Contents

[1. Introduction to Layouts 8](#_Toc445413381)

[Defining Layouts 8](#_Toc445413382)

[Orchard and Layouts 8](#_Toc445413383)

[Layout and Elements 9](#_Toc445413384)

[When to use Orchard.Layouts? 10](#_Toc445413385)

[Option 1 - Direct HTML manipulation 11](#_Toc445413386)

[Option 2 - Widgets and Zones 11](#_Toc445413387)

[Option 3 - Content Fields and Placement.info 12](#_Toc445413388)

[Enter Orchard.Layouts 13](#_Toc445413389)

[Where did the Body Part go? 14](#_Toc445413390)

[What happens to my existing site and its contents when upgrading to Orchard 1.9? 14](#_Toc445413391)

[The Nature of Elements 15](#_Toc445413392)

[Does Orchard.Layouts work with grid systems such as Bootstrap? 15](#_Toc445413393)

[Summary 16](#_Toc445413394)

[2. First Look 17](#_Toc445413395)

[The Main Players 17](#_Toc445413396)

[The Layouts Feature 17](#_Toc445413397)

[The Layout Part 17](#_Toc445413398)

[Elements 17](#_Toc445413399)

[The Layout Editor 18](#_Toc445413400)

[Working with the Layout Editor 20](#_Toc445413401)

[Element Editor Controls 20](#_Toc445413402)

[Keyboard Support 23](#_Toc445413403)

[Moving Elements within its Container 26](#_Toc445413404)

[Moving Elements across Containers 26](#_Toc445413405)

[Re-sizing Columns 26](#_Toc445413406)

[Layouts on the Front-end 26](#_Toc445413407)

[Summary 27](#_Toc445413408)

[3. Meet the Elements 28](#_Toc445413409)

[Element Categories 28](#_Toc445413410)

[Layout 30](#_Toc445413411)

[Grid 30](#_Toc445413412)

[Row 31](#_Toc445413413)

[Column 32](#_Toc445413414)

[Canvas 33](#_Toc445413415)

[Content 34](#_Toc445413416)

[Break 34](#_Toc445413417)

[Content Item 34](#_Toc445413418)

[Heading 35](#_Toc445413419)

[Html 35](#_Toc445413420)

[Markdown 35](#_Toc445413421)

[Paragraph 36](#_Toc445413422)

[Projection 36](#_Toc445413423)

[Text 36](#_Toc445413424)

[Media 37](#_Toc445413425)

[Image 37](#_Toc445413426)

[Media Item 37](#_Toc445413427)

[Vector Image 38](#_Toc445413428)

[Parts 38](#_Toc445413429)

[Placeable Parts 38](#_Toc445413430)

[Fields 39](#_Toc445413431)

[Snippets 39](#_Toc445413432)

[Shape 39](#_Toc445413433)

[Snippet Elements 40](#_Toc445413434)

[UI 40](#_Toc445413435)

[Widgets 42](#_Toc445413436)

[Summary 42](#_Toc445413437)

[4. Layout Templates 43](#_Toc445413438)

[Sealed Elements and Placeholder Containers 44](#_Toc445413439)

[Summary 44](#_Toc445413440)

[5. Element Blueprints 46](#_Toc445413441)

[When to use Element Blueprints 46](#_Toc445413442)

[Creating Element Blueprints 46](#_Toc445413443)

[Trying it out: Creating an Element Blueprint 47](#_Toc445413444)

[Summary 50](#_Toc445413445)

[6. Elements as Widgets 51](#_Toc445413446)

[Why Elements as Widgets? 51](#_Toc445413447)

[Using Elements as Widgets 52](#_Toc445413448)

[Element Wrapper Part 52](#_Toc445413449)

[Trying it out: Creating a Widget based on an Element 52](#_Toc445413450)

[Existing Widgets based on Elements 54](#_Toc445413451)

[Summary 54](#_Toc445413452)

[7. Element Tokens 55](#_Toc445413453)

[The Element.Display Token 55](#_Toc445413454)

[Trying it out: Using Element.Display 56](#_Toc445413455)

[Summary 58](#_Toc445413456)

[8. Element Rules 59](#_Toc445413457)

[Available Functions 59](#_Toc445413458)

[Applying Element Rules 60](#_Toc445413459)

[Summary 61](#_Toc445413460)

[9. Theming 62](#_Toc445413461)

[A Primer on Shapes 62](#_Toc445413462)

[Anatomy of a Shape 63](#_Toc445413463)

[Shape Templates 64](#_Toc445413464)

[Elements and Shapes 66](#_Toc445413465)

[Overriding Element Shape Templates 68](#_Toc445413466)

[Custom Alternates 69](#_Toc445413467)

[Trying it out: Creating Custom Alternates for Element Shapes 69](#_Toc445413468)

[Content Part and Field Elements 71](#_Toc445413469)

[Updating Placement.info 72](#_Toc445413470)

[Summary 73](#_Toc445413471)

[10. Bootstrap 74](#_Toc445413472)

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.

