.css",
"TileElement.css");
            manifest.DefineScript("TileElement").SetUrl("TileElement.min.js",
"TileElement.js").SetDependencies("Layouts.LayoutEditor");
        }
    }
}
```

Nothing fancy here, but notice the dependency on the “Layouts.LayoutEditor” resource, which is provided by the Layouts module.

The Tile Model Map

Now that we have the server-side **Tile** element and client-side representation in place, it is time to implement the model map.

Fortunately, this is easy. As mentioned earlier, you need to implement the **ILayoutModelMap** interface, but to make that process easier, a base class called **LayoutModelMapBase<T>** has been made available which we'll use.

All we need to do is create a class that derives from **LayoutModelMapBase<T>**, substituting **T** with **Tile**:

```
using OffTheGrid.Demos.Layouts.Elements;
using Orchard.Layouts.Services;

namespace OffTheGrid.Demos.Layouts.Handlers {
    public class TileModelMap : LayoutModelMapBase<Tile> {
    }
}
```

I mentioned earlier that the model map needs to implement the **LayoutElementType** property by returning the element type name ("Tile"). This is done for us by the base class, which is implemented as follows:

```
public virtual string LayoutElementType { get { return typeof(T).Name; } }
```

Since **T** is of type **Tile**, the resulting string will be "Tile". Exactly what we want.

Test Drive

When you launch the site now, you should be able to add the **Tile** element to the canvas and provide a background image and size, and see

the element appear on the canvas. You should also be able to add elements to the Tile element.

When you view the page on the front-end, you should see the Tile element with the configured background image and contained Html element rendered as expected.

Tile Sample

Sunday, March 13, 2016 10:57:00 AM

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur hendrerit sapien mi, at pellentesque libero mattis et. In vitae magna ut tellus congue molestie eget non sem. Nulla vel mauris malesuada, varius est in, efficitur felis. Fusce dapibus, nisl at accumsan rutrum, felis ante viverra arcu, eget sodales orci leo ut orci. Integer vestibulum enim eget ornare ullamcorper. Aliquam tempor vulputate justo, vitae sollicitudin magna lacinia egestas. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Phasellus et ex venenatis, interdum odio eget, viverra augue.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur hendrerit sapien mi, at pellentesque libero mattis et. In vitae magna ut tellus congue molestie eget non sem. Nulla vel mauris malesuada, varius est in, efficitur felis. Fusce dapibus, nisl at accumsan rutrum, felis ante viverra arcu, eget sodales orci leo ut orci. Integer vestibulum enim eget ornare ullamcorper. Aliquam tempor vulputate justo, vitae sollicitudin magna lacinia egestas. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Phasellus et ex venenatis, interdum odio eget, viverra augue.



Figure 13-2 - A Grid element inside the Tile element with a background image.

Now, the reason we're not seeing the selected background image in the layout editor is because we're not rendering the element using the **Elements/Tile.cshtml** template. Instead, the Angular directive's template use now used, which is necessary to render its children.

> The reason regular *Content* client-side elements are able to render their templates using the *Design* display type is because those elements are actually pre-rendered on the server side. The resulting HTML is then passed into the **Content** client-side model, and used by its Angular directive's template to display it. We could in principle do the same for the **Tile** element, if only if it didn't need the ability to receive child elements from drag & drop operations for example.

All this means is that instead of relying on the *Elements/Tile.cshtml* shape template to render the background, we'll just have to render it from the directive's template. This does require us to provide the *image URL* to the client-side **Tile** model and update the directive's template to model-bind against that value.

And while we're at it, we may just as well add support for editing the background size property of the element from the property editor, so that the user doesn't have to launch the element's editor dialog to change the background size. Instead, they would be able to do so directly from the layout editor and instantly see the result.

Let's see how that works.

Updating TileModelMap

The first thing we need to do is get the background image URL and background image size down to our client-side **Tile** model. To do so, update the **TileModelMap** class as follows:

```

using Newtonsoft.Json.Linq;
using OffTheGrid.Demos.Layouts.Elements;
using Orchard.ContentManagement;
using Orchard.Layouts.Services;
using Orchard.MediaLibrary.Models;

namespace OffTheGrid.Demos.Layouts.Handlers {
    public class TileModelMap : LayoutModelMapBase<Tile> {
        private readonly IContentManager _contentManager;

        public TileModelMap(IContentManager contentManager) {
            _contentManager = contentManager;
        }

        public override void FromElement(Tile element, DescribeElementsContext
describeContext, JToken node) {
            base.FromElement(element, describeContext, node);

            var backgroundImage = element.BackgroundImageId != null
                ? _contentManager.Get<ImagePart>(element.BackgroundImageId.Value)
                : default(ImagePart);
            var backgroundImageUrl = backgroundImage?.As<MediaPart>().MediaUrl;

            node["backgroundUrl"] = backgroundImageUrl;
            node["backgroundSize"] = element.BackgroundSize;
        }

        protected override void ToElement(Tile element, JToken node) {
            base.ToElement(element, node);

            element.BackgroundSize = (string)node["backgroundSize"];
        }
    }
}

```

The above code implements the **FromElement** and **ToElement** methods to map the background image URL and background size properties. Remember, these values are sent to and received from the client-side representation of the Tile element, which we'll update next.

Updating Tile/Model.js

Since we want to model-bind the background image URL and background size properties with the Tile element directive, we'll need to

stick those values into our Tile model. I highlighted the changed code in the updated Tile/Model.js file:

```
var LayoutEditor;
(function (LayoutEditor) {

    LayoutEditor.Tile = function (data, contentType, htmlId, htmlClass, htmlStyle,
isTemplated, rule, hasEditor, backgroundUrl, backgroundColor, children) {
        var self = this;

        // Inherit from the Element base class.
        LayoutEditor.Element.call(self, "Tile", data, htmlId, htmlClass,
htmlStyle, isTemplated, rule);

        // Inherit from the Container base class.
        LayoutEditor.Container.call(self, ["Canvas", "Grid", "Content"],
children);

        // This Tile element is containable, which means it can be added to any
container, including Tiles.
        self.isContainable = true;

        // Used by the layout editor to determine if it should launch
        // the element editor dialog when creating new Tile elements.
        // Also used by our "LayoutEditor.Template.Tile.cshtml" view that is used
as the layout-tile directive's template.
        self.hasEditor = hasEditor;

        // The element type name, which is sent back to the element editor
controller when being edited.
        self.contentType = contentType;

        // The "layout-common-holder" CSS class is used by the layout editor to
identify drop targets.
        self.dropTargetClass = "layout-common-holder";

        // The configured background image URL and background size.
self.backgroundUrl = backgroundUrl;
self.backgroundColor = backgroundColor;

        // Implements the toObject serialization function.
        // This is called when the layout is being serialized into JSON.
        var toObject = self.toObject; // Get a reference to the default
implementation before we override it.
        self.toObject = function () {
            var result = toObject(); // Invoke the original (base) implementation.
            result.children = self.childrenToObject();
result.backgroundUrl = self.backgroundUrl;
result.backgroundColor = self.backgroundColor;
            return result;
        };

        // Override the getEditorObject so we can include our backgroundColor
property.
    });
});
```

```

// This is called when the element editor dialog is invoked and we need to
// pass in the client side values.
var getEditorObjectBase = this.getEditorObject;
this.getEditorObject = function () {
    var props = getEditorObjectBase();
    props.BackgroundColor = self.backgroundColor;
    return props;
}

// Executed after the element editor dialog closes.
this.applyElementEditorModel = function (data) {
    self.backgroundColor = data.backgroundColor;
    self.backgroundColor = data.backgroundColor;
    self.applyBackground();
}

this.hasBackground = function () {
    return self.backgroundColor && self.backgroundColor.length > 0;
};

this.applyBackground = function () {
    if (self.hasBackground()) {
        var styles = {
            "background-image": "url('" + self.backgroundColor + "')",
            "background-size": self.backgroundColor &&
self.backgroundColor.length > 0 ? self.backgroundColor : "cover"
        };

        if (self.children.length == 0)
            self.templateStyles = styles;
        else
            self.containerTemplateStyles = styles;
    }
    else {
        self.templateStyles = {};
        self.containerTemplateStyles = {};
    }
}

self.applyBackground();
};

// Registers the factory function with the element factory.
LayoutEditor.registerFactory("Tile", function (value) {
    return new LayoutEditor.Tile(
        value.data,
        value.contentType,
        value.htmlId,
        value.htmlClass,
        value.htmlStyle,
        value.isTemplated,
        value.rule,
        value.hasEditor,
        value.backgroundColor,
        value.backgroundColor,
        LayoutEditor.childrenFrom(value.children));
});

```

```
});  
})(LayoutEditor || (LayoutEditor = {}));
```

Let's go over each section that's been added from top to bottom:

```
LayoutEditor.Tile = function (data, contentType, htmlId, htmlClass, htmlStyle,  
isTemplated, rule, hasEditor, backgroundImage, backgroundSize, children) {
```

Notice the addition of the **backgroundUrl** and **backgroundSize** parameters, which are stored as fields here:

```
// The configured background image URL and background size.  
self.backgroundUrl = backgroundImage;  
self.backgroundSize = backgroundSize;
```

Since we want to allow the user to change the background size directly from the layout editor, we'll need to ensure that this value gets serialized, which is taken care of as follows:

```
// Implements the toObject serialization function.  
// This is called when the layout is being serialized into JSON.  
var toObject = self.toObject; // Get a reference to the default implementation  
before we override it.  
self.toObject = function () {  
    var result = toObject(); // Invoke the original (base) implementation.  
    result.children = self.childrenToObject();  
    result.backgroundUrl = self.backgroundUrl;  
    result.backgroundSize = self.backgroundSize;  
    return result;  
};
```

The same goes for the **backgroundUrl** property. Although the user can't change this value directly, the layout editor uses the *serialization mechanism for copy & paste*. If we don't include the **backgroundUrl** as part of this serialization process, it will appear as if we “lost” the background when the user copies and pastes a Tile element (although in reality the selected background ID will still be part of the element's Data property).

The following code handles the scenario where the user invokes the element's editor dialog. Since the user can change the background size directly from the layout editor, we should pass along that value when the element editor is invoked. To do that, we need to make sure the value becomes part of the object created by the **getEditorObject** function:

```
// Override the getEditorObject so we can include our backgroundSize property.
// This is called when the element editor dialog is being invoked and we need to
// pass in the client side values.
var getEditorObjectBase = this.getEditorObject;
this.getEditorObject = function () {
    var props = getEditorObjectBase();
    props.BackgroundSize = self.backgroundSize;
    return props;
}
```

Notice the name **BackgroundSize** – this needs to exactly match the key as used by the Tile element C# class for its **BackgroundSize** property. This editor object is converted into a dictionary and merged into the element's Data dictionary when rendering the element's editor dialog.

We also need to handle the reversed scenario, where the user changes the background size value from the element editor dialog screen and then saves that change. We'll want to update the client-side Tile model's **backgroundSize** as well as **backgroundUrl** field to reflect any changes made. That's what the following code is for:

```
// Executed after the element editor dialog closes.
this.applyElementEditorModel = function (data) {
    self.backgroundUrl = data.backgroundUrl;
    self.backgroundSize = data.backgroundSize;
    self.applyBackground();
}
```

The **applyElementEditorModel** is called by the layout editor when the element editor dialog is closed using the *Save* command. In our implementation, we're calling a function called **applyBackground**:


```

this.hasBackground = function () {
    return self.backgroundUrl && self.backgroundUrl.length > 0;
};

this.applyBackground = function () {
    if (self.hasBackground()) {
        var styles = {
            "background-image": "url('" + self.backgroundUrl + "')",
            "background-size": self.backgroundSize && self.backgroundSize.length >
0 ? self.backgroundSize : "cover"
        };

        if (self.children.length == 0)
            self.templateStyles = styles;
        else
            self.containerTemplateStyles = styles;
    }
    else {
        self.templateStyles = {};
        self.containerTemplateStyles = {};
    }
}

```

What this code does is prepare a JSON object called **styles**, and then depending on the length of the **children** array, assigns it to either the **templateStyles** or the **containerTemplateStyles** field. But what are those things?

Looking back at the “Views/LayoutEditor.Template.Tile.cshtml” view, notice that we're binding the **style** attribute on the children placeholder <div> element against a function called **getTemplateStyles()** here:

```

<div class="layout-container-children-placeholder"
style="{{element.getTemplateStyles()}}">
    @T("Drag an element from the toolbox and drop it here to add content.")
</div>

```

That function is defined by the client-side **Element** model, which converts the **templateStyles** JSON object into a CSS string.

Similarly, the client-side **Container** model provides a function called **getContainerTemplateStyles()** that turns the **containerTemplateStyles**

JSON object into a CSS string as well. That value is then bound against some `<div>` element in the “`LayoutEditor.Template.Children.cshtml`” view provided by the Layouts module, which looks like this:

```
<div class="layout-children clearfix" ng-model="element.children" ui-sortable="sortableOptions" style="{{element.getContainerTemplateStyles()}}">
  <div class="clearfix" ng-repeat="child in element.children" ng-class="getClasses(child)" ng-mouseenter="child.setIsActive(true)" ng-mouseleave="child.setIsActive(false)" ng-click="click(child, $event)" tabindex="{{{$id}}}">
    <orc-layout-child element="child" />
  </div>
</div>
```

So essentially, the **templateStyles** and **containerTemplateStyles** fields allow us to provide styles on two different places on the element directive template, which we use to set a background image and background size.

The reason we're setting both fields is because the first one is only rendered if our element does not contain any elements, and the second one is only rendered if there is at least one child element.

Updating Tile/Directive.js

Although not strictly necessary, we should update the Tile directive file as well. The reason being that when the user changes the background size property from the property window of the element, ideally we would want to see that change reflected instantly. Although that text field is bound against the **backgroundSize** field of the element, that is not enough to reflect the change, since setting the background image is done by the **applyBackground** function.

So let's go ahead and add a *linker* to our directive:

```
link: function ($scope, $element, $attrs) {
    $element.on("change", "[ng-model='element.backgroundSize']", function () {
        $scope.element.applyBackground();
    });
}
```

This linker binds the **change** event of the text field bound against the **element.backgroundSize** property. Whenever the contents change of that text field, we call the **applyBackground** function on the bound element.

Updating LayoutEditor.Template.Tile.cshtml

Finally, we'll also need to update the directive's template so that the property editor actually displays a text field for the background size property. The easiest way to do that is by providing a list of **LayoutEditorPropertiesItem** to the **LayoutEditor_Template_Properties** shape that's being used by the directive template. This enables us to simply declare additional property editors without having to duplicate existing ones and without having to provide markup.

The updated template looks like this:

```
@using Orchard.Layouts.ViewModels
@{
    var additionalProperties = new[] {
        new LayoutEditorPropertiesItem() {
            Label = T("Background Size:").ToString(),
            Model = "element.backgroundSize"
        }
    };
}
<div class="layout-element-wrapper layout-element-background" ng-class="{ 'layout-
container-empty': getShowChildrenPlaceholder()}">
    <ul class="layout-panel layout-panel-main">
        <li class="layout-panel-item layout-panel-label">Tile</li>
        <li class="layout-panel-item layout-panel-action layout-panel-action-edit"
ng-show="{element.hasEditor}" title="Edit tile (Enter)" ng-click="edit()"><i
class="fa fa-code"></i></li>
```