# 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.

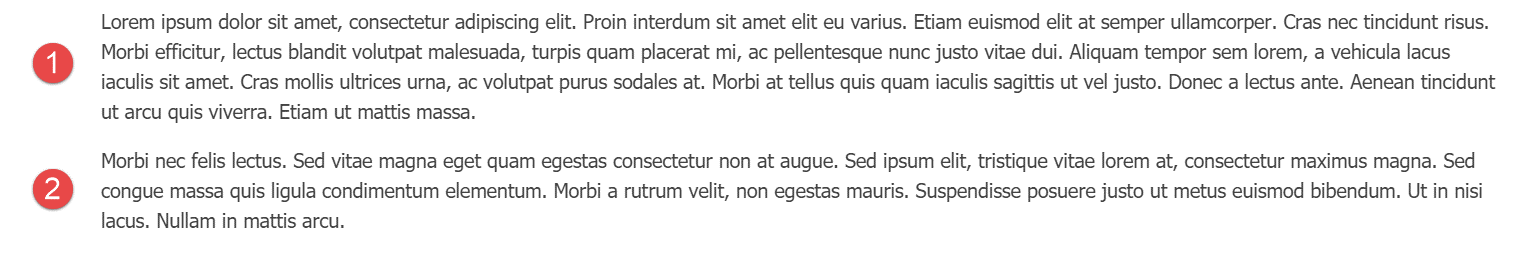


Figure 1‑1 – Two paragraphs, vertically stacked.

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

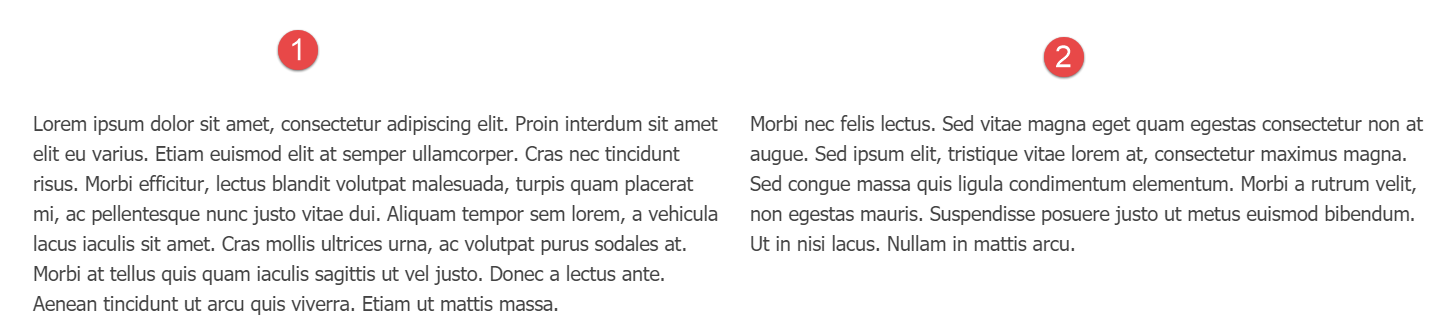


Figure 1‑2 – The same two paragraphs, horizontally laid out.

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.

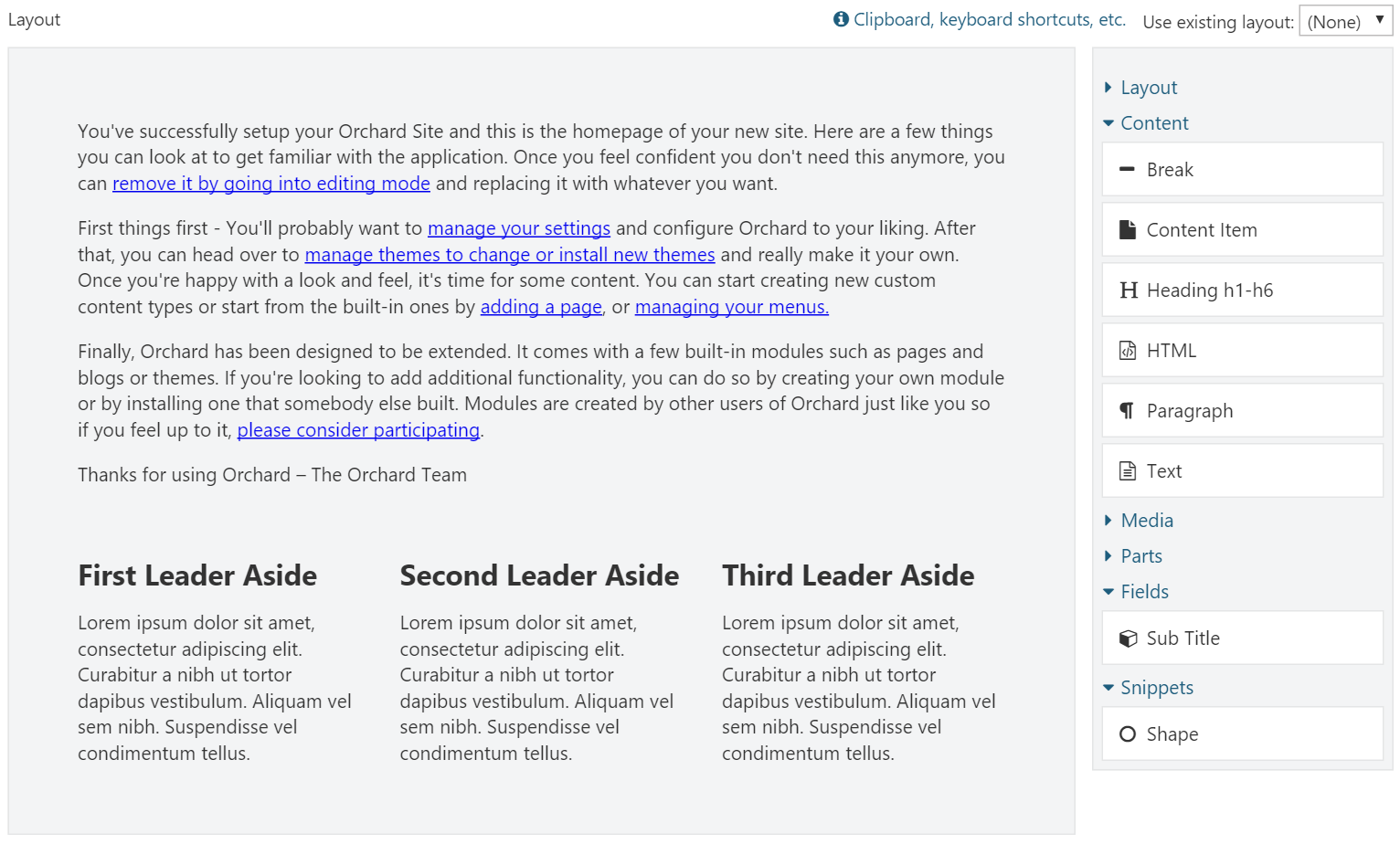


Figure 1‑3- The Layout Editor.

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.