```

        @Display(New.LayoutEditor_Template_Properties(ElementTypeName: "tile",
Items: additionalProperties))
        <li class="layout-panel-item layout-panel-action" title="@T("Delete tile
(Del)")" ng-click="delete(element)" ng-show="element.canDelete()"><i class="fa fa-
remove"></i></li>
        <li class="layout-panel-item layout-panel-action" title="@T("Move tile up
(Ctrl+Up)")" ng-click="element.moveUp()" ng-show="element.canMoveUp()"><i
class="fa fa-chevron-up"></i></li>
        <li class="layout-panel-item layout-panel-action" title="@T("Move tile
down (Ctrl+Down)")" ng-click="element.moveDown()" ng-
show="element.canMoveDown()"><i class="fa fa-chevron-down"></i></li>
    </ul>
    <div class="layout-container-children-placeholder"
style="{{element.getTemplateStyles()}}">
        @T("Drag an element from the toolbox and drop it here to add content.")
    </div>
    @Display(New.LayoutEditor_Template_Children())
</div>

```

Before trying out the changes, make sure to execute the Build task from the Task Runner Explorer to update the generated files.

Notice that as you change the value for the Background Size property, the change is reflected instantly. Also notice that when you launch the element editor dialog, the same background size value is used. When you change that value from the editor and hit **Save**, that value is once again round-tripped back to the layout editor.

One final touch we could make is to provide some CSS that gives the child elements a semi-transparent background to increase the contrast when working with the layout editor and background images.

Open the file “Assets/Elements/Tile/Style.less” and enter the following CSS rules:

```

.layout-element-background {
    .layout-content-markup {
        background: rgba(255, 255, 255, 0.7);
    }
}

```

This gives all child elements a semi-transparent (70% opaque) white background. Make sure to execute the Build task from the Task Runner Explorer again, and then refresh the page:

Summary

Well, that was quite a journey. In this chapter, we learned all of the intricacies of writing custom container elements, and I'll admit it: it would be nice if this process were a little bit more straightforward.

Implementing custom container elements require the following steps:

1. Derive from ``Container``;
2. Create a driver;
3. Derive from ``LayoutModelMapBase<T>``;
4. Implement a client-side model and directive;
5. Implement a resource manifest provider;
6. Implement a shape table provider.

This developer's story is bound to be improved in newer versions of Orchard, but this is what it is for the time being. And I hope you agree that it is quite worth the effort.

14. Element Harvesters

In this chapter, we will talk about element harvesters, what they are and how we can write our own.

Element Harvesters are, simply put, providers of elements. They are the ones that make available all the elements we can choose from the layout editor's toolbox. The reason they are called "harvesters" is because of how they provide elements: they collect elements from a variety of sources.

As we have seen in the previous chapter, we can add custom elements to the system by writing classes that derive from **Element**. The Layouts module, however, doesn't discover these types directly based on the mere existence of those types. Instead, it relies on element harvesters to provide *element descriptors*. The Layouts module comes with a few element harvesters out of the box, one of them being the **TypedElementHarvester**. That class is the one responsible for providing element descriptors based on the classes derived from **Element**.

This decoupling of Element types and Element Descriptors offers great flexibility and is why Orchard is able to provide all sorts of elements dynamically. For example, the **Part** and **Field** elements are provided by the **ContentPartElementHarvester** and **ContentFieldHarvester**, respectively. There are no actual Element classes for each content part in the system. Instead, these harvesters yield element descriptors dynamically based on the existence of content part and field definitions.

Element Descriptors

An element descriptor, as its name implies, is an object that describes an element. It contains information such as the concrete .NET type to use when creating element instances, the technical name, description and display text. It also contains delegates to methods that are responsible for displaying element instances and element event handlers.

Element descriptors are provided by element harvesters. When you look at the Layout Editor Toolbox, the elements you see there represent the available element descriptors.

Element Harvesters Out of the Box

The following table lists all of the element harvesters that ship with Orchard, along with a description of what source is used to yield element descriptors:

Type	Description
BlueprintElementHarvester	Provides elements based on blueprint (pre-configured) elements.
ContentFieldElementHarvester	Provides elements based on the available content field definitions.

ContentPartElementHarvester	Provides elements based on the available content part definitions.
PlaceableContentElementHarvester	Provides elements based on the content types whose Placeable setting is set to true .
SnippetElementHarvester	Provides elements based on the existence of Razor views whose name end in “Snippet.cshtml”.
TypedElementHarvester	Provides elements based on the existence of .NET types that inherit from the Element type.
WidgetElementHarvester	Provides elements based on the existence of Widget content types (content types whose Stereotype is “Widget”).

With all of the available element harvesters, you probably won't need to implement your own anytime soon. However, there may be advanced scenarios where it makes perfect sense to dynamically provide your own elements. For example, you could be writing a module that integrates with a third-party CRM that provides customer related fields as defined by that

system, and you want to be able to add these fields as elements to content pages. Another example could be a custom harvester that yields elements based on the existence of content items of a certain type.

Whatever the case may be, knowing how to write custom element harvesters may come in handy. So in the next two sections, we'll see how to implement our own.

IElementHarvester

All element harvesters implement the **IElementHarvester** interface, which lives in the **Orchard.Layouts.Framework.Harvesters** namespace.

```
public interface IElementHarvester : ISingletonDependency {  
    IEnumerable<ElementDescriptor> HarvestElements(HarvestElementsContext  
context);  
}
```

The interface contains a single member that accepts a **HarvestElementsContext** argument and is expected to return a collection of **ElementDescriptor** objects.

Notice that the interface is derived from **ISingletonDependency**, which will cause any implementation to be registered with the Singleton lifetime scope with the IoC container. This means that you need to be aware of the lifetime scopes of other dependencies you may inject in your implementations.

For example, if you wanted to use the **IContentManager** service in your element harvester class, which has a *per-request* lifetime scope, you will have to inject that service using the **Work<T>** wrapper so that instances of **IContentManager** implementations are resolved in the current

request context, as opposed to the application lifetime scope. Failing to do so in this example will cause connection disposed errors, since the content manager holds a reference to a database connection, which will be disposed of at the end of each request.

The **HarvestElementsContext** object has only one property:

```
public class HarvestElementsContext {  
    public IContent Content { get; set; }  
}
```

The **Content** property is provided whenever the harvesters are invoked from a class or controller class that is somehow related to a content item. For example, in the case of the **LayoutEditorPartDriver**, which is responsible for creating the Layout Editor shape (which includes the elements toolbox), it passes along the content item associated with the **LayoutPart**. It is up to the individual harvesters if they need to do something with this information.

There are also cases where element harvesters are invoked outside the context of a specific content item. For example, when you create a blueprint element and are presented with a list of available base elements, there is no content item available to be used as context, so the **Content** property will be null. Therefore it's important to always check for null if your harvester can potentially do something with the **Content** property.

Two examples that take the **Content** property into account are the **ContentPartElementHarvester** and **ContentFieldElementHarvester**. These harvesters yield part and field element descriptors based on the parts and fields attached to the current content item for which the layout editor is being displayed.

Trying it out: Writing a Custom Element Harvester

In this tutorial we will write a simple element harvester that demonstrates how to dynamically provide elements to the system.

For demo purposes, we will implement a custom harvester that selects all fields from a custom content part called User Profile Part, and present those fields as elements available from the toolbox. The user can then create pages using these elements.

Let's see how that works.

Step 1: The UserProfilePart

The first thing to do is to define the **UserProfilePart** with bunch of content fields. You can do this either via the dashboard or via a migration as shown below:

```
using Orchard.ContentManagement.Metadata;
using Orchard.Core.Contents.Extensions;
using Orchard.Data.Migration;

namespace OffTheGrid.Demos.Layouts.Migrations {
    public class UserProfileMigrations : DataMigrationImpl {
        public int Create() {

            // Define the UserProfilePart.
            ContentDefinitionManager.AlterPartDefinition("UserProfilePart", part
=> part
                .WithField("FirstName", f => f
                    .OfType("TextField")
                    .WithSetting("TextFieldSettings.Flavor", "Wide")
                    .WithDisplayName("First Name"))
                .WithField("LastName", f => f
                    .OfType("TextField")
                    .WithSetting("TextFieldSettings.Flavor", "Wide")
                    .WithDisplayName("Last Name"))
                .WithField("TwitterHandle", f => f
```

```

        .OfType("TextField")
        .WithSetting("TextFieldSettings.Flavor", "Wide")
        .WithDisplayName("Twitter Handle"))
        .WithDescription("Provides additional information about the
user.")
        .Attachable());

    // Attach the UserProfilePart to the User content type.
    ContentDefinitionManager.AlterTypeDefinition("User", type => type
        .WithPart("UserProfilePart"));

    return 1;
}
}
}

```

Step 2: The Element Harvester

Next, create a new class that implements **IElementHarvester** as follows:

```

using System.Collections.Generic;
using Orchard.Layouts.Framework.Elements;
using Orchard.Layouts.Framework.Harvesters;
namespace OffTheGrid.Demos.Layouts.Layouts.Harvesters {
    public class UserProfileElementHarvester : IElementHarvester {
        public IEnumerable<ElementDescriptor>
HarvestElements(HarvestElementsContext context) {
            // TODO: Return element descriptors based on the content fields
            attached to the UserProfilePart.
        }
    }
}

```

To complete the above implementation, this is what we need to do:

1. Get the **UserProfilePart** content part definition and query the attached content fields.
2. For each content field that is attached to the User Profile Part, instantiate a new **ElementDescriptor** and configure it with appropriate data and handlers that implement the element's functionality. The **ElementDescriptor** constructor requires a **Type** based on **Element**, so we will need to implement one. To prevent this element from being harvested by the

TypedElementHarvester, we'll need to override the **IsSystemElement** property.

The following is the complete code for the harvester:

```
using System;
using System.Collections.Generic;
using System.Linq;
using OffTheGrid.Demos.Layouts.Elements;
using Orchard;
using Orchard.ContentManagement.Metadata;
using Orchard.Environment;
using Orchard.Layouts.Framework.Display;
using Orchard.Layouts.Framework.Elements;
using Orchard.Layouts.Framework.Harvesters;
using Orchard.Layouts.Services;

namespace OffTheGrid.Demos.Layouts
{
    public class UserProfileElementHarvester : Component, IElementHarvester
    {
        private readonly Work<IContentDefinitionManager>
        _contentDefinitionManager;
        private readonly Work<IContentFieldDisplay> _contentFieldDisplay;
        private readonly IWorkContextAccessor _workContextAccesor;

        public UserProfileElementHarvester(
            Work<IContentDefinitionManager> contentDefinitionManager,
            Work<IContentFieldDisplay> contentFieldDisplay,
            IWorkContextAccessor workContextAccesor) {

            _contentDefinitionManager = contentDefinitionManager;
            _workContextAccesor = workContextAccesor;
            _contentFieldDisplay = contentFieldDisplay;
        }

        public IEnumerable<ElementDescriptor>
        HarvestElements(HarvestElementsContext context)
        {
            // Get the UserProfilePart definition.
            var partDefinition =
                _contentDefinitionManager
                .Value
                .GetPartDefinition("UserProfilePart");

            // Get the content fields from the UserProfilePart definition.
            var fieldDefinitions = partDefinition.Fields;

            // For each field, yield an element descriptor.
            return from field in fieldDefinitions
                   let settingKeys = field.Settings.Keys
```

```

        let descriptionKey = settingKeys.FirstOrDefault(x =>
x.IndexOf("description", StringComparison.OrdinalIgnoreCase) >= 0)
        let description = descriptionKey != null ?
field.Settings[descriptionKey] : $"The {field.DisplayName} field."
        select new ElementDescriptor(
            elementType: typeof(UserProfileField),
            typeName: $"UserProfile.{field.Name}",
            displayText: T(field.DisplayName),
            description: T(description),
            category: "User"
        )
    {
        ToolboxIcon = "\uf040",
        Displaying = displayingContext =>
            OnDisplaying(field.Name, displayingContext)
    };
}

private void OnDisplaying(
    string fieldName,
    ElementDisplayingContext context)
{
    var workContext = _workContextAccesor.GetContext();
    var currentUser = workContext.CurrentUser;
    var profilePart = currentUser?.ContentItem.Parts.SingleOrDefault(x =>
x.PartDefinition.Name == "UserProfilePart");
    var field = profilePart?.Fields.SingleOrDefault(x => x.Name ==
fieldName);

    if (field == null)
    {
        // The field is no longer part the UserProfilePart.
        // This situation can occur when a user removed the field
        // and the harvested element descriptors cache entry hasn't been
        // evicted yet.
        return;
    }

    // Render the field and add it to the element shape.
    var fieldShape = _contentFieldDisplay.Value.BuildDisplay(currentUser,
field, context.DisplayType);
    context.ElementShape.ContentField = fieldShape;
}
}
}

```

Let's go over the above code section by section, starting with the **HarvestElements**:

```
// Get the UserProfilePart definition.
var partDefinition =
    _contentDefinitionManager
        .Value
        .GetPartDefinition("UserProfilePart");

// Get the content fields from the UserProfilePart definition.
var fieldDefinitions = partDefinition.Fields;
```

Here we simply use the Content Definition Manager to get the User Profile Part definition so that we can get a collection of its attached fields, which we use to generate element descriptors, as seen next:

```
// For each field, yield an element descriptor.
return from field in fieldDefinitions
    let settingKeys = field.Settings.Keys
    let descriptionKey = settingKeys.FirstOrDefault(x =>
x.IndexOf("description", StringComparison.OrdinalIgnoreCase) >= 0)
    let description = descriptionKey != null ? field.Settings[descriptionKey] :
$"The {field.DisplayName} field."
    select new ElementDescriptor(
        elementType: typeof(UserProfileField),
        typeName: $"UserProfile.{field.Name}",
        displayText: T(field.DisplayName),
        description: T(description),
        category: "User"
    )
    {
        ToolboxIcon = "\uf040",
        Displaying = displayingContext => OnDisplaying(field.Name,
displayingContext)
    };
};
```

The above code projects the field definitions into **ElementDescriptors**. Notice the assignment of the **Displaying** property of the descriptors being created, which is a delegate type. Whenever a field element is being displayed, this delegate is invoked, which provides you with the opportunity to configure the element shape that will be displayed. In the case of this example, we need to actually render the content field, which looks like this:

```
private void OnDisplaying(string fieldName, ElementDisplayingContext context)
{
    var workContext = _workContextAccesor.GetContext();
    var currentUser = workContext.CurrentUser;
```

```

var profilePart = currentUser?.ContentItem.Parts.SingleOrDefault(x =>
x.PartDefinition.Name == "UserProfilePart");
var field = profilePart?.Fields.SingleOrDefault(x => x.Name == fieldName);

if (field == null)
{
    // The field is no longer part the UserProfilePart.
    // This situation can occur when a user removed the field
    // and the harvested element descriptors cache entry hasn't been
    // evicted yet.
    return;
}

// Render the field and add it to the element shape.
var fieldShape = _contentFieldDisplay.Value.BuildDisplay(currentUser, field,
context.DisplayType);
context.ElementShape.ContentField = fieldShape;
}

```

The above code basically gets a reference to the authenticated user and the specified content field of the **UserProfilePart**.

The harvester references a type called **UserProfileField**, which you'll need to create as well:

```

using Orchard.Layouts.Framework.Elements;
namespace OffTheGrid.Demos.Layouts.Elements
{
    public class UserProfileField : Element
    {
        public override string Category => "Demo";
        public override bool IsSystemElement => true;
    }
}

```

This class will be used to instantiate user profile field elements. The **IsSystemElement** is overridden to return **true**, which prevents the **TypedElementHarvester** from harvesting this element class as an element. This is important, otherwise the user would be able to add elements of type **UserProfileField** to the canvas, which would be rather useless.

Next, we need to create a shape template for the **UserProfileField** element so that we can actually render the content field. As mentioned

earlier, an alternate is added to the Element shape based on the .NET type name of the element class. Since the harvester uses the **UserProfileField** type as the element type, there will be an alternate available called **“Elements.UserProfileField”**.

Let's go ahead and create a Razor view called **“UserProfileField.cshtml”** in the **Views/Elements** folder of the custom module and provide the following code:

```
@if(Model.ContentField != null) {  
    @Display(Model.ContentField)  
}  
else {  
    <!-- The content field is not available. -->  
}
```

All this code does is render the ContentField shape created by the harvester, if one was set.

And with that in place, you should now be able to attach content fields to the **UserProfilePart** and add those fields as elements to the canvas.



Figure 14-1 - Adding dynamically provided elements to the canvas

Improvements

When you use the profile field elements as-is, they render their content field using the default template for that field. Although we could use one of the existing alternates based on content field name or content type, the downside is that these changes will affect the way the fields are rendered everywhere. Therefore, let's add our own alternate based on our element's type name and the content field type and name in question. This way, we can customize the rendering of content fields just for the user profile field elements without affecting the default rendering of these fields.

The alternates we will add will be based on the element type name (**UserProfileField**), the content field type name (for example “**TextField**”) and the content field name (for example “**FirstName**”). With these alternates in place we can then create common templates for **UserProfileField** elements in general, or provide more specific templates for elements rendering a specific content field type, and even customize individual content fields.

To add these alternates, we first need to get our hands onto the shape (or shapes) created by the content field driver (or drivers). Because the thing is, **IContentFieldDisplay** does not actually return the field shapes directly. Instead, it returns a shape of type **ContentField** which has a **Content** property of type **ZoneHolding**, which in turn holds the shapes returned by the content field drivers.

Since content field drivers can return any number of shapes of any given type, we will iterate over each content field shape and add the following alternates to them.

- An alternate based on the shape type name and the term **“UserProfile”** to indicate that it is create by the element harvester.
- Another alternate that is the same as the first one, but appended with the technical name of the content field.

What follows next is an updated version of the **OnDisplaying** method that will add the alternates:

```
private void OnDisplaying(string fieldName, ElementDisplayingContext context)
{
    var workContext = _workContextAccesor.GetContext();
    var currentUser = workContext.CurrentUser;
    var profilePart = currentUser?.ContentItem.Parts.SingleOrDefault(x =>
x.PartDefinition.Name == "UserProfilePart");
    var field = profilePart?.Fields.SingleOrDefault(x => x.Name == fieldName);

    if (field == null)
    {
        // The field is no longer part the UserProfilePart.
        // This situation can occur when a user removed the field
        // and the harvested element descriptors cache entry hasn't been
        // evicted yet.
        return;
    }

    // Render the field.
    var contentFieldShapeHolder =
_contentFieldDisplay.Value.BuildDisplay(currentUser, field, context.DisplayType);

    // The returned shape is a ContentField shape that has a Content property of
type ZoneHolding,
    // which in turn contains a single shape which is the field shape we're
interested in.
    var fieldShapes =
((IEnumerable<dynamic>)contentFieldShapeHolder.Content.Items);

    // Add alternates to each content field shape.
    foreach (var fieldShape in fieldShapes)
    {
        fieldShape.Metadata.Alternates.Add($"{fieldShape.Metadata.Type}__UserProfile");

        fieldShape.Metadata.Alternates.Add($"{fieldShape.Metadata.Type}__UserProfile__{fie
ld.Name}");
    }

    // Assign the field shape to a new property of the element shape.
```

```
context.ElementShape.ContentField = contentFieldShapeHolder;  
}
```

Now let's create two Razor templates that customize the Text Field shapes:

1. One text field template to get rid of the content field name.
2. Another text field template that is specific to the **TwitterHandle** field so that we can render it as a hyperlink.

Create the first Razor view with the following contents:

Views/Fields/Common.Text-UserProfile.cshtml:

```
@if (HasText(Model.Value)) {  
    <p>@Model.Value</p>  
}
```

And the second Razor view specific to the twitter handle field:

Views/Fields/Common.Text-UserProfile-TwitterHandle.cshtml:

```
@if (HasText(Model.Value)) {  
    <a href="@String.Format("https://twitter.com/{0}",  
Model.Value)">@Model.Value</a>  
}
```

When you now add the user profile fields to a page and check it out on the front end, you'll see something like figure 14.2.

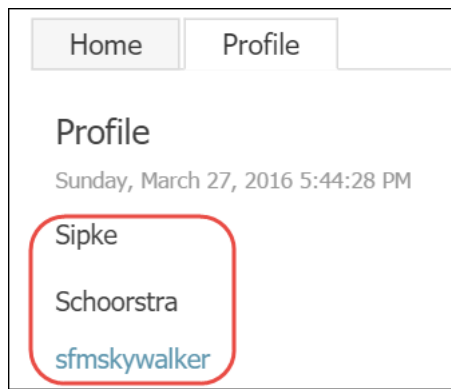


Figure 14-2 - The user profile fields as displayed on the front end for the currently authenticated user.

Summary

In this chapter, we learned about element harvesters, which are providers of element types called *element descriptors*.

Orchard comes with a few element harvesters of its own, the most important one being the **TypedElementHarvester**, which populates the toolbox with elements based on the mere existence of classes that derive from ``Element``.

We then saw how easy it is to implement a custom harvester through a tutorial.

Being able to write custom harvesters has the potential to unlock many more advanced scenarios for your web application built on Orchard.

15. Extending Existing Elements

In the previous chapters, we learned how to create new types of elements by creating new element classes and harvesters.

In this chapter, instead of creating new element types, we will see how we can extend existing ones by creating *additional element drivers* and *element handlers*.

By creating drivers and handlers, we can store additional information for an element, provide additional UI to manage that information, and provide additional behavior.

Extending elements with additional information does not work the same as with content types. The content type system is one of *composition*. Elements on the other hand are *atomic*, which means one does not simply stick additional properties on them. Which of course begs the question: how can we add additional information to element types that we haven't written ourselves?

As it turns out, the answer to that question is quite simple. Elements have a public data dictionary which we can access to read and write additional information. This dictionary is exposed from the base Element class via a property called **Data** and is of type **IDictionary<string, string>**.

Using Multiple Drivers

Another way that elements can be extended are by creating additional element drivers, which allows you to provide additional UI on

the element editor dialog screen. You can write drivers for a specific element type, or for a base class that is shared by multiple element types.

Using Multiple Handlers

In addition to writing additional drivers, you can also implement additional *element handlers*. In fact, the very element driver system itself is implemented by such an element handler. This means that we can extend very specific element types as well as any other element types based on whatever criteria we like.

To create a custom element handler, all you need to do is create a class that either implements the **IElementEventHandler** interface, or derives from the **ElementEventHandlerBase** class.

Let's see how this all works.

Trying It Out: Extending Elements

If you know how to write an element handler or element driver, you already know how to extend existing elements, since the process is the same. The primary difference is that we don't "own" the element, and that we can't implement strongly-typed properties on them that access their data dictionary.

To see how it all works, let's implement a common driver for any element type and provide an editor UI for a custom property called **FadeIn**. This will demonstrate how to add additional properties to existing elements. This property will be a simple Boolean value, and when set to

true, will append a certain CSS class that will cause the element to fade in on page load.

Creating the CommonElementDriver

The goal of the **CommonElementDriver** is to provide an additional editor for *all* elements, so we'll create a driver for the base **Element** class.

The driver will implement the **OnBuildEditor** method to display and handle post backs. For that we'll create a view model to pass in and receive back the **FadeIn** value. We'll also assign the editor UI to a tab named **“Visibility”**.

To simplify working with the “FadeIn” property from code, we'll create two extension methods on the **Element** class so we don't have to deal with converting to and from strings all the time. The extension methods look like this:

```
using System;
using Orchard.ContentManagement;
using Orchard.Layouts.Framework.Elements;
using Orchard.Layouts.Helpers;

namespace OffTheGrid.Demos.Layouts.Helpers {
    public static class ElementExtensions {
        private const string FadeInKey = "FadeIn";
        private const string DefaultFadeInValue = "false";

        public static bool GetFadeIn(this Element element) {
            return XmlHelper.Parse<bool>(element.Data.Get(FadeInKey,
DefaultFadeInValue));
        }

        public static void SetFadeIn(this Element element, bool value) {
            element.Data[FadeInKey] = XmlHelper.ToString(value);
        }
    }
}
```


As you can see, the extension methods work with the element's data dictionary directly and set and get the value stored by the “**FadeIn**” key.

The view model is defined as follows:

```
using System;

namespace OffTheGrid.Demos.Layouts.ViewModels {
    public class ElementViewModel {
        public bool FadeIn { get; set; }
    }
}
```

And with that in place, we can implement the driver as follows:

```
using OffTheGrid.Demos.Layouts.Helpers;
using OffTheGrid.Demos.Layouts.ViewModels;
using Orchard.Layouts.Framework.Drivers;
using Orchard.Layouts.Framework.Elements;

namespace OffTheGrid.Demos.Layouts.Elements {
    public class CommonElementDriver : ElementDriver<Element> {

        protected override EditorResult OnBuildEditor(Element element,
            ElementEditorContext context) {
            // Initialize the view model with existing data.
            var viewModel = new ElementViewModel {
                FadeIn = element.GetFadeIn()
            };

            // Model bind the view model if an Updater is provided.
            if(context.Updater != null) {
                if (context.Updater.TryUpdateModel(viewModel, context.Prefix,
                    null, null)) {
                    element.SetFadeIn(viewModel.FadeIn);
                }
            }

            // Create the editor template shape.
            var visibilityEditorTemplate = context.ShapeFactory.EditorTemplate(
                TemplateName: "Elements/Common.Visibility",
                Model: viewModel,
                Prefix: context.Prefix);

            // Specify the position of the editor shapes.
            // This is key to assigning editor templates to a tab.
            visibilityEditorTemplate.Metadata.Position = "Visibility:1";

            // Return the editor shape.
            return Editor(context, visibilityEditorTemplate);
        }
    }
}
```

```
}
    }
}
```

Now all that's left is actually implementing the editor template's view and implement the **FadeIn** property so that it will add a certain CSS class and a stylesheet that defines the rule for that CSS class. This CSS class will achieve the fade in effect on the element when the page is loaded.

Views/EditorTemplates/Elements/Common.Visibility.cshtml:

```
@model OffTheGrid.Demos.Layouts.ViewModels.ElementViewModel
<fieldset>
    <div class="form-group">
        @Html.CheckBoxFor(m => m.FadeIn)
        @Html.LabelFor(m => m.FadeIn, T("Fade In").ToString(), new { @class =
"forcheckbox" } )
        @Html.Hint(T("Causes the element to fade in on document ready (using CSS
animations)."))
    </div>
</fieldset>
```

The result should look like this:

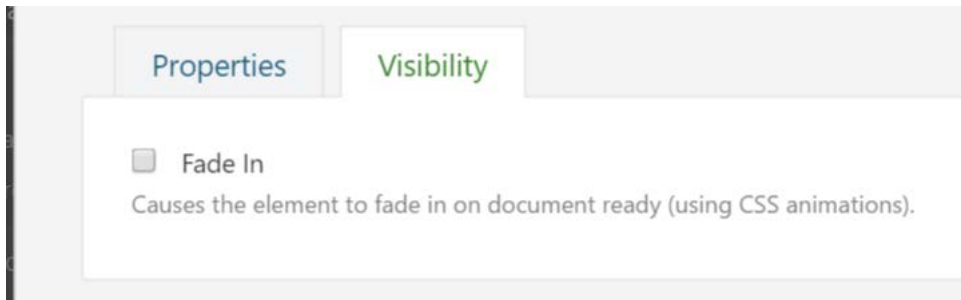


Figure 15-1 - All elements now have an additional tab called Visibility

Implementing the FadeIn Behavior

To make the **FadeIn** setting actually do something, we need to do the following:

1. Append a CSS class to the element's tag being rendered
2. Provide the CSS rule for that class
 - a. Which means we need a stylesheet
 - b. Get that stylesheet included on the page

Appending the CSS class is easy. All we need to do is implement the **OnDisplaying** method in the driver, and add a CSS class to the **Classes** list of the element shape:

```
protected override void OnDisplaying(Element element, ElementDisplayingContext context) {  
    if (context.DisplayType == "Design")  
        return;  
  
    if (!element.GetFadeIn())  
        return;  
  
    context.ElementShape.Classes.Add("auto-fade-in");  
}
```

Notice I added a check on the display type. Although this check is not strictly necessary, it is cleaner if we don't add the CSS class when the element is rendered in design mode.

Next, create the following LESS file:

Assets/Elements/Common/Style.less

And type in the following CSS:

```
.auto-fade-in {  
    -webkit-animation: fadein 2s; /* Safari, Chrome and Opera > 12.1 */  
    -moz-animation: fadein 2s; /* Firefox < 16 */  
    -ms-animation: fadein 2s; /* Internet Explorer */  
    -o-animation: fadein 2s; /* Opera < 12.1 */  
    animation: fadein 2s;  
}  
  
@keyframes fadein {  
    from { opacity: 0; }  
    to { opacity: 1; }  
}
```

```

/* Firefox < 16 */
@-moz-keyframes fadein {
    from { opacity: 0; }
    to   { opacity: 1; }
}

/* Safari, Chrome and Opera > 12.1 */
@-webkit-keyframes fadein {
    from { opacity: 0; }
    to   { opacity: 1; }
}

/* Internet Explorer */
@-ms-keyframes fadein {
    from { opacity: 0; }
    to   { opacity: 1; }
}

/* Opera < 12.1 */
@-o-keyframes fadein {
    from { opacity: 0; }
    to   { opacity: 1; }
}

```

Also add the following configuration to *Assets.json*:

```

{
    "inputs": [
        "Assets/Elements/Common/Style.less"
    ],
    "output": "Styles/CommonElement.css"
}

```

The tricky part is registering this stylesheet. Since this driver executes for all shapes being rendered, there's no single shape template that we can use to include our stylesheet, and we definitely don't want to override every element shape template out there.

So the best thing we can do is define a resource manifest for the stylesheet, and require it using the **IResourceManager**.

The resource manifest provider looks like this:

```
using Orchard.UI.Resources;

namespace OffTheGrid.Demos.Layouts.Handlers {
    public class CommonElementResourceManifest : IResourceManifestProvider {
        public void BuildManifests(ResourceManifestBuilder builder) {
            var manifest = builder.Add();
            manifest.DefineStyle("CommonElement").SetUrl("CommonElement.min.css",
"CommonElement.css");
        }
    }
}
```

Next, update the driver's **OnDisplaying** method by adding the following line:

```
_resourceManager.Require("stylesheet", "CommonElement");
```

The **_resourceManager** is a private field on the driver's class and injected as follows:

```
private readonly IResourceManager _resourceManager;

public CommonElementDriver(IResourceManager resourceManager) {
    _resourceManager = resourceManager;
}
```

With that in place, we can now have any element we like fading in. Except for the elements Canvas, Grid, Row and Column, since they are defined as not having an element editor. Hopefully we get more control over dynamically setting the **HasEditor** setting for those elements in the future, so we can extend their settings and behavior as well.

Go ahead and try out this new setting. When **FadeIn** is enabled on an element, that element should fade in when displayed on the front end.

An Alternative Implementation

We have just seen how to extend an element using an element driver. But what if you wanted to extend only elements that meet criteria other

than being of a certain type? For example, what if we only wanted to expose the `FadeIn` property for the following elements: **Html** and **Image**?

One approach could be to write two specific drivers of course, and that would be fine in this scenario. But a better approach which is much more flexible would be to implement an element handler instead.

The following code demonstrates exactly that.