# 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).

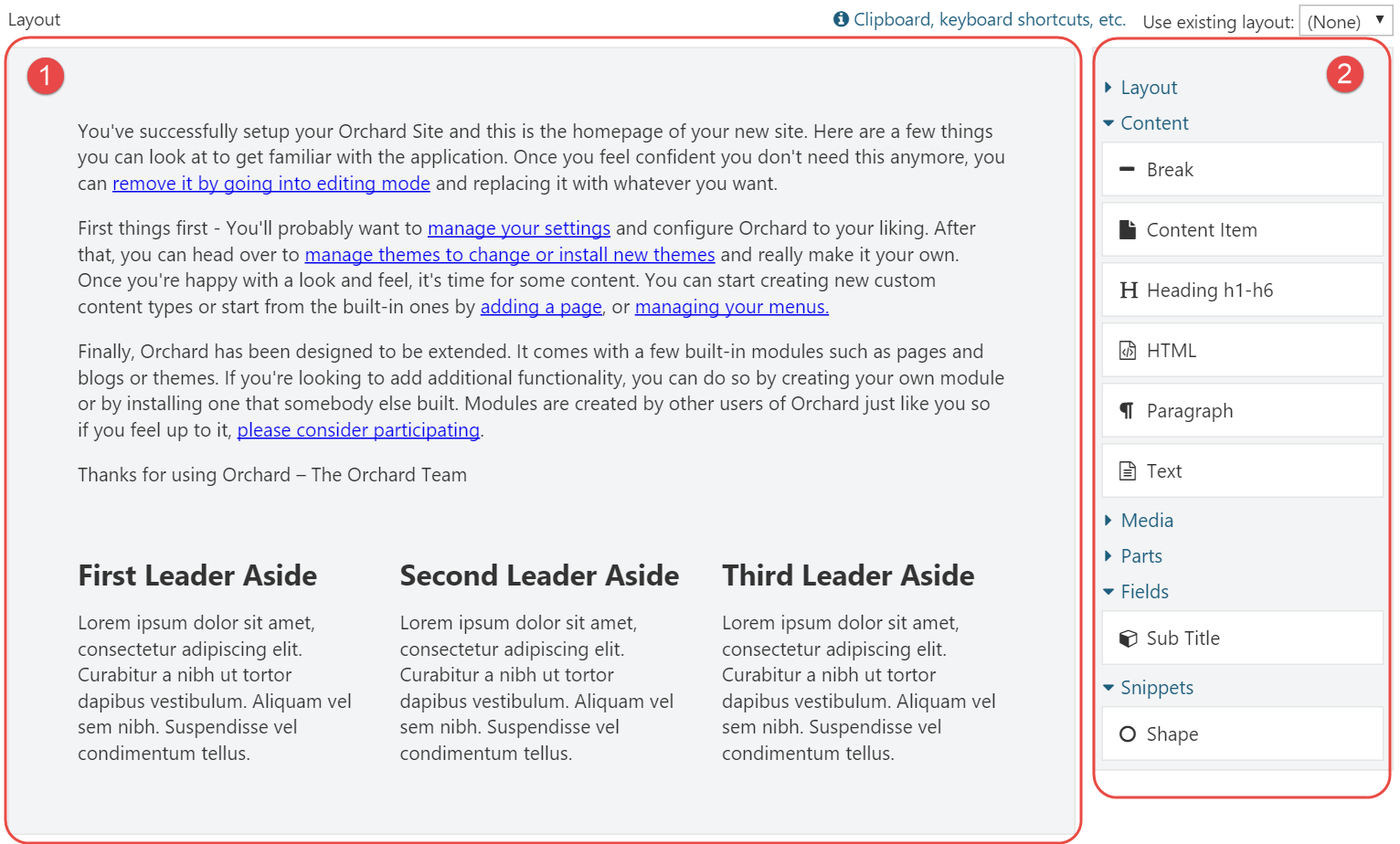


Figure 2‑1 – The Layout Editor consists of the canvas (1) and the toolbox (2).

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.

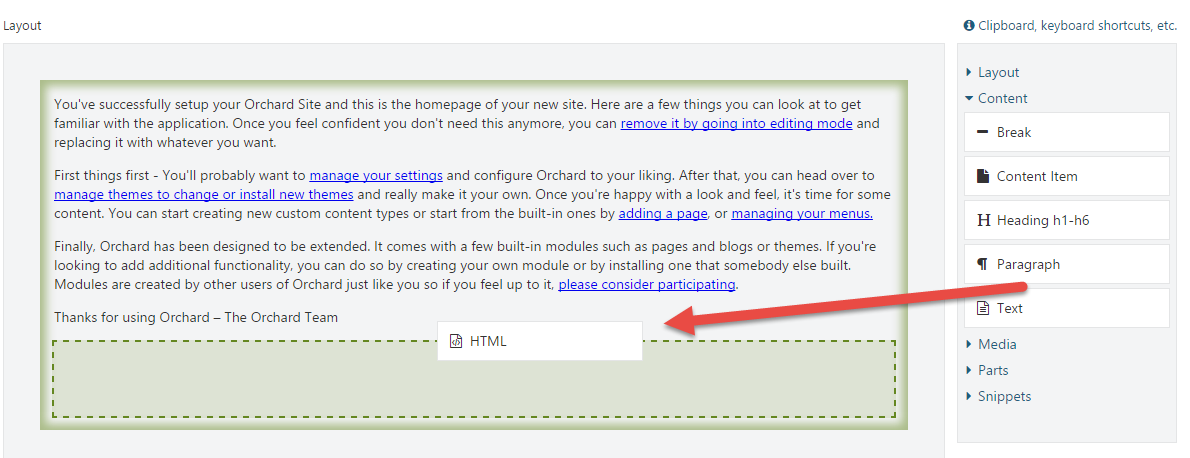


Figure 2‑2 – The user drags and drops elements from the toolbox to the canvas.

## 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.

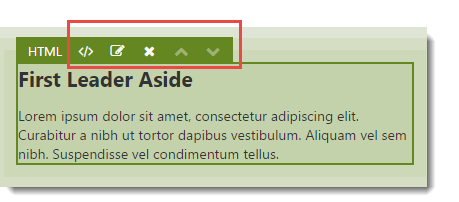


Figure 2‑3 - Each element has a toolbar to control properties of and perform operations to the element.

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.