```
using System;
using System.Collections.Generic;
using OffTheGrid.Demos.Layouts.Helpers;
using OffTheGrid.Demos.Layouts.ViewModels;
using Orchard.Layouts.Elements;
using Orchard.Layouts.Framework.Display;
using Orchard.Layouts.Framework.Drivers;
using Orchard.Layouts.Services;
using Orchard.UI.Resources;

namespace OffTheGrid.Demos.Layouts.Elements {
    public class CommonElementHandler : ElementEventHandlerBase {
        private readonly IResourceManager _resourceManager;

        public CommonElementHandler(IResourceManager resourceManager) {
            _resourceManager = resourceManager;
        }

        private IList<Type> SupportedTypes = new List<Type> {
            { typeof(Html) },
            { typeof(Image) },
        };

        public override void BuildEditor(ElementEditorContext context) {
            var element = context.Element;
            var elementType = element.GetType();

            if (!SupportedTypes.Contains(elementType))
                return;

            // Initialize the view model with existing data.
            var viewModel = new ElementViewModel {
                FadeIn = element.GetFadeIn()
            };

            // Model bind the view model if an Updater is provided.
            if (context.Updater != null) {
                if (context.Updater.TryUpdateModel(viewModel, context.Prefix,
                    null, null)) {
                    element.SetFadeIn(viewModel.FadeIn);
                }
            }
        }
    }
}
```

```

    }
}

// Create the editor template shape.
var visibilityEditorTemplate = context.ShapeFactory.EditorTemplate(
    TemplateName: "Elements/Common.Visibility",
    Model: viewModel,
    Prefix: context.Prefix);

// Specify the position of the editor shapes.
// This is key to assigning editor templates to a tab.
visibilityEditorTemplate.Metadata.Position = "Visibility:1";

// Add the editor shape.
context.EditorResult.Add(visibilityEditorTemplate);
}

public override void Displaying(ElementDisplayingContext context) {
    if (context.DisplayType == "Design")
        return;

    var element = context.Element;
    var elementType = element.GetType();

    if (!SupportedTypes.Contains(elementType))
        return;

    if (!element.GetFadeIn())
        return;

    context.ElementShape.Classes.Add("auto-fade-in");
    _resourceManager.Require("stylesheet", "CommonElement");
}
}
}

```

Notice that the element handler is almost the same as the driver implementation, with a few noticeable differences:

- The handler inherits from **ElementEventHandlerBase** instead of **ElementDriver<T>**
- The handler gets a reference to the element being displayed from the **context** argument, since it's not provided as a separate argument like the driver implementation
- The handler implements specific logic to when the FadeIn editor and behavior is applied. The handler declares a

SupportedTypes collection property as a white list. Only elements within that white list can be configured to fade in.

Everything else remains the same: no need to change anything in the views.

Summary

So there you have it. This chapter explained how one can extend existing elements with additional properties and behavior. You can do this at a very granular level or on a global level, or anything in between, by implementing element drivers and element handlers.

16. Layout and Element APIs

So far we've seen how to write custom elements by creating custom element classes and harvesters. In this chapter, we will learn about the APIs at our disposal to programmatically work with elements. More specifically, we will learn how to:

- Work with the element manager to query all available element categories and descriptors
- Use the element factory to instantiate elements
- Render elements and layouts of elements with the layout manager and element manager
- Serialize and deserialize elements
- Handle element events by implementing the **IElementEventHandler** interface
- Reuse the layout editor from your own module

Knowing about these APIs will enable you to take advantage of all that the Layouts module has to offer and implement all sorts of applications yourself. For example, in the next part, we will write a **SlideShowPart** that uses the APIs from this chapter to enable the user to create slides that are based on layouts. To achieve such functionality, it is important to have a decent understanding of how the various APIs work.

We'll get started with an overview of the most interesting APIs first, and then move forward with examples and see how they work.

Managing Layouts and Elements

The Layouts module comes with two services that you can use to programmatically manage layouts and elements:

- Layout Manager
- Element Manager

We'll discuss these next.

The Layout Manager

The Layout Manager is provided via the **ILayoutManager** interface, and relies on the Element Manager (discussed next) to manage elements on a somewhat higher level.

For example, it has a method that accepts a serialized string of elements which it can render into a complete tree of element shapes. It also provides methods to manage Layout content items and apply layout templates.

The following table provides a description for each member of the **ILayoutManager** interface:

Member	Description
GetTemplates	Returns a list of all content items with a LayoutPart attached whose

	<p>IsTemplate content type-part setting is set to true. This method is used to populate the Templates dropdown list on the layout editor from which the user can choose a layout template to apply to the current layout.</p>
GetLayouts	<p>Returns a list of all content items with a LayoutPart attached (regardless of the IsTemplate setting).</p>
GetLayout	<p>Returns a content item with a LayoutPart attached with the specified content item ID.</p>
LoadElements	<p>Deserializes the LayoutData string value of the specified ILayoutAspect (which is typically a content item with a LayoutPart which itself implements this interface) and returns a list of elements.</p>
RenderLayout	<p>Deserializes the specified data string into a list of elements</p>

	<p>(internally invoking <code>LoadElements</code>) and then invokes <code>IElementDisplay.DisplayElements</code> internally, which returns a hierarchy of element shapes. In addition, this method accepts two optional arguments: <code>displayType</code> and <code>content</code>. The display type is used to add shape alternates. The content argument should be specified if there is any.</p>
<code>ApplyTemplate</code>	<p>Used by the layout editor to apply the elements of the selected layout template to the elements of the current layout elements, essentially combining the two sets of elements and returning the resulting set.</p>
<code>DetachTemplate</code>	<p>The inverse of <code>ApplyTemplate</code>, stripping out all sealed (templated) elements of the specified list and returning the result of that.</p>
<code>GetTemplateClients</code>	<p>Returns a list of all content items that use the specified layout template. This is used by the layout content handler to re-apply an</p>

	updated layout template to all content items using the updated layout template.
CreateDefaultLayout	Returns a list of elements that represents a default layout which consists of a root Canvas, a Grid, Row and Column.
Exporting	Invokes the Exporting method on the element manager service for each element in the specified list of elements (as provided via the context argument).
Exported	Invokes the Exported method on the element manager service for each element in the specified list of elements (as provided via the context argument).
Importing	Invokes the Importing method on the element manager service for each element in the specified list of elements (as provided via the context argument).

Imported	Invokes the Imported method on the element manager service for each element in the specified list of elements (as provided via the context argument).
ImportCompleted	Invokes the ImportCompleted method on the element manager service for each element in the specified list of elements (as provided via the context argument).

The Element Manager

The Element Manager is a service that deals with elements, such as loading and rendering them.

The element manager is provided in the form of the **IElementManager** interface, and has the following methods:

Member	Description
DescribeElements	Returns a list of all available element descriptors by querying all

	available element harvesters. This method is used by the Layout Editor to provide the elements you see in the toolbox, as well as by the blueprint element controller
GetCategories	Returns a list of all the element categories.
GetElementDescriptorByTypeName	Returns a single element descriptor with the specified type name.
GetElementDescriptorByType	Returns a single element descriptor with the type name based on the specified generic argument type.
ActivateElement	Instantiates an element based on the given element descriptor. It internally relies on the IElementFactory to instantiate elements.
GetDrivers	Returns a list of all element drivers for the specified element descriptor or type, if specified. If no descriptor

	or type is specified, all element drivers are returned.
BuildEditor	Invokes the BuildEditor method on all IElementEventHandler implementations.
UpdateEditor	Invokes the UpdateEditor method on all IElementEventHandler implementations.
Saving	Invokes the LayoutSaving method on all IElementEventHandler implementations.
Removing	Invokes the Removing method on all IElementEventHandler implementations.
Exporting	Invokes the Exporting method on all IElementEventHandler implementations.

Exported	Invokes the Exported method on all IElementEventHandler implementations.
Importing	Invokes the Importing method on all IElementEventHandler implementations.
Imported	Invokes the Imported method on all IElementEventHandler implementations.
ImportCompleted	Invokes the ImportCompleted method on all IElementEventHandler implementations.

Element Events

Part of writing custom elements consists of implementing element drivers, which allow you to handle events specific to the element type of that driver. Alternatively, we can implement event handlers to handle events for any type of element. To do so, you need to implement the **IElementEventHandler** interface, which looks like this:

IElementEventHandler

```
interface IElementEventHandler : IEventHandler
{
    void Creating(ElementCreatingContext context);
    void Created(ElementCreatedContext context);
    void CreatingDisplay(ElementCreatingDisplayShapeContext context);
    void Displaying(ElementDisplayingContext context);
    void Displayed(ElementDisplayedContext context);
    void BuildEditor(ElementEditorContext context);
    void UpdateEditor(ElementEditorContext context);
    void LayoutSaving(ElementSavingContext context);
    void Removing(ElementRemovingContext context);
    void Exporting(ExportElementContext context);
    void Exported(ExportElementContext context);
    void Importing(ImportElementContext context);
    void Imported(ImportElementContext context);
    void ImportCompleted(ImportElementContext context);
}
```

The majority of these methods are invoked either by the Element Manager or the Element Display service.

Oftentimes, when implementing an event handler, you're typically only interested in one or some methods to implement. So instead of implementing this interface directly, you can implement the abstract class **ElementEventHandlerBase** instead. That way you just need to override the method you're interested in handling.

Displaying Elements

The service that is responsible for creating shapes from elements is the Element Display service, which is defined as follows.

IElementDisplay

```
interface IElementDisplay : IDependency
{
    dynamic DisplayElement(Element element, IContent content, string displayType = null, IUpdateModel updater = null);
    dynamic DisplayElements(IEnumerable<Element> elements, IContent content, string displayType = null, IUpdateModel updater = null);
}
```

The **DisplayElement** method returns a shape of type “**Element**”, while the **DisplayElements** method returns a shape of type “**LayoutRoot**”, whose child shapes are all of type “**Element**”.

The Layout Editor

The Layout Editor that is used by the Layout Part can be reused by your own modules. The Layout Editor is implemented as a simple view model class that is rendered using a Razor editor template view. To get your hands on a fully initialized layout editor view model, use the Layout Editor Factory service.

ILayoutEditorFactory

The Layout Editor Factory service enables you to instantiate a fully initialized **LayoutEditor** object, which is used as a view model. You use the standard MVC Html helpers **Editor** and **EditorFor** on **LayoutEditor** objects to display the entire layout editor. You typically use this layout editor factory class from your own controller, and then supply the **LayoutEditor** object to your view.

The following code shows the interface declaration:

```
public interface ILayoutEditorFactory : IDependency
{
    LayoutEditor Create(LayoutPart layoutPart);
    LayoutEditor Create(string layoutData, string sessionKey, int? templateId = null, IContent content = null);
}
```

```
}
```

As you see, there is a single method called `Create` with two overloads. The first overload takes a `LayoutPart` to read element information from, while the second overload allows you to provide this information yourself.

We'll see a fully functioning example of how to work with the layout editor in this chapter.

Serialization

The Layouts module comes with two services that handle serialization and deserialization of elements:

- Layout Serializer
- Element Serializer

ILayoutSerializer

The Layout Serializer service is provided via the **ILayoutSerializer** interface. This service can serialize an array of elements into a JSON string, and deserialize a JSON string back into an array of elements.

Internally, it relies on the Element Serializer service which contains the logic to serialize and deserialize individual elements and JSON nodes.

The following table describes each member of **ILayoutSerializer**:

Member	Description
Serialize	Serializes a hierarchy of elements into a JSON string.
Deserialize	Deserializes a JSON string into a hierarchy of elements.

IElementSerializer

The Element Serializer is provided via the IElementSerializer interface, and is responsible for converting individual elements from and to JSON.

The following table describes the members of **IElementSerializer**:

Member	Description
Serialize	Serializes an element into a JSON string. If the element has child elements, they too will be serialized recursively.
Deserialize	Deserializes a JSON string into a single element. If the element has

	child elements, they too will be deserialized recursively.
ToDto	This is used by the Serialize method to turn a given element into an anonymous object, which is then serialized into a JSON string. The reason this method is exposed on the interface is because ILayoutSerializer needs to use the same code.
ParseNode	This is the counterpart of ToDto and used by the Deserialize method to parse a given JSON node into an actual element instance. The reason this method is exposed on the interface is because ILayoutSerializer needs to use the same code.

Trying it out: Working with the APIs

In this tutorial, we'll see how to work with the various APIs covered in this chapter. We'll see how to manually construct a tree of elements,

initialize them, serialize and deserialize them and finally render them. We'll then see how to work with the layout editor.

An easy way to demonstrate the APIs is to work with them from a controller, so that's what we'll create first.

Creating the Controller

Create the following controller:

```
using System.Web.Mvc;

namespace OffTheGrid.Demos.Layouts.Controllers {
    public class ElementsApiController : Controller {
        public ActionResult Index() {
            return View();
        }
    }
}
```

Make sure to create an empty Razor View as well.

Creating Elements

With that in place, let's create an element of type **Html** and initialize it with some HTML content:

```
using System.Web.Mvc;
using Orchard.Layouts.Elements;
using Orchard.Layouts.Services;

namespace OffTheGrid.Demos.Layouts.Controllers {
    [Themed]
    public class ElementsApiController : Controller {
        private readonly IElementManager _elementManager;

        public ElementsApiController(IElementManager elementManager) {
            _elementManager = elementManager;
        }

        public ActionResult Index() {
```



```

        // Create a new instance of the Html element using the element
manager.
        var html = _elementManager.ActivateElement<Html>();

        // Configure the element.
        html.Content = "<p>This is an <strong>HTML</strong> element.</p>";
        html.HtmlClass = "demo-content";
        html.HtmlId = "Paragraph1";
        html.HtmlStyle = "color: hotpink;";

        return View();
    }
}

```

The above code listing demonstrates how to work with the **IElementManager** to instantiate an element of a given type.

Instead of setting up the instantiated HTML element as I did above, I could have provided an anonymous function instead, which looks like this:

```

// Create a new instance of the Html element using the element manager.
var html = _elementManager.ActivateElement<Html>(e => {
    // Configure the element.
    e.Content = "<p>This is an <strong>HTML</strong> element.</p>";
    e.HtmlClass = "demo-content";
    e.HtmlId = "Paragraph1";
    e.HtmlStyle = "color: hotpink;";
});

```

It's almost the same, but this time I used the element initializer callback instead. Another difference is that this code initializes the element before the **Created** event is triggered. Although this may be irrelevant in most cases, but it's good to keep in mind for certain advanced scenarios where you handle the Created event and require a fully initialized element.

When invoking **_elementManager.ActivateElement<Html>**, what happens under the covers is that the element manager uses the C# **typeof** keyword to get the .NET type to get the **FullName** property of the type, by which the element descriptors are registered. To get a descriptor by name directly, use the **GetElementDescriptorByTypeName** method. This is useful

in scenarios where an element is not registered by its .NET type name, but dynamically, such as element blue prints for example.

The following code demonstrates how we could instantiate the Html element by its element type name:

```
// Create a new instance of the Html element using the element manager.
var htmlDescriptor =
    _elementManager
        .GetElementDescriptorByTypeName(DescribeElementsContext.Empty,
            "Orchard.Layouts.Elements.Html");

// Need to cast to Html, since this overload does not know the .NET type of the
// element being activated.
var html = (Html)_elementManager.ActivateElement(htmlDescriptor);
```

Rendering Elements

Now that we have seen how to programmatically instantiate elements, it would be nice if we knew how to render them. You could of course simply send the element to the view, and render it from there. For example, the following would totally work:

```
public ActionResult Index() {

    // Create a new instance of the Html element using the element manager.
    var html = _elementManager.ActivateElement<Html>(e => {
        // Configure the element.
        e.Content = "<p>This is an <strong>HTML</strong> element.</p>";
        e.HtmlClass = "demo-content";
        e.HtmlId = "Paragraph1";
        e.HtmlStyle = "color: hotpink;";
    });

    // Assign the Html element to a property on the dynamic ViewBag.
    ViewBag.HtmlElement = html;
    return View();
}
```

And then in the “Index.cshtml” view:

```
@using Orchard.Layouts.Elements
@{
    // Access the Html element from the ViewBag.
    var htmlElement = (Html)ViewBag.HtmlElement;
}
<div id="@htmlElement.HtmlId" style="@htmlElement.HtmlStyle"
class="@htmlElement.HtmlClass">
    @Html.Raw(htmlElement.Content)
</div>
```

However, this will become quite cumbersome when you want to create entire trees of elements, because the view now requires hardcoded knowledge about the structure of the element tree, including the types of each element. Now, you could create Html helpers and partial views that render individual elements and their children recursively, and that would work. But you would be reinventing the wheel.

Let's see how it should be done instead by leveraging the Element Display service.

Change the controller code as follows:

```
public ActionResult Index() {

    // Create a new instance of the Html element using the element manager.
    var html = _elementManager.ActivateElement<Html>(e => {
        // Configure the element.
        e.Content = "<p>This is an <strong>HTML</strong> element.</p>";
        e.HtmlClass = "demo-content";
        e.HtmlId = "Paragraph1";
        e.HtmlStyle = "color: hotpink;";
    });

    // Render the Html element.
    var htmlShape = _elementDisplay.DisplayElement(html, content: null);

    // Assign the HtmlShape to a property on the dynamic ViewBag.
    ViewBag.HtmlShape = htmlShape;

    return View();
}
```

The above code shows the usage of the **IElementDisplay** service, which is injected via the constructor (not shown above). Invoking the

element display service is easy: pass the element instance to display and optionally a content item to serve as context to elements, and what you get back is an element shape that is ready for rendering, which is done from the view:

@Display(ViewBag.HtmlShape)

This also means that, by virtue of leveraging shapes, any alternates get applied to the shape templates.

Before we move on to serialization, let's see how to create a more advanced element hierarchy. For example, let's create a Grid with one Row and 2 Columns. We'll add one Html element to each column.

```
public ActionResult Index() {  
  
    // Create an hierarchy of elements.  
    var rootElement = New<Grid>(grid => {  
        // Row.  
        grid.Elements.Add(New<Row>(row => {  
            // Column 1.  
            row.Elements.Add(New<Column>(column => {  
                column.Width = 8;  
                column.Elements.Add(New<Html>(html => html.Content = "This is the  
<strong>first</strong> column."));  
            }));  
            // Column 2.  
            row.Elements.Add(New<Column>(column => {  
                column.Width = 4;  
                column.Elements.Add(New<Html>(html => html.Content = "This is the  
<strong>second</strong> column."));  
            }));  
        }));  
    });  
  
    // Render the Html element.  
    var gridShape = _elementDisplay.DisplayElement(rootElement, content: null);  
  
    // Assign the HtmlShape to a property on the dynamic ViewBag.  
    ViewBag.RootElementShape = gridShape;  
  
    return View();  
}
```

```
// A private helper method that acts as an alias to:
_elementManager.ActivateElement.
private T New<T>(Action<T> initialize) where T:Element {
    return _elementManager.ActivateElement<T>(initialize);
}
```

Then the View looks like this:

```
@Display(ViewBag.RootElementShape)
```

When you now navigate to
`/OrchardLocal/Books.MasteringLayouts/ElementsApi`, you should see the following:

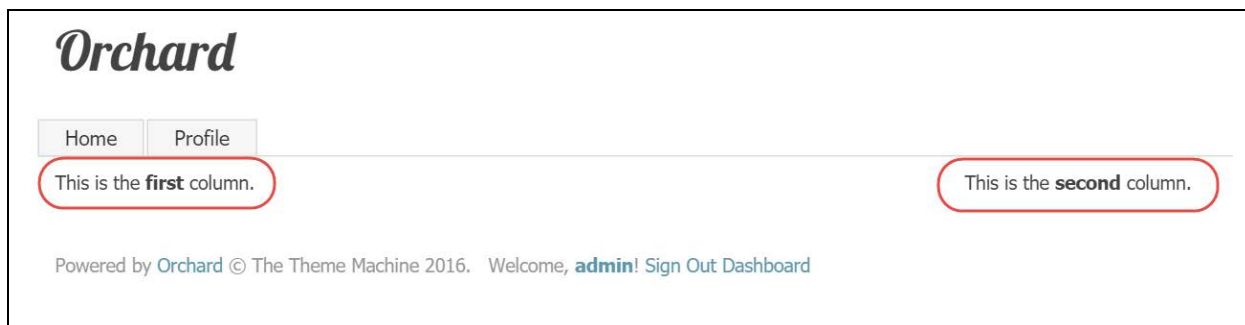


Figure 16-1 - A grid and two columns created and rendered using the Elements API

Next, let's see how we can serialize and deserialize elements.

Element Serialization

The following code demonstrates working with **ILayoutSerializer** to serialize and deserialize a hierarchy of elements. To make it a little bit more interesting, we'll provide the resulting JSON string to a **text area** control and allow the user to make changes to it and submit those changes back to the controller. We'll then deserialize the submitted JSON and render the updated set of elements.

The following code shows element serialization in action:

```
// Serialize the root element.
var json = _elementSerializer.Serialize(rootElement);
```

Next, we'll add the JSON string to the ViewBag of our controller:

```
// Assign the JSON string to a property in the dynamic ViewBag.
ViewBag.LayoutData = json;
```

Then update the view as follows:

```
<h2>@T("Layout Display")</h2>
@Display(ViewBag.RootElementShape)

<h2>@T("Layout Data")</h2>
@using (Html.BeginFormAntiForgeryPost()) {
    @Html.TextArea("LayoutData", (string) ViewBag.LayoutData, new {rows = 30, cols
= 150})
    <button type="submit">@T("Update")</button>
}
```

Now let's update the action method so that it will handle the form submission by deserializing the posted JSON string and rendering the elements.

The completed controller looks like this:

```
using System;
using System.Web.Mvc;
using Orchard.Layouts.Elements;
using Orchard.Layouts.Framework.Display;
using Orchard.Layouts.Framework.Elements;
using Orchard.Layouts.Services;
using Orchard.Themes;

namespace OffTheGrid.Demos.Layouts.Controllers {
    [Themed]
    public class ElementsApiController : Controller {
        private readonly IElementManager _elementManager;
        private readonly IElementDisplay _elementDisplay;
        private readonly IElementSerializer _elementSerializer;

        public ElementsApiController(IElementManager elementManager,
IElementDisplay elementDisplay, IElementSerializer elementSerializer) {
            _elementManager = elementManager;
            _elementDisplay = elementDisplay;
            _elementSerializer = elementSerializer;
        }
    }
}
```

```

public ActionResult Index(string layoutData = null) {

    Element rootElement;

    if (layoutData == null) {
        // Create a default hierarchy of elements.
        rootElement = New<Grid>(grid => {
            // Row.
            grid.Elements.Add(New<Row>(row => {
                // Column 1.
                row.Elements.Add(New<Column>(column => {
                    column.Width = 8;
                    column.Elements.Add(New<Html>(html => html.Content =
"This is the <strong>first</strong> column."));
                }));

                // Column 2.
                row.Elements.Add(New<Column>(column => {
                    column.Width = 4;
                    column.Elements.Add(New<Html>(html => html.Content =
"This is the <strong>second</strong> column."));
                }));
            }));

            // Serialize the root element.
            var json = _elementSerializer.Serialize(rootElement);

            // Assign the JSON string to a property in the dynamic ViewBag.
            ViewBag.LayoutData = json;
        }
    }
    else {
        rootElement = _elementSerializer.Deserialize(layoutData,
DescribeElementsContext.Empty);
    }

    // Render the Html element.
    var rootElementShape = _elementDisplay.DisplayElement(rootElement,
content: null);

    // Assign the HtmlShape to a property on the dynamic ViewBag.
    ViewBag.RootElementShape = rootElementShape;

    return View();
}

private T New<T>(Action<T> initialize) where T:Element {
    return _elementManager.ActivateElement<T>(initialize);
}
}

```

The key updates are the addition of the optional **layoutData** parameter, which, if not null, is deserialized using the following code:

```
rootElement = _elementSerializer.Deserialize(layoutData,
DescribeElementsContext.Empty);
```

So basically, we either construct a layout of elements manually, or deserialize a JSON string, and render the resulting element instance. We can now view the default JSON, manipulate it, and submit it back to the controller.

Layout Display

This is the **first** column.

This is the **second** column.

Layout Data

```
{
  "typeName": "Orchard.Layouts.Elements.Grid",
  "data": "",
  "exportableData": "",
  "index": 0,
  "elements": [
    {
      "typeName": "Orchard.Layouts.Elements.Row",
      "data": "",
      "exportableData": "",
      "index": 0,
      "elements": [
        {
          "typeName": "Orchard.Layouts.Elements.Column",
          "data": "Width=8",
          "exportableData": "",
          "index": 0,
          "elements": [
            {
              "typeName": "Orchard.Layouts.Elements.Html",
              "data": "Content=This+is+the+%3cstrong%3c%2fstrong%3e+column.",
              "exportableData": "",
              "index": 0,
              "elements": [
                {
                  "isTemplated": false,
                  "htmlId": null,
                  "htmlClass": null,
                  "htmlStyle": null,
                  "rule": null
                },
                {
                  "isTemplated": false,
                  "htmlId": null,
                  "htmlClass": null,
                  "htmlStyle": null,
                  "rule": null
                }
              ]
            },
            {
              "typeName": "Orchard.Layouts.Elements.Column",
              "data": "Width=4",
              "exportableData": "",
              "index": 1,
              "elements": [
                {
                  "typeName": "Orchard.Layouts.Elements.Html",
                  "data": "Content=This+is+the+%3cstrong%3e+second%3c%2fstrong%3e+column.",
                  "exportableData": "",
                  "index": 0,
                  "elements": [
                    {
                      "isTemplated": false,
                      "htmlId": null,
                      "htmlClass": null,
                      "htmlStyle": null,
                      "rule": null
                    },
                    {
                      "isTemplated": false,
                      "htmlId": null,
                      "htmlClass": null,
                      "htmlStyle": null,
                      "rule": null
                    }
                  ]
                },
                {
                  "isTemplated": false,
                  "htmlId": null,
                  "htmlClass": null,
                  "htmlStyle": null,
                  "rule": null
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

Update

Figure 16-2 - The text area allows editing of the JSON representation of the layout.

Working with the Layout Editor

Although editing layouts of elements with raw JSON is far from ideal, it does demonstrate the potential for implementing more advanced editors, since JSON is easy to work with.

The Layouts module itself provides a fully-functioning layout editor that we can reuse.

Let's continue with the current sample controller and replace the text area with a full-blown layout editor.

The first thing to do is to inject the **ILayoutEditorFactory** and create the **LayoutEditor** viewmodel, which we'll render from the view. We'll also change out the **IElementSerializer** with the **ILayoutSerializer**. The primary difference between the two is that the latter one expects an array of elements instead of a single element, which in turn the default **ILayoutEditorFactory** implementation expects.

> The layout editor requires the root element to be a **Canvas**, so the default implementation of **ILayoutEditorFactory** relies on **ILayoutEditorFactory**, which always serializes from and to an array of elements, the first element expected to be a Canvas. If the first element is *not* a canvas, a root Canvas element is created on the fly to which the array of elements is added.

We'll also use the **Canvas** element as the root, which is a requirement for the layout editor.

Another thing we'll change is replace the **ThemedAttribute** with the **AdminAttribute**, since the layout editor is designed to work from the backend only.

Layout JSON vs Layout Editor JSON

The Layout Editor works with a slightly different JSON schema than the schema used by the layout serializer. The reason for this is that the editor needs additional information about the layout model.

What this means is that we need to be able to convert from and to one format to the other. Fortunately, the Layouts module comes with a service for that called the **Layout Model Mapper**.

When working with **ILayoutEditorFactory**, its **Create** method expects the standard layout JSON format as its first argument. However, when the layout editor submits its data, that data is in the form of the *layout editor JSON*.

Therefore, whatever data we get back from the layout editor, it needs to be mapped back into the standard layout JSON if we want to deserialize it with the layout serializer.

Layout Editor Factory

Taking all of the above into account, the following code demonstrates how to work with the layout editor factory:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Orchard.Layouts.Elements;
using Orchard.Layouts.Framework.Display;
using Orchard.Layouts.Framework.Elements;
using Orchard.Layouts.Services;
using Orchard.Layouts.ViewModels;
using Orchard.UI.Admin;

namespace OffTheGrid.Demos.Layouts.Controllers {
    [Admin] // The layout editor is designed to work from the back end.
    public class ElementsApiController : Controller {
        private readonly IElementManager _elementManager;
        private readonly IElementDisplay _elementDisplay;
        private readonly ILayoutSerializer _layoutSerializer;
        private readonly ILayoutEditorFactory _layoutEditorFactory;
        private readonly ILayoutModelMapper _modelMapper;

        public ElementsApiController(
            IElementManager elementManager,
            IElementDisplay elementDisplay,
            ILayoutSerializer layoutSerializer,
```

```

        ILayoutEditorFactory layoutEditorFactory,
        ILayoutModelMapper modelMapper) {

    _elementManager = elementManager;
    _elementDisplay = elementDisplay;
    _layoutSerializer = layoutSerializer;
    _layoutEditorFactory = layoutEditorFactory;
    _modelMapper = modelMapper;
}

[ValidateInput(false)] // The submitted data may contain HTML.
public ActionResult Index(LayoutEditor layoutEditor /* The LayoutEditor
type serves as a viewmodel which can be modelbound. */) {

    IEnumerable<Element> layout;
    string layoutData = null;

    if (layoutEditor.Data != null) {
        // The posted layout data is not the raw Layouts JSON format, but
a more tailored one specific to the layout editor.
        // Before we can use it, we need to map it to the standard layout
format.
        layout = _modelMapper.ToLayoutModel(layoutEditor.Data,
DescribeElementsContext.Empty).ToList();

        // Serialize the layout.
        layoutData = _layoutSerializer.Serialize(layout);
    }
    else {
        // Create a default hierarchy of elements.
        layout = CreateDefaultLayout();

        // Serialize the layout.
        layoutData = _layoutSerializer.Serialize(layout);
    }

    // The session key is used for the IObjectStore service
    // used by the layout editor to transfer data to the element editor.
    // The actual value doesn't matter, just as long as its unique within
the application.
    var sessionKey = "DemoSessionKey";

    // Create and initialize a new LayoutEditor object.
    layoutEditor = _layoutEditorFactory.Create(layoutData, sessionKey);

    // Assign the LayoutEditor to a property on the dynamic ViewBag.
    ViewBag.LayoutEditor = layoutEditor;

    return View();
}

// Creates an element tree with a default layout (Grid, Row, and two
Columns).
private IEnumerable<Element> CreateDefaultLayout() {
    return new[] { New<Canvas>(canvas => {
        canvas.Elements.Add(

```

```

        New<Grid>(grid => {
            // Row.
            grid.Elements.Add(New<Row>(row => {
                // Column 1.
                row.Elements.Add(New<Column>(column => {
                    column.Width = 6;
                    column.Elements.Add(New<Html>(html => html.Content =
"This is the <strong>first</strong> column."));
                }));

                // Column 2.
                row.Elements.Add(New<Column>(column => {
                    column.Width = 6;
                    column.Elements.Add(New<Html>(html => html.Content =
"This is the <strong>second</strong> column."));
                }));
            }));
        });

        // An alias to IElementManager.ActivateElement<T>.
        private T New<T>(Action<T> initialize) where T : Element {
            return _elementManager.ActivateElement<T>(initialize);
        }
    }
}

```

Notice that I'm reusing the **LayoutEditor** class as an action parameter. This will cause the model binder to bind the posted values to an object of that class, which is very convenient, as it means we don't have to create a view model ourselves.

If a non-null value is provided for the **LayoutEditor.Data** property, we use the **ILayoutModelMapper** to parse the posted JSON string into a list of element objects, which we then serialize back to a standard JSON string. The reason for doing that is that we need to re-create the **LayoutEditor** object using the layout editor factory, since not all of its properties are being round-tripped.

If there was no data posted, we create a default layout by constructing a tree of elements manually, which we then serialize.

Whatever the source of the **layoutData** string, we feed it into the layout editor factory to get a new, properly configured, **LayoutEditor** object.

One final point of interest is the **sessionKey** variable that is being sent into the layout editor's **Create** method. This value is used for the **IObjectStore** service used by the layout editor to transfer data to the element editor. The actual value doesn't matter, just as long as it's unique within the application.

The view is updated with the following code:

```
@using Orchard.Layouts.ViewModels;
@{
    var layoutEditor = (LayoutEditor)ViewBag.LayoutEditor;
    Style.Include("~/Modules/Orchard.Layouts/Styles/default-grid.css");
}
@using (Html.BeginFormAntiForgeryPost()) {
    @Html.EditorFor(m => layoutEditor)
    <button type="submit">@T("Update")</button>
}
```

Notice the usage of the **Html.EditorFor** HTML helper method. This method will automatically select the “LayoutEditor.cshtml” view that is provided by the Layouts module to render the editor.

Now, with the updated controller and view in place, this is what our layout editor looks like when run:

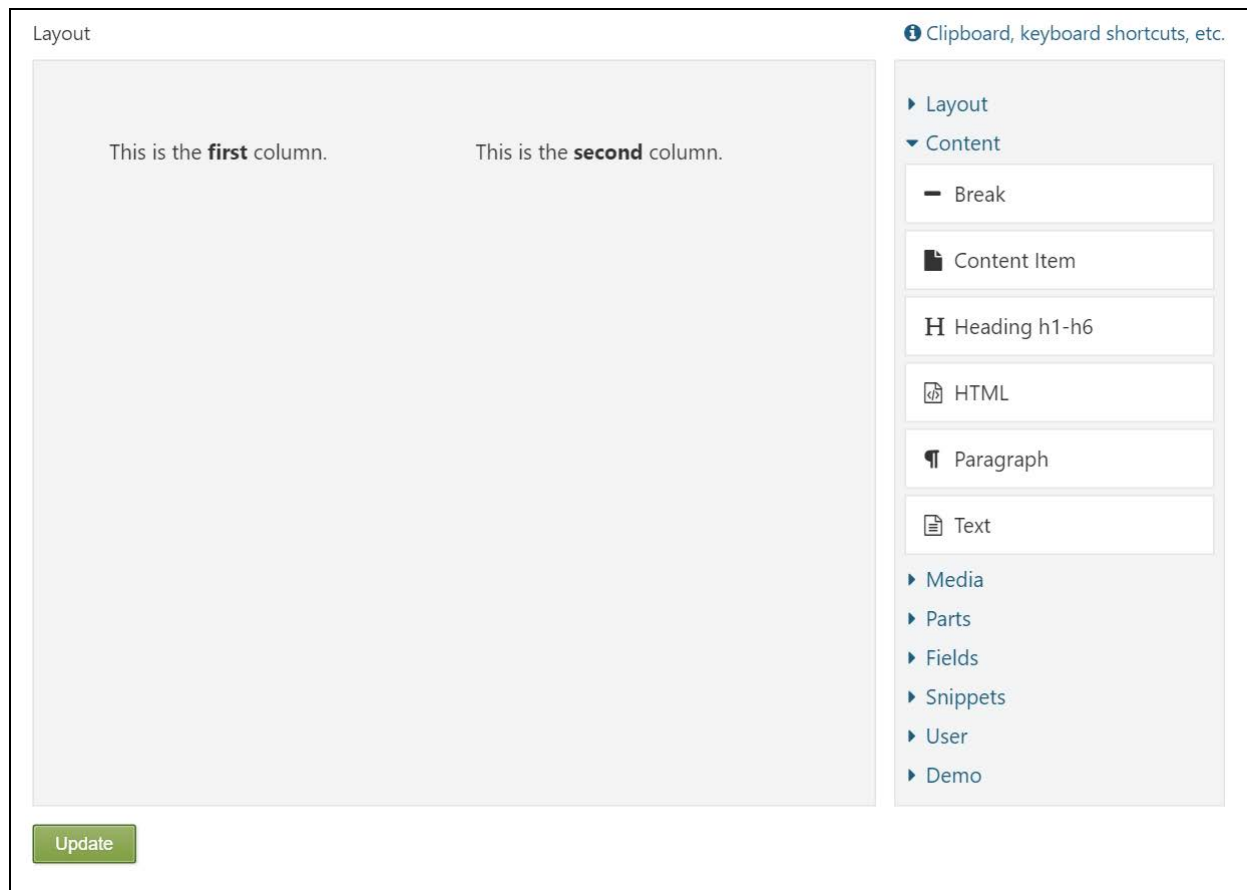


Figure 16-3 - The fully-functioning layout editor in action

As you can see, it's the fully-functioning layout editor. We can add and remove elements, and when we hit the **Update** button, the changes are persisted across form submissions.

Although we aren't storing the layout data in more durable storage such as a database, it would be easy to do so, since all we need to store and retrieve is a simple layout data string.

Some additional work that would need to be done is handle the deletion of elements. When a user deletes elements from the canvas, those elements are posted back to the controller via the **LayoutEditor.RecycleBin** property. You would model-map that string to a list of elements as well, and then invoke **IElementManager.Removing** to give each element a

chance to do some cleanup. Such code would look like this (taken from the **LayoutPartDriver**):

```
var describeContext = DescribeElementsContext.Empty;
var recycleBin = (RecycleBin)_mapper.ToLayoutModel(layoutEditor.RecycleBin,
describeContext).SingleOrDefault();
var updater = this; // This requires your controller to impement IUpdater.
var context = new LayoutSavingContext {
    Updater = updater,
    Elements = layout,
    RemovedElements = recycleBin?.Elements ?? Enumerable.Empty<Element>()
};
_elementManager.Removing(context);
```

Notice that the **RecycleBin** data string actually results in a **RecycleBin** element, which owns the removed elements.

In addition to handling the removal of elements, you would also have to invoke the **IElementManager.Saving** method, which will invoke the **Saving** event on all elements.

Summary

In this chapter we learned about various services that are provided by the Layouts module, which itself relies on them to implement the layout editor and Layout Part for example. Knowing how to use these services enable you to implement more advanced modules that rely on elements.

In the next chapter, we will take everything we learned in this chapter and use it to build a rather advanced element called **SlideShow**. We'll see how we can have that element store each individual slide and how to implement slide editors using layout and element APIs.

17. Writing a SlideShow Element

With everything learned so far, you are already able to implement many types of elements.

However, there is one scenario we haven't covered yet: element editors that span multiple pages. Let's say that your element can store a collection of things, and for each thing, you want the user to be able to edit its properties. One could certainly implement this using a single element editor screen using JavaScript techniques. But depending on the nature of such an individual item, the UI could become quite complex.

In this chapter, we will see how to implement advanced element editors by walking through the process of implementing an element called **SlideShow**.

Defining the Slide Show Element

A slide show basically consists of a collection of slides. What these slides are, that is up to us. In this example, I decided that each slide is a simple class that stores a *layout of elements*.

This means that each individual slide of the slide show element will be a layout itself. This enables the user to create any sort of layout, ranging from simple elements to more complex layouts using grid elements.

When implementing the **SlideShow** element, we will need one screen to display the slides (in thumbnail form) and the slide show player specific configuration (interval, wrap, etc). The user will also need a way to edit

each individual slide using the layout editor. For that, we'll create a custom controller that handles the *create*, *edit* and *delete* operations of a given slide.

The slide show element will have the following functionality:

- Being an element, the user can drag & drop the element onto a canvas.
- Each slide of a slideshow itself will be a layout stored as part of the element's data dictionary. This means that all the slides will be persisted as part of the element itself.
- The user can add/remove individual slides and rearrange them using drag & drop.
- The user can edit individual slides by reusing the layout editor.
- The “design” display type of the element will render the first slide.
- The slides will be animated using Bootstrap's *Carousel* component.
- The user will be able to configure various properties of the slide show's playback, such as the interval and whether to display previous/next controls.

The slide show editor will be divided into two tabs:

1. Slides
2. Player

The **Slides** tab enables the user to add, edit, remove and rearrange slides:

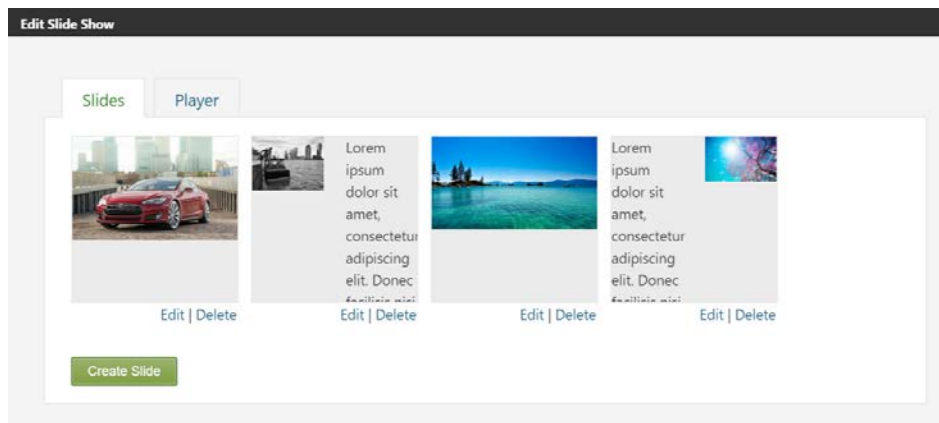


Figure 17-1 - The Slides tab of the SlideShow element editor

The **Player** tab enables the user to configure Bootstrap Carousel-specific properties:

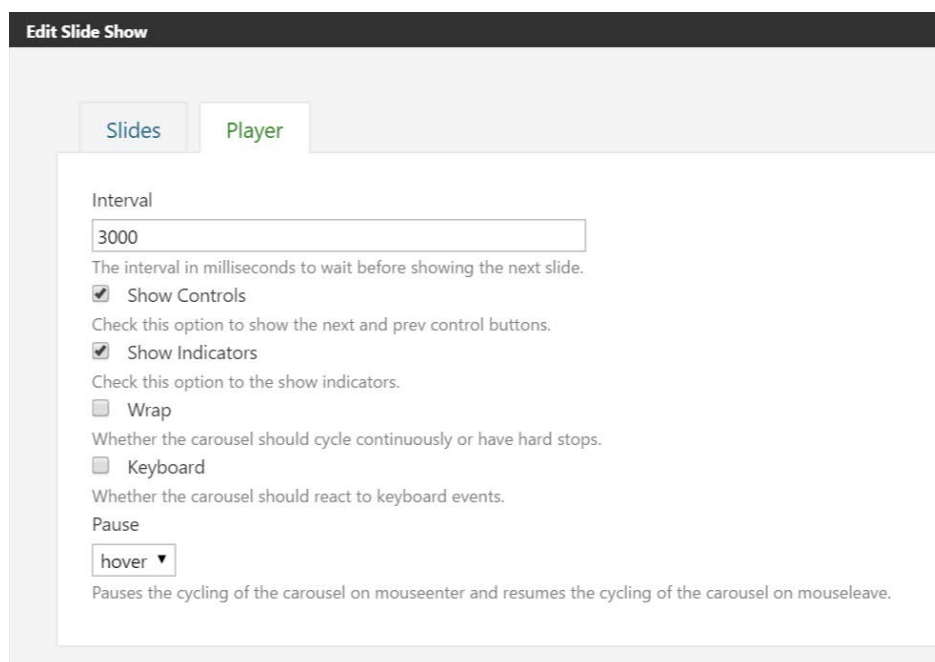


Figure 17-2 - The Player tab of the SlideShow element editor

And on the front-end, each slide will be displayed one at a time by the Bootstrap Carousel script:

Welcome to Orchard!

Sunday, March 27, 2016 5:13:00 PM

You've successfully setup your Orchard Site and this is the homepage of your new site. Here are a few things you can look at to get familiar with the application. Once you feel confident you don't need this anymore, you can [remove it by going into editing mode](#) and replacing it with whatever you want.

First things first - You'll probably want to [manage your settings](#) and configure Orchard to your liking. After that, you can head over to [manage themes to change or install new themes](#) and really make it your own. Once you're happy with a look and feel, it's time for some content. You can start creating new custom content types or start from the built-in ones by [adding a page](#), or [managing your menus](#).

Finally, Orchard has been designed to be extended. It comes with a few built-in modules such as pages and blogs or themes. If you're looking to add additional functionality, you can do so by creating your own module or by installing one that somebody else built. Modules are created by other users of Orchard just like you so if you feel up to it, [please consider participating](#).

Thanks for using Orchard – The Orchard Team



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec facilisis nisi nisi, eu convallis lorem scelerisque dignissim. Fusce vitae dolor in urna consectetur imperdiet vel ut tortor. Aliquam et luctus leo. Mauris dignissim felis sit amet odio ultricies, eget consequat massa egestas. Sed egestas tempus metus, a finibus orci aliquet a. Nulla facilisi. Donec finibus at erat in pulvinar. Sed maximus tellus lacinia pharetra eleifend. Proin venenatis magna ac massa ornare, ac vulputate neque fringilla. Proin eu varius magna, vel aliquet augue. Aenean vitae sem mi.

First Leader Aside

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur a nibh ut tortor dapibus vestibulum. Aliquam vel sem nibh. Suspendisse vel condimentum tellus.

Second Leader Aside

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur a nibh ut tortor dapibus vestibulum. Aliquam vel sem nibh. Suspendisse vel condimentum tellus.

Third Leader Aside

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur a nibh ut tortor dapibus vestibulum. Aliquam vel sem nibh. Suspendisse vel condimentum tellus.

Figure 17-3 - the SlideShow element as seen on the front-end when added to the Home page

In order to implement this type of element, you need to understand how element data is transferred from the layout editor to the element editor dialog in the first place, so that will be our first topic.

Transferring Element Data

When the user clicks the **Edit** icon on an element on the layout editor, here is what happens:

1. The element data (which is stored as part of the client-side object model) is **POST**-ed to a new window (the Element Editor Dialog). More specifically, this data is posted to the **Edit** action on the **ElementController** in the Layouts module.
2. The **Edit** action stores the posted data into something called the **object store**. The object store is basically a durable key/value store. The default implementation of **IObjectStore** relies on session state to store and retrieve entries. An important aspect of storing this data is the **key** being used. Whenever you store something in the object store, you need to provide a key. You use that same key to read the data.
3. The **Edit** action returns a redirect result to an overloaded version of the **Edit** action, passing in the key being used to store the element data. This key (called **session**) is then used to get the element data, query its drivers and display its editors.

The fact that the element state is stored in an intermediary object store is what enables us to create any number of screens we need to service our elements, since there is no direct way to retrieve the element status from the database (since elements may not have been persisted yet in the first place). Just as long as our custom controllers receive the key, we can get our data from the object store, make changes, and put it back into the store. Once we're done in the custom controller, we need to make sure to ultimately redirect back to the **ElementController**'s **Edit** action with the correct key.

Without further ado, let's start writing the SlideShow element.

The SlideShow Element Class

Start by creating a new class called **SlideShow** as follows:

```
using Orchard.Layouts.Framework.Elements;
using Orchard.Layouts.Helpers;

namespace OffTheGrid.Demos.Layouts.Elements {
    public class SlideShow : Element {
        public override string Category => "Media";
        public override string ToolboxIcon => "\uf03e";

        public int Interval {
            get { return this.Retrieve(x => x.Interval, () => 3000); }
            set { this.Store(x => x.Interval, value); }
        }

        public bool Controls {
            get { return this.Retrieve(x => x.Controls, () => true); }
            set { this.Store(x => x.Controls, value); }
        }

        public bool Indicators {
            get { return this.Retrieve(x => x.Indicators, () => true); }
            set { this.Store(x => x.Indicators, value); }
        }

        public string Pause {
            get { return this.Retrieve(x => x.Pause); }
            set { this.Store(x => x.Pause, value); }
        }

        public bool Wrap {
            get { return this.Retrieve(x => x.Wrap); }
            set { this.Store(x => x.Wrap, value); }
        }

        public bool Keyboard {
            get { return this.Retrieve(x => x.Keyboard); }
            set { this.Store(x => x.Keyboard, value); }
        }

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

Nothing we haven't seen before; just another element class with a bunch of properties. We'll allow the user to change these properties from

the element editor dialog, and use the configured values when rendering the HTML such that it works with Bootstrap Carousel.

The SlideShow Driver Class

Next, we need a driver that will handle displaying the element editor, storing the posted values back into the element, and rendering each slide object into an actual slide shape using the layout APIs.

As mentioned, each slide is a “layout”. What this means is that each slide simply stores a collection of elements, representing a hierarchy of elements.

We already know how to serialize individual layouts to JSON strings. What we need now is a way to store a list of serialized layouts. One easy way is to simply store the slides as JSON, where each slide consists of a serialized Layout string and any selected **TemplateId** (since the layout editor enables users to select a layout template).