# 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.

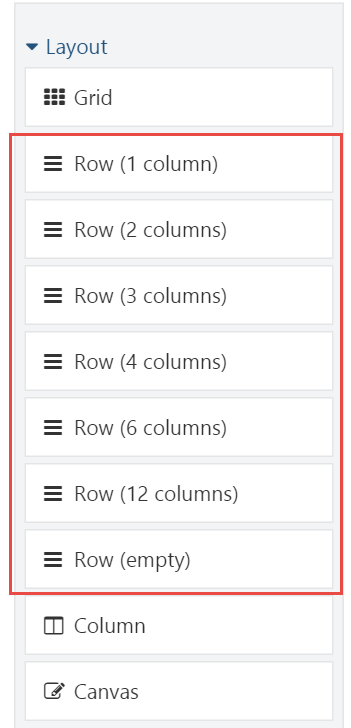


Figure 3‑1 - The Row element is represented in the toolbox as a set of pre-configured rows.

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 doesis 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 **<img>** 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 <img> 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 <img> 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.

# 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.

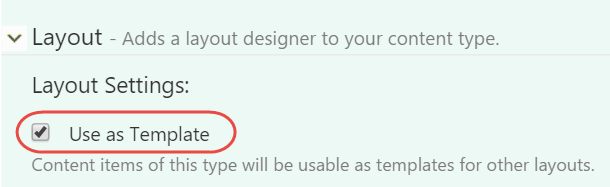


Figure 4‑1 - The Use as Template setting on the Layout Part turns the content type into a layout template.

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

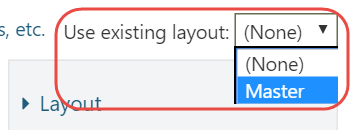


Figure 4‑2 - The Layout Editor displays a dropdown list with available templates.

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.

# 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 like this:

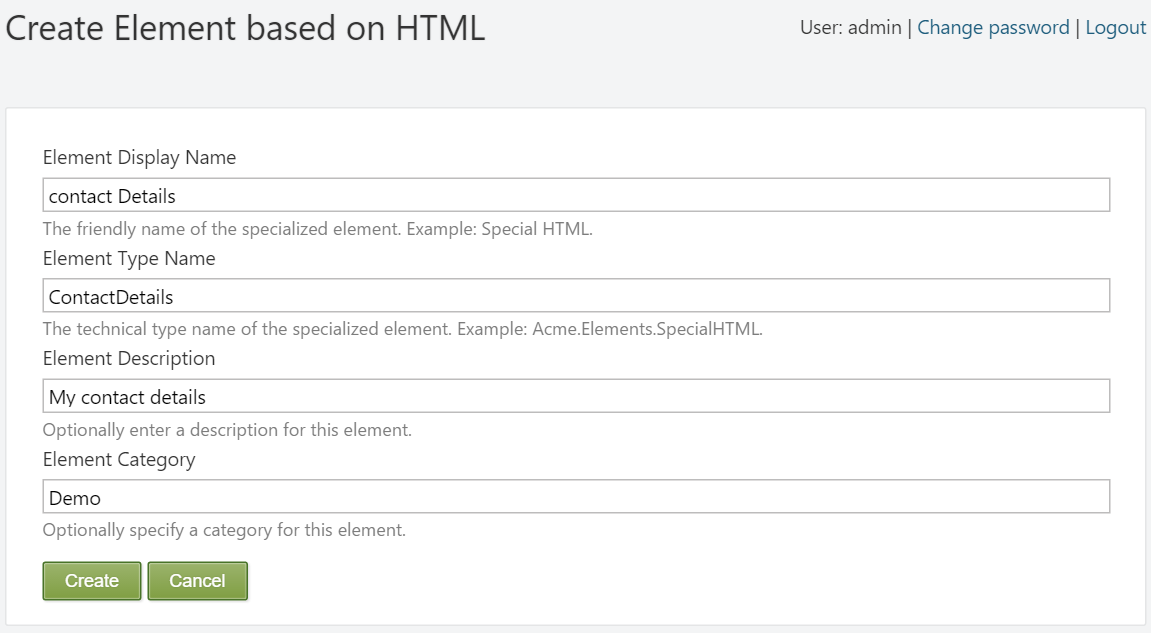


Figure 5‑1 – Creating a new Element Blueprint.

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](mailto:j.vandyke@acme.com%3c/a%3e%3c/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.