To formalize the structure, it's a good idea to create a class that represents a slide as follows:

```
namespace OffTheGrid.Demos.Layouts.Models {
    public class Slide {
        public string LayoutData { get; set; }
        public int? TemplateId { get; set; }
    }
}
```

To help with the serialization, we'll create the following service:

```
using System.Collections.Generic;
using Orchard;

namespace OffTheGrid.Demos.Layouts.Models {
```

```

public interface ISlidesSerializer : IDependency {
    string Serialize(IEnumerable<Slide> value);
    IEnumerable<Slide> Deserialize(string value);
}

```

This serialization service can serialize a list of slides to and from JSON strings. Its implementation is as follows:

```

using System;
using System.Collections.Generic;
using System.Linq;
using Newtonsoft.Json;
using Orchard;

namespace OffTheGrid.Demos.Layouts.Models {

    public class SlidesSerializer : Component, ISlidesSerializer {
        public string Serialize(IEnumerable<Slide> value) {
            return JsonConvert.SerializeObject(value.ToList());
        }

        public IEnumerable<Slide> Deserialize(string value) {
            if (String.IsNullOrEmpty(value))
                return Enumerable.Empty<Slide>();

            return JsonConvert.DeserializeObject<List<Slide>>(value);
        }
    }
}

```

The **SlidesSerializer** is mostly just a thin wrapper around the **JsonConvert** utility class.

With that in place, it's time to define the view model that we'll use to handle the position of each slide, as well as the Bootstrap Carousel slide show player settings. The following is a list of the Bootstrap Carousel options I'd like the user to be able to configure:

Option	Description
Interval	The time to wait before moving to the next slide
Pause	The event name to use to cause the carousel to pause. Currently only one such event is supported: “hover”.
Wrap	Whether to “wrap around”, or start from the beginning, when the last slide has been displayed. If set to false, the carousel simply stops. If set to true, the carousel displays the first slide after the last one has been displayed.
Keyboard	Whether the user can use the arrow keys to move to the next or previous slide.
Controls	Whether to display the left and right arrow buttons that can be

	clicked to move to the next or previous slide.
Indicators	Whether to display indicators. Indicators visualize the number of slides using small circles, and the current slide index is visualized by highlighting the circle at that index. Also, it provides a way for the user to move to a specific slide by clicking an indicator.

In order to be able to pass the configuration as well as the slide positions to and from the view, let's define the following view model:

```
using System.Collections.Generic;
using OffTheGrid.Demos.Layouts.Elements;

namespace OffTheGrid.Demos.Layouts.ViewModels {
    public class SlideShowViewModel {
        public SlideShow Element { get; set; }
        public string Session { get; set; }
        public IList<dynamic> Slides { get; set; }
        public IList<int> Indices { get; set; }
        public string SlidesData { get; set; }
        public int Interval { get; set; }
        public bool Controls { get; set; }
        public bool Indicators { get; set; }
        public string Pause { get; set; }
        public bool Wrap { get; set; }
        public bool Keyboard { get; set; }
    }
}
```

The **Element** and **Slides** properties are “view only”, which means that although we'll set them from the driver, we don't expect any data

back to model bind these properties. They're just information that the views can use. The **Slides** property is a list of “rendered” slides, which means that each Layout stored in a slide is rendered using the **ILayoutManager.RenderLayout** method.

The following code shows the complete implementation for the driver:

```
using Orchard.Layouts.Framework.Drivers;
using OffTheGrid.Demos.Layouts.ViewModels;
using Orchard.Layouts.Framework.Display;
using System.Collections.Generic;
using OffTheGrid.Demos.Layouts.Models;
using System.Linq;
using Orchard.Layouts.Services;
using Orchard.ContentManagement;

namespace OffTheGrid.Demos.Layouts.Elements {
    public class SlideShowDriver : ElementDriver<SlideShow> {
        private ILayoutManager _layoutManager;
        private ISlidesSerializer _slidesSerializer;

        public SlideShowDriver(ILayoutManager layoutManager, ISlidesSerializer
slidesSerializer) {
            _layoutManager = layoutManager;
            _slidesSerializer = slidesSerializer;
        }

        protected override EditorResult OnBuildEditor(SlideShow element,
ElementEditorContext context) {
            var slides = RetrieveSlides(element).ToList();
            var slideShapes = DisplaySlides(slides, context.Content).ToList();
            var viewModel = new SlideShowViewModel {
                Element = element,
                Session = context.Session,
                Slides = slideShapes,
                Controls = element.Controls,
                Indicators = element.Indicators,
                Interval = element.Interval,
                Keyboard = element.Keyboard,
                Pause = element.Pause,
                Wrap = element.Wrap
            };

            if (context.Updater != null) {
                if (context.Updater.TryUpdateModel(viewModel, context.Prefix,
null, new[] { "Element", "Session", "Slides" })) {
                    var currentSlides = slides;
                    var newSlides = new List<Slide>(currentSlides.Count);
```

```

        newSlides.AddRange(viewModel.Indices.Select(index =>
currentSlides[index]));
        StoreSlides(element, newSlides);

        element.Controls = viewModel.Controls;
        element.Indicators = viewModel.Indicators;
        element.Interval = viewModel.Interval;
        element.Keyboard = viewModel.Keyboard;
        element.Pause = viewModel.Pause;
        element.Wrap = viewModel.Wrap;
    }
}

var slidesEditor = context.ShapeFactory.EditorTemplate(TemplateName:
"Elements/SlideShow.Slides", Prefix: context.Prefix, Model: viewModel);
var playerEditor = context.ShapeFactory.EditorTemplate(TemplateName:
"Elements/SlideShow.Player", Prefix: context.Prefix, Model: viewModel);

slidesEditor.Metadata.Position = "Slides:0";
playerEditor.Metadata.Position = "Player:1";

return Editor(context, slidesEditor, playerEditor);
}

protected override void OnDisplaying(SlideShow element,
ElementDisplayingContext context) {
    var slides = RetrieveSlides(element);
    context.ElementShape.Slides = DisplaySlides(slides,
context.Content).ToList();
}

private IEnumerable<dynamic> DisplaySlides(IEnumerable<Slide> slides,
IContent content) {
    return slides.Select(x => _layoutManager.RenderLayout(x.LayoutData,
content: content));
}

private IEnumerable<Slide> RetrieveSlides(SlideShow element) {
    var slidesData = element.SlidesData;
    return _slidesSerializer.Deserialize(slidesData);
}

private void StoreSlides(SlideShow element, IEnumerable<Slide> slides) {
    element.SlidesData = _slidesSerializer.Serialize(slides);
}
}
}

```

Nothing too complex. It contains code to save and load Slide objects from a given SlideShow element as well as to render them using the Layout Manager. One interesting part is how we're model-binding and then using a list called **Indices**. That list represents a list of slide indexes. When the

user rearranges slides on the client, we'll get back an array of the newly positioned indexes. The driver simply uses this new order to create a new list of slides by that order:

```
newSlides.AddRange(viewModel.Indices.Select(index => currentSlides[index]));
```

The **OnBuildEditor** method builds two **EditorTemplate** shapes: one to handle the slides, and another to handle the Bootstrap Carousel player configuration. Both editor templates use the same view model.

The **OnDisplaying** method is very simple. All it does is get the list of Slide objects and then render them. We'll look at the “Elements/SlideShow.cshtml” template shortly to see how to render these slide shapes in combination with Bootstrap related attributes.

The SlideShow Editor Views

Since the driver returns two editor template shapes, we'll have to provide two views:

- EditorTemplates/Elements/SlideShow.Slides.cshtml
- EditorTemplates/Elements/SlideShow.Player.cshtml

The first one handles the management of slides, and looks like this:

```
@model OffTheGrid.Demos.Layouts.ViewModels.SlideShowViewModel
@{
    Style.Include("~/Modules/Orchard.Layouts/Styles/default-grid.css");
    Style.Include("SlideShow.Admin.css", "SlideShow.Admin.min.css");
    Script.Require("jQuery");
    Script.Require("jQueryUI_Sortable");
    Script.Require("ShapesBase");
    Script.Include("SlideShow.Admin.js", "SlideShow.Admin.min.js");
}
@{
    var slides = Model.Slides.ToArray();
```

```

    if (!slides.Any()) {
        <div class="message message-Information">@T("No slides have been added
yet.")</div>
    }
    else {
        <div class="slides-wrapper interactive">
            <div class="dirty-message message message-Warning">@T("You have
unsaved changes.")</div>
            <div class="group">
                <ul class="slides">
                    @{ var slideIndex = 0;}
                    @foreach (var slide in slides) {
                        <li>
                            <input type="hidden" name="@Html.FieldNameFor(m =>
m.Indices)" value="@slideIndex" />
                            <div class="slide-wrapper">
                                <div class="slide-preview">
                                    @Display(slide)
                                </div>
                                </div>
                                <div class="actions">
                                    @Html.ActionLink(T("Edit").Text, "Edit",
"SlideAdmin", new { session = Model.Session, index = slideIndex, area =
"OffTheGrid.Demos.Layouts" }, null)
                                    @T(" | ")
                                    @Html.ActionLink(T("Delete").Text, "Delete",
"SlideAdmin", new { Session = Model.Session, index = slideIndex, area =
"OffTheGrid.Demos.Layouts" }, new { data_unsafe_url = T("Are you sure you want to
delete this slide?") })
                                </div>
                            </li>
                            ++slideIndex;
                        }
                    </ul>
                </div>
            </div>
        }
        @Html.ActionLink(T("Create Slide").Text, "Create", "SlideAdmin", new { Session
= Model.Session, area = "OffTheGrid.Demos.Layouts" }, new { @class = "button" })
    }

```

The reason we're including “default-grid.css” from the Layouts module is because we are rendering each individual slide (scaled down), which are in fact layouts.

Notice that I'm passing in the “Session” value when generating the links to the **Edit** and **Delete** actions of the (yet to be created) **SlideAdminController**. The controller will use this value to read the element's state from the Object Store.

We're also including a custom stylesheet called “SlideShow.Admin.css” and a custom script called “SlideShow.Admin.js”. Both files are Gulp output files, based on the following asset files:

```
Assets
  Elements
    SlideShow
      Admin.js
      Admin.less
```

Assets.json contains the following entries to turn the asset files into the mentioned output files:

```
{
  "inputs": [ "Assets/Elements/SlideShow/Admin.less" ],
  "output": "Styles/SlideShow.Admin.css"
},
{
  "inputs": [ "Assets/Elements/SlideShow/Admin.js" ],
  "output": "Scripts/SlideShow.Admin.js"
}
```

The following shows the contents of Admin.less:

```
.slides-wrapper
{
  margin-bottom: 2em;
  position: relative;
}

.slides-wrapper .sortable-placeholder
{
  display: block;
  height: 125px;
}

.slides-wrapper .dirty-message
{
  display: none;
}

.slides-wrapper ul
{
  display: none;
  float: left;
  list-style: none;
  margin: 0;
  padding: 0;
```

```

}

.slides-wrapper ul.slides li
{
    float: left;
    width: 150px;
    margin: 0 1em 0.2em 0;
    border: none;
    list-style: none;
}

.slides-wrapper ul.slides li .slide-wrapper
{
    background: #ebebeb;
    border: 1px solid #ebebeb;
}

.slides-wrapper.interactive ul.slides li:hover .slide-wrapper
{
    border-color: #aeaeae;
}

.slides-wrapper ul.slides li img
{
    vertical-align: middle;
    display: block;
    max-width: 100%;
    height: auto;
}

```

And the contents of Admin.js:

```

(function ($) {
    // Slides preview.
    var applyCssScale = function (element, scale, translate) {
        var browserPrefixes = ["", "-ms-", "-moz-", "-webkit-", "-o-"],
            offset = ((1 - scale) * 50) + "%",
            scaleString = (translate !== false ? "translate(-" + offset + ", -" +
offset + ") " : "") + "scale(" + scale + "," + scale + ")";
        $(browserPrefixes).each(function () {
            element
                .css(this + "transform", scaleString);
        });
        element
            .data({ scale: scale })
            .addClass("scaled");
    };

    var scaleSlides = function () {
        $(".slides")
            .css("display", "block")
            .each(function () {
                var slideshow = $(this),
                    slide = slideshow.find(".slide-preview"),
                    parent = slide.parent(),

```

```

        width = 150,
        height = 150,
        boundingDimension = 150,
        slideStyle = slide.attr("style");

        if ((slideStyle != null && slideStyle.indexOf("width:") == -1))
width = 1024;
        if ((slideStyle != null && slideStyle.indexOf("height:") == -1))
height = 768;

        slide.css({
            width: width + "px",
            height: height + "px",
            position: "absolute"
        });
        var scaledForWidth = width > height,
            largestDimension = (scaledForWidth ? width : height),
            scale = boundingDimension / largestDimension;

        parent.css({
            width: Math.floor(width * scale) + "px",
            height: Math.floor(height * scale) + "px",
            position: "relative",
            overflow: "hidden"
        });

        applyCssScale(slide, scale);
        slideshow.parent(".slides-wrapper").css("overflow", "visible");
    });

    };

    $(window).load(function () {
        scaleSlides();
    });

    // Sortable slides.
    $(function () {
        $(".slides-wrapper.interactive").each(function () {
            var wrapper = $(this);

            var showChangedMessage = function () {
                wrapper.find(".dirty-message").show();
            };

            var onSlideIndexChanged = function (e, ui) {
                showChangedMessage();
            };

            var slidesList = wrapper.find("ul.slides");
            slidesList.sortable({
                placeholder: "sortable-placeholder",
                stop: onSlideIndexChanged
            });
            slidesList.disableSelection();
        });
    });
});

```



```
})(jQuery);
```

The first part of the script takes care of resizing each slide layout. Now, I didn't write that code myself; Instead I borrowed it from my good friend Bertrand Leroy's work on the **Onestop.SlideShow** gallery module.

The second part of the script takes care of turning the rendered list of slides into a sortable list, which enables the user to drag and drop the slides into position.

The “SlideShow.Player.cshtml” view is much simpler:

```
@model OffTheGrid.Demos.Layouts.ViewModels.SlideShowViewModel
@{
    var pauseList = new[] { "hover" };
    var pauseOptions = pauseList.Select(x => new SelectListItem { Text = x, Value = x, Selected = x == Model.Pause }).ToList();
}
<fieldset>
    <div class="form-group">
        @Html.LabelFor(m => m.Interval, T("Interval"))
        @Html.TextBoxFor(m => m.Interval, new { @class = "text medium" })
        @Html.Hint(T("The interval in milliseconds to wait before showing the next slide."))
    </div>
    <div class="form-group">
        @Html.CheckBoxFor(m => m.Controls)
        @Html.LabelFor(m => m.Controls, T("Show Controls").Text, new { @class = "forcheckbox" })
        @Html.Hint(T("Check this option to show the next and prev control buttons."))
    </div>
    <div class="form-group">
        @Html.CheckBoxFor(m => m.Indicators)
        @Html.LabelFor(m => m.Indicators, T("Show Indicators").Text, new { @class = "forcheckbox" })
        @Html.Hint(T("Check this option to the show indicators."))
    </div>
    <div class="form-group">
        @Html.CheckBoxFor(m => m.Wrap)
        @Html.LabelFor(m => m.Wrap, T("Wrap").Text, new { @class = "forcheckbox" })
    </div>
    @Html.Hint(T("Whether the carousel should cycle continuously or have hard stops."))
    </div>
    <div class="form-group">
        @Html.CheckBoxFor(m => m.Keyboard)
```

```

        @Html.LabelFor(m => m.Keyboard, T("Keyboard").Text, new { @class =
"forcheckbox" })
        @Html.Hint(T("Whether the carousel should react to keyboard events.))
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Pause, T("Pause"))
        @Html.DropDownListFor(m => m.Pause, pauseOptions, T("never").ToString())
        @Html.Hint(T("Pauses the cycling of the carousel on mouseenter and resumes
the cycling of the carousel on mouseleave.))
    </div>
</fieldset>

```

Next, let's have a look at the “Elements/SlideShow.cshtml” template itself as well as its alternate “Elements/SlideShow.design.cshtml”. The latter one will be used when the SlideShow element is rendered in the Layout editor:

```

@{
    var slides = (IList<dynamic>)Model.Slides;
}
@if (!slides.Any()) {
    <div class="message message-Information">@T("No slides have been added
yet.")</div>
}
else {
    var firstSlide = slides.First();
    <div class="slides-wrapper img-responsive">
        @Display(firstSlide)
    </div>
}

```

As you can see, all we're doing here is getting the first rendered slide shape, if any, and render that one.

The “Elements/SlideShow.cshtml” is more interesting, as it contains the actual markup that we need for the Bootstrap Carousel component to work:

```

@using OffTheGrid.Demos.Layouts.Elements
@{
    Style.Include("SlideShow.css", "SlideShow.min.css");
    Script.Require("jQuery");
    Script.Include("SlideShow.js", "SlideShow.min.js");

    var element = (SlideShow)Model.Element;
    var slides = (IList<dynamic>)Model.Slides;
}

```

```

var slideShowId = String.Format("Slideshow-{0}", DateTime.Now.Ticks);
}
@if (slides.Any()) {
<div id="@slideShowId" class="carousel slide"
    data-ride="carousel"
    data-interval="@element.Interval"
    data-wrap="@element.Wrap.ToString().ToLower()"
    data-keyboard="@element.Keyboard.ToString().ToLower()"
    data-pause="@element.Pause">
    @if (element.Indicators) {
        <!-- Indicators -->
        <ol class="carousel-indicators">
            @for (var i = 0; i < slides.Count; i++) {
                <li data-target="#@slideShowId" data-slide-to="@i" @if (i ==
0) { <text> class="active" </text> }></li>
            }
        </ol>
    }

    <!-- Wrapper for slides -->
    <div class="carousel-inner" role="listbox">
        @for (var i = 0; i < slides.Count; i++) {
            <div class="item @if(i == 0){<text>active</text>}">
                @Display(slides[i])
            </div>
        }
    </div>

    @if (element.Controls) {
        <!-- Controls -->
        <a class="left carousel-control" href="#@slideShowId" role="button"
data-slide="prev">
            <span class="glyphicon glyphicon-chevron-left" aria-
hidden="true"></span>
            <span class="sr-only">@T("Previous")</span>
        </a>
        <a class="right carousel-control" href="#@slideShowId"
role="button" data-slide="next">
            <span class="glyphicon glyphicon-chevron-right" aria-
hidden="true"></span>
            <span class="sr-only">@T("Next")</span>
        </a>
    }
</div>
}

```

Both the SlideShow.css and SlideShow.js are actually Bootstrap's Carousel component's script and CSS that I stored in the Assets folder and processed using Gulp, but those files are too long to show here. Instead. If you're coding along, you can get the files from the accompanying demo source code.

I essentially copied the markup from the <http://getbootstrap.com> website and replaced some of it with some Razor code to conditionally render the indicators and controls for example. As for the slides, all I had to do is iterate over each of them and then render:

```
@for (var i = 0; i < slides.Count; i++) {  
    <div class="item @if(i == 0){<text>active</text>}">  
        @Display(slides[i])  
    </div>  
}
```

At this point, only one major thing is missing to make it all work: actually being able to create and edit individual slides.

The SlideAdmin Controller Class

In order to enable the user to create, edit and delete individual slides, we will need a controller to handle those actions. When creating and editing a slide, we will have the controller reuse the layout editor.

Let's go ahead and create a new controller called **SlideAdminController** with the following skeleton code:

```
using System.Web.Mvc;  
using OffTheGrid.Demos.Layouts.ViewModels;  
using Orchard.UI.Admin;  
  
namespace OffTheGrid.Demos.Layouts.Controllers {  
    [Admin]  
    public class SlideAdminController : Controller {  
  
        public ActionResult Create(string session) {  
        }  
  
        [HttpPost]  
        [ValidateInput(false)]  
        public ActionResult Create(SlideEditorViewModel viewModel) {  
        }  
  
        public ActionResult Edit(string session, int index) {  
        }  
    }  
}
```

```

        [HttpPost]
        [ValidateInput(false)]
        public ActionResult Edit(SlideEditorViewModel viewModel, int index) {
        }

        [HttpPost]
        public ActionResult Delete(string session, int index) {
        }
    }
}

```

I added action methods for **Create**, **Edit** and **Delete**, including the “post-back” variants.

The Create Action

The first action we'll implement is the Create action, which will render the layout editor and handle post-back. Notice that we're expecting the **session** key value to be provided by the slide show element editor.

```

public ActionResult Create(string session) {
    var viewModel = new SlideEditorViewModel {
        Session = session,
        LayoutEditor = _layoutEditorFactory.Create(null,
        _objectStore.GenerateKey())
    };
    return View(viewModel);
}

```

What we did here is instantiate a new **SlideEditorViewModel** and a **LayoutEditor** using the layout editor factory API, so be sure to inject an **ILayoutEditorFactory**.

The **SlideEditorViewModel** is defined as follows:

```

using Orchard.Layouts.ViewModels;

namespace OffTheGrid.Demos.Layouts.ViewModels {
    public class SlideEditorViewModel {
        public string Session { get; set; }
        public int? SlideIndex { get; set; }
        public LayoutEditor LayoutEditor { get; set; }
    }
}

```

```
}
```

The **SlideIndex** is needed when we edit an individual slide. Since the object store stores the entire slide show's element data, including its slides, we need a way to know which slide we need to edit.

The “Create.cshtml” view looks like this:

```
@model OffTheGrid.Demos.Layouts.ViewModels.SlideEditorViewModel
@{
    Layout.Title = T("Create Slide");
}
@using (Html.BeginFormAntiForgeryPost()) {
    @Html.HiddenFor(m => m.Session)
    @Html.HiddenFor(m => m.SlideIndex)
    @Html.EditorFor(m => m.LayoutEditor)
    <fieldset>
        <button type="submit" name="submit.Save"
value="submit.Save">@T("Create")</button>
        @Html.ActionLink(T("Cancel").ToString(), "Edit", "Element", new { session
= Model.Session, area = "Orchard.Layouts" }, new { @class = "button" })
    </fieldset>
}
```

To implement the post-back, we need to do the following:

- Get a reference to the slide show element's data being edited and activate the element.
- Get the slides from the slide show element so that we can add a new slide to it.
- Serialize the list slides and store it back into the slide show element.
- Serialize and store the updated slide show element into the object store.
- Redirect back to the slide show element edit screen.

The implementation looks like this:

```

[HttpPost]
[ValidateInput(false)]
public ActionResult Create(SlideEditorViewModel viewModel) {
    var slideShowSessionState = _objectStore.Get<ElementSessionState>(session);
    var elementData =
        ElementDataHelper.Deserialize(slideShowSessionState.ElementData);
    var slideShow = _elementManager.ActivateElement<SlideShow>(x => x.Data =
        elementData);
    var slides = _slidesSerializer.Deserialize(slideShow.SlidesData).ToList();
    var slide = new Slide();
    var slideLayout =
        _layoutModelMapper.ToLayoutModel(viewModel.LayoutEditor.Data,
        DescribeElementsContext.Empty).ToList();
    var recycleBin =
        (RecycleBin)_layoutModelMapper.ToLayoutModel(viewModel.LayoutEditor.RecycleBin,
        DescribeElementsContext.Empty).First();
    var context = new LayoutSavingContext {
        Updater = this,
        Elements = slideLayout,
        RemovedElements = recycleBin.Elements
    };

    // Trigger the Saving and Removing events on the elements.
    _elementManager.Saving(context);
    _elementManager.Removing(context);

    // Create a new slide.
    var slide = new Slide {
        TemplateId = viewModel.LayoutEditor.TemplateId;
        LayoutData = _layoutSerializer.Serialize(slideLayout);
    };

    // Add the slide.
    slides.Add(slide);

    // Serialize the in-memory list and assign it back to the SlidesData property
    of the SlideShow element.
    slideShow.SlidesData = _slidesSerializer.Serialize(slides);

    // Serialize the slide show element itself.
    slideShowSessionState.ElementData =
        ElementDataHelper.Serialize(slideShow.Data);

    // Replace the slide show in the object store with the updated data.
    _objectStore.Set(session, slideShowSessionState);

    // Display a success notification.
    _notifier.Information(T("That slide has been added."));

    // Redirect back to the element editor screen.
    return RedirectToAction("Edit", "Element", new { session =
        viewModel.Session, area = "Orchard.Layouts" });
}

```

Notice that this method relies on a few new services that we haven't declared yet:

- **__elementManager** of type **IElementManager**
- **__slidesSerializer** of type **ISlidesSerializer**
- **__layoutModelMapper** of type **ILayoutModelMapper**
- **__layoutSerializer** of type **ILayoutSerializer**
- **__objectStore** of type **IObjectStore**
- **INotifier** of type **INotifier**
- **T** of type **Localizer**

Make sure to inject these services and store them in private fields. Except for the **T** property, which is property-injected by Orchard rather than constructor-injected.

The Edit Action

The Edit action is very similar to the Create action, the primary differences being:

1. The view model needs to be initialized with existing layout data.
2. When posting back, instead of adding a slide, we need to get a reference to an existing slide (by index).

The Edit action looks like this:

```
public ActionResult Edit(string session, int index) {
    // Get the element's state from the object store.
    var slideShowSessionState = _objectStore.Get<ElementSessionState>(session);

    // Deserialize the element's state into a dictionary.
    var elementData =
        ElementDataHelper.Deserialize(slideShowSessionState.ElementData);
```



```

    // Activate a new SlideShow element and initialize it with the state
    dictionary.
    var slideShow = _elementManager.ActivateElement<SlideShow>(x => x.Data =
    elementData);

    // Deserialize the slide show slides data into a list of actual Slide objects.
    var slides = _slidesSerializer.Deserialize(slideShow.SlidesData).ToList();

    // Get a reference to the specified slide by index.
    var slide = slides[index];

    // Create the view model and initialize the layout editor with the slide's
    layout data.
    var viewModel = new SlideEditorViewModel {
        SlideIndex = index,
        Session = session,
        LayoutEditor = _layoutEditorFactory.Create(slide.LayoutData,
        _objectStore.GenerateKey(), slide.TemplateId)
    };

    // Return the view.
    return View(viewModel);
}

```

The Edit view looks very similar to the Create view:

```

@model OffTheGrid.Demos.Layouts.ViewModels.SlideEditorViewModel
@{
    Layout.Title = T("Edit Slide");
}
@using (Html.BeginFormAntiForgeryPost()) {
    @Html.HiddenFor(m => m.Session)
    @Html.HiddenFor(m => m.SlideIndex)
    @Html.EditorFor(m => m.LayoutEditor)
    <fieldset>
        <button type="submit" name="submit.Save"
value="submit.Save">@T("Save")</button>
        @Html.ActionLink(T("Cancel").ToString(), "Edit", "Element", new { session
= Model.Session, area = "Orchard.Layouts" }, new { @class = "button" })
    </fieldset>
}

```

The only real difference is the title being set to “Edit Slide”.

The post-back version of the Edit action, too, looks very similar to the Create action method:

```

[HttpPost]
[ValidateInput(false)]
public ActionResult Edit(SlideEditorViewModel viewModel) {

```

```

        var slideShowSessionState = _objectStore.Get<ElementSessionState>(session);
        var elementData =
            ElementDataHelper.Deserialize(slideShowSessionState.ElementData);
        var slideShow = _elementManager.ActivateElement<SlideShow>(x => x.Data =
            elementData);
        var slides = _slidesSerializer.Deserialize(slideShow.SlidesData).ToList();
        var slide = slides[viewModel.SlideIndex.Value];
        var slideLayout =
            _layoutModelMapper.ToLayoutModel(viewModel.LayoutEditor.Data,
            DescribeElementsContext.Empty).ToList();
        var recycleBin =
            (RecycleBin)_layoutModelMapper.ToLayoutModel(viewModel.LayoutEditor.RecycleBin,
            DescribeElementsContext.Empty).First();
        var context = new LayoutSavingContext {
            Updater = this,
            Elements = slideLayout,
            RemovedElements = recycleBin.Elements
        };

        // Trigger the Saving and Removing events on the elements.
        _elementManager.Saving(context);
        _elementManager.Removing(context);

        // Update the slide.
        slide.TemplateId = viewModel.LayoutEditor.TemplateId;
        slide.LayoutData = _layoutSerializer.Serialize(slideLayout);

        // Serialize the in-memory list and assign it back to the SlidesData property
        of the SlideShow element.
        slideShow.SlidesData = _slidesSerializer.Serialize(slides);

        // Serialize the slide show element itself.
        slideShowSessionState.ElementData =
            ElementDataHelper.Serialize(slideShow.Data);

        // Replace the slide show in the object store with the updated data.
        _objectStore.Set(session, slideShowSessionState);

        // Display a success notification.
        _notifier.Information(T("That slide has been updated.));

        // Redirect back to the element editor screen.
        return RedirectToAction("Edit", "Element", new { session =
            viewModel.Session, area = "Orchard.Layouts" });
    }

```

The Delete Action

The Delete action is responsible for removing a slide at the specified index. In terms of implementation, it is similar to the Create and Edit methods, except for the fact that you don't need to deal with the view model. All we need to do is materialize the slide show element, remove a

slide from its list, then serialize and store it back into the object store. The implementation looks like this:

```
[HttpPost]
public ActionResult Delete(string session, int index) {
    var slideShowSessionState = _objectStore.Get<ElementSessionState>(session);
    var elementData =
        ElementDataHelper.Deserialize(slideShowSessionState.ElementData);
    var slideShow = _elementManager.ActivateElement<SlideShow>(x => x.Data =
        elementData);
    var slides = _slidesSerializer.Deserialize(slideShow.SlidesData).ToList();

    // Delete the slide at the specified index.
    slides.RemoveAt(index);

    // Serialize the in-memory list and assign it back to the SlidesData property
    of the SlideShow element.
    slideShow.SlidesData = _slidesSerializer.Serialize(slides);

    // Serialize the slide show element itself.
    slideShowSessionState.ElementData =
        ElementDataHelper.Serialize(slideShow.Data);

    // Replace the slide show in the object store with the updated data.
    _objectStore.Set(session, slideShowSessionState);

    // Redirect back to the element editor. The returnUrl contains the session
    key.
    _notifier.Information(T("That slide has been deleted.));

    // Redirect back to the element editor screen.
    return RedirectToAction("Edit", "Element", new { session =
        viewModel.Session, area = "Orchard.Layouts" });
}
```

And that's really all there is to it! With this code, you can now drag & drop slide show elements onto the canvas and manage individual slides.

Suggestions for Improvements

Although being able to craft individual slides using the layout editor is powerful, it is also potentially cumbersome in cases where all you really need is a list of simple images. In that case, it would be much nicer if the user had an option to simply select one or more images from the media library, and automatically generate a list of slides, where each slide would

still be based on a layout, but with one single Image element. This has the benefit of quickly creating slide shows while retaining the ability to edit individual ones.

Another improvement could be the decoupling of the slide show player. Although the theme could use completely different carousel scripts rather than Bootstrap Carousel, right now the slide show element only supports Bootstrap Carousel-specific settings. You could consider making this set of settings more universal, which has the upside of potentially supporting any carousel script out there, but also has the disadvantage that some settings may not be applicable to certain players, while certain advanced players provide settings that the user cannot configure.

Similar to decoupling the actual player, another improvement could be to decouple the source from which the slide show element gets its slides from. For example, imagine a Query from the Projections module that yields a set of content items, which you want to display as a slide show. Or another provider based on a Twitter feed.

Summary

In this chapter, we put various layout APIs to the test and implemented a much more advanced element editor than we have seen before. We implemented a Slide Show element that enables the user to create and edit individual slides, each slide being very dynamic in terms of content thanks to the reusability of the layout editor.