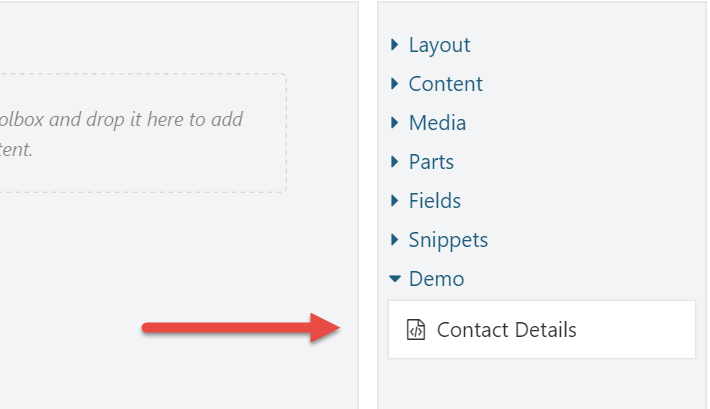


Figure 5‑2 - The newly created element is available from the toolbox.

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.

# 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.

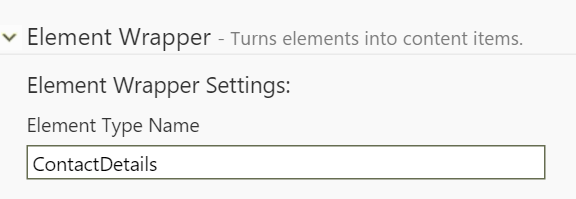


Figure 6‑1 - The Element Wrapper Part settings,

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

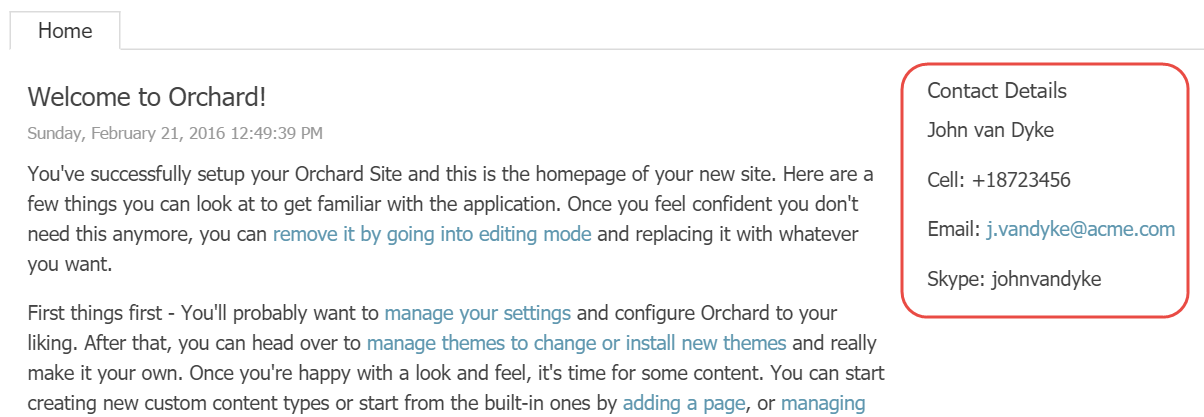


Figure 6‑2 - The Contact Details widget as it appears on the front-end in the AsideSecond zone.

## 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.

# 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.

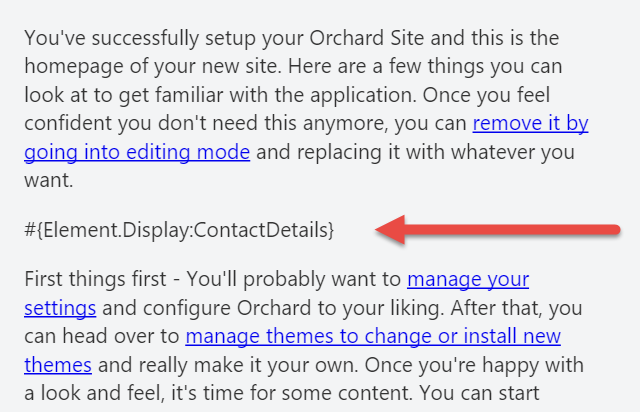


Figure 7‑1 - Enter the Element.Display token somewhere in the Html element's contents.

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.

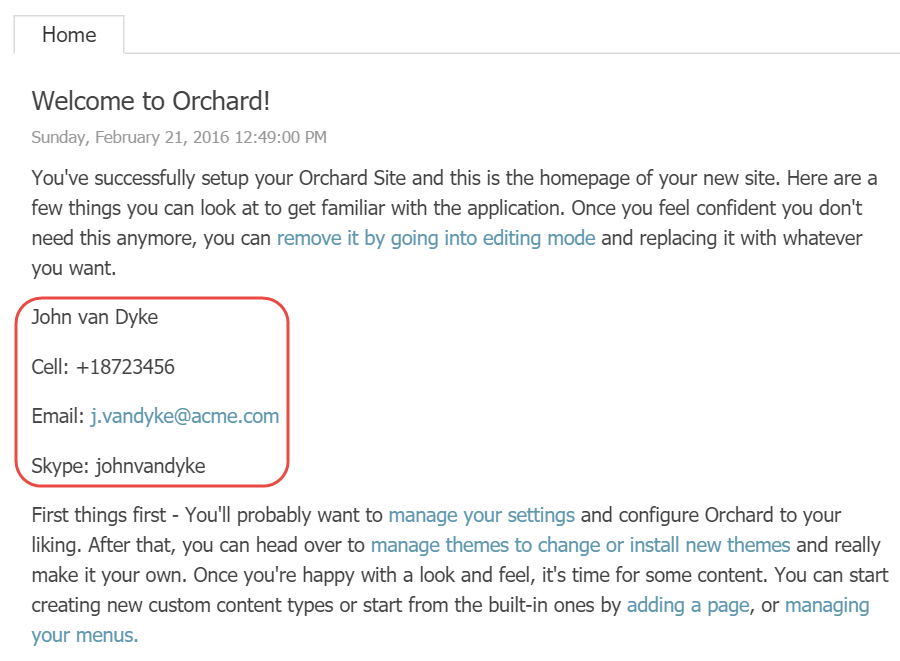


Figure 7‑2 - The Element.Display token is dynamically replaced with its actual display.

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.

# 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: contentype(“Page”) |
| url | Evaluates to true if the current content item being displayed is of the specified content type, false otherwise. Example: url('~/contact') |

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.

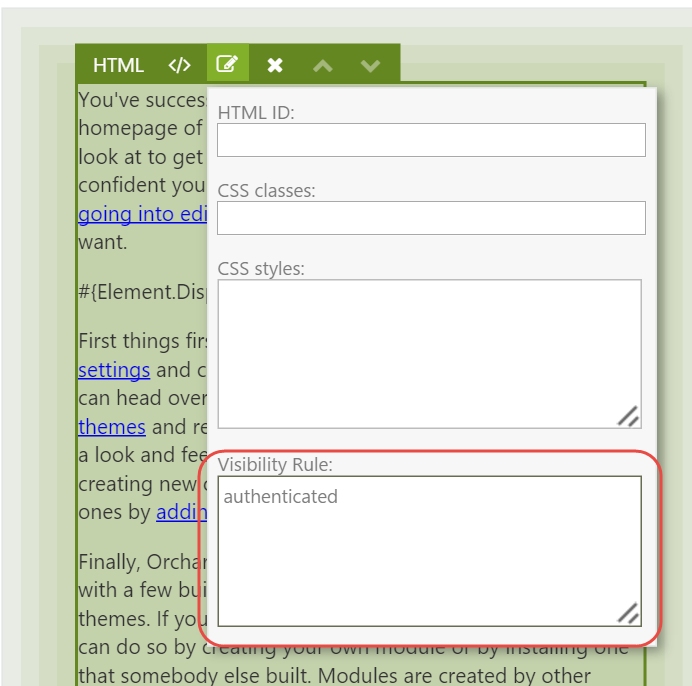


Figure 8‑1 - The Element Rule editor.

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.

# 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 run-time, using an internal dictionary to store members and their values.

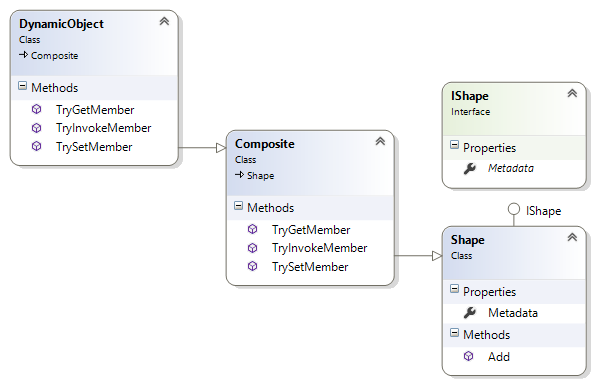


Figure 9‑1 - The class hierarchy of the Shape class.

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 mssage "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 form 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.

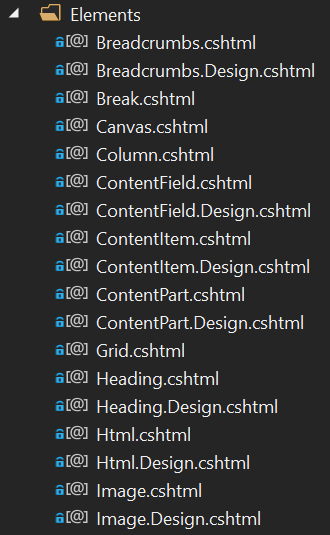


Figure 9‑2 - Element shape templates can be organized in Views/Elements.

## 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.IsNullOrWhiteSpace(htmlId))

return;

var typeName = element.GetType().Name;

htmlId = htmlId.ToSafeName();

// Example: Elements/Html-MyJumbotron.cshtml

var alternate = $"Elements\_{typeName}\_\_{htmlId}";

context.ShapeMetadata.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.

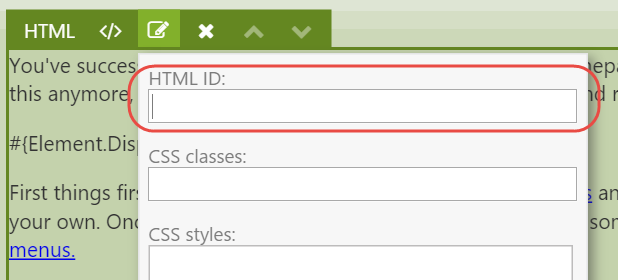


Figure 9‑3 - Every element has an HTML ID property.

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.

# 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 created grids. For example, take the following layout:

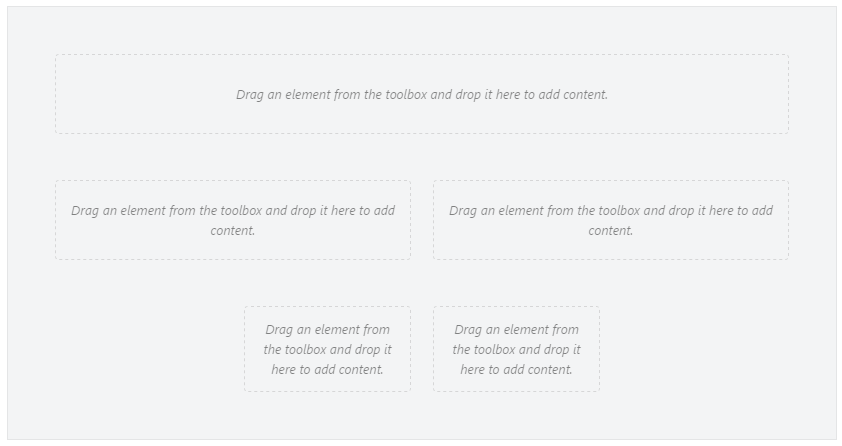


Figure 10‑1 - A layout created with the Layout Editor.

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:

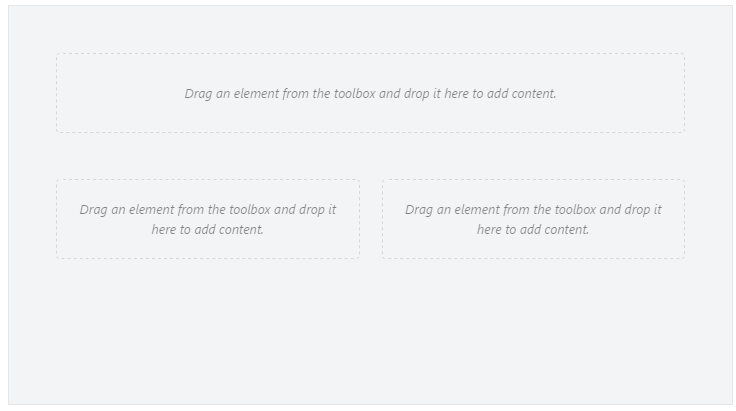


Figure 10‑2

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

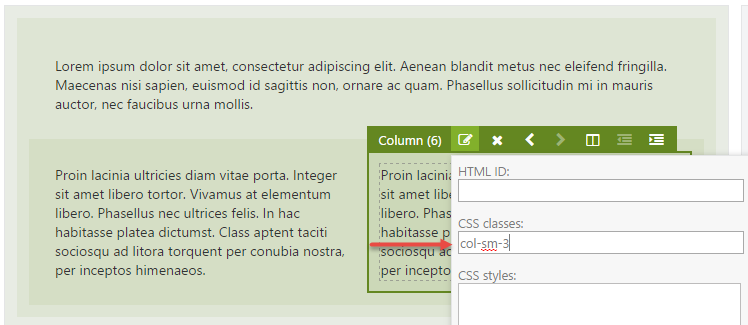


Figure 10‑3 - Setting the CSS classes property to col-sm-3. Don't forget to also set the CSS classes of the left column to col-sm-9.

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](http://getbootstrap.com/css).