

ASP.NET Website Development Made Easy



Orchard CMS

Up and Running

O'REILLY®

John Zablocki

www.it-ebooks.info

Orchard CMS: Up and Running

Use your C# skills to build your next website with Orchard, the popular content management system based on ASP.NET MVC. With step-by-step guidance, you'll learn your way around the Orchard environment by constructing a complete, real-world site throughout the course of this book. You'll create, manage, and display dynamic content with out-of-the-box functionality, and then build themes, modules, and widgets to customize the site.

Author John Zablocki gets you started by showing you how to obtain and compile the Orchard source code, so you can more efficiently customize and manage the sites you create.

- Create or extend Orchard content types to manage dynamic content
- Use alternate templates to change the way Orchard displays content
- Design a theme to define your website's look and feel
- Build custom modules when the Orchard Gallery doesn't have extensions you need
- Create reusable content pieces by creating widgets
- Explore options for adding multi-language support to a site
- Learn hosting options for your Orchard sites, including the cloud
- Package your custom themes and modules to share in Orchard Gallery

Purchase the ebook edition of this O'Reilly title at oreilly.com and get free updates for the life of the edition. Our ebooks are optimized for several electronic formats, including PDF, EPUB, Mobi, APK, and DAISY—all DRM-free.

Twitter: @oreillymedia
facebook.com/oreilly

US \$19.99

CAN \$20.99

ISBN: 978-1-449-32021-8



5 1 9 9 9

O'REILLY®
oreilly.com

Orchard CMS: Up and Running

John Zablocki

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

www.it-ebooks.info

Orchard CMS: Up and Running

by John Zablocki

Copyright © 2012 John Zablocki. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Rachel Roumeliotis

Production Editor: Melanie Yarbrough

Proofreader: Melanie Yarbrough

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Robert Romano

Revision History for the First Edition:

2012-05-24 First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449320218> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Orchard CMS: Up and Running*, the cover image of a Brush-tailed Bettong, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-32021-8

[LSI]

1337864455

To Lady and Mary Katherine

Table of Contents

Preface	ix
 1. Getting Started with Orchard	 1
Prerequisites	1
Development Environment	1
Obtaining the Orchard Solution	2
Packaged Releases	2
Using Source Control	2
The Contents of the Solution	4
Setting Up the Website	4
Configuring Orchard for the First Time	4
The Orchard Dashboard	7
Creating Content	7
Adding Widgets	10
Orchard Modules	12
Orchard Security	13
Summary	14
 2. Creating and Managing Content	 15
Bio Items	15
Content Types	16
Content Parts	16
Content Fields	18
Projections and Queries	20
Filters	21
Sort Criteria	21
Layouts	21
Event Items	22
Content Types and Fields Continued	22
Projections and Queries Continued	24
Daisy's Blog	25

The Blog Module	26
About Page	27
Contact Page	28
Home Page	30
Zones and Layers	31
Gallery	32
Summary	34
3. Displaying Content	35
Customizing Biography Content	35
Content Templates	36
Alternate Templates	37
Customizing Events	41
Placement Files	41
Field Templates	42
Shape Tracing	43
Summary	47
4. Creating Themes	49
The Orchard Command-Line Interface	49
Getting Help	49
Code Generation Tools	50
Code Generating a Theme	51
The Structure of a Theme	52
Default Content Templates	53
Working with Views	54
Layouts	55
Zones and Layers	56
Alternate Templates	57
Theme Inheritance	57
Basic Styling	60
Styling Projections	61
Shape Wrapping	62
Theme Metadata	63
Theme Previews	63
Theme Credits	63
Summary	64
5. Creating Modules	65
The Places Field	65
Getting Places Data	66
Module Code Generation	66
The Places Field Project	67

Places Field Model	67
Drivers	69
Field Templates	72
Settings	75
Controllers	77
Module Metadata	80
Using the Places Field	80
Creating Content with the Places Field	80
Displaying the Places Field	83
Summary	84
6. Creating Widgets	85
Content Parts	85
The JW Player Widget	85
Creating the Module	86
JW Player Model	87
Database Migrations	88
Handlers and Drivers	90
Placement	91
Enabling Our Module	92
A Second Migration	92
The Widget Views	93
Adding the Widget to a Zone	95
Widget Metadata	96
Summary	98
7. Localization	99
Orchard Site Localization	99
Admin Localization	99
PO Files	101
Content Item Localization	102
Summary	104
8. Maintaining Orchard Sites	105
Packaging Themes and Modules	105
Deploying Orchard Sites	107
Shared Hosting	107
Dedicated Hosting	108
Cloud Hosting	108
Multi-Tenancy	108
Modules and Performance	108
Workflow	109
Upgrading	109

Summary	110
9. Conclusion	111

Preface

A few months back, I wanted to create an online presence for my band, Daisy's Gone. In the past, I would have started from scratch. But I remembered all the domains I've accumulated over the years that are now nothing more than parking pages at my registrar. It's generally not hard to throw together a few simple brochure-ware pages. I certainly could have done just that for my band's site. However, even simple sites often have content and behavior that needs to be dynamic.

The obvious solution is to use a content management system (CMS). CMS platforms such as WordPress and Drupal offer users prebuilt functionality for everything from creating pages to managing site registration. A full-blown CMS will allow non-technical users to create and manage content and will allow programmers and designers to extend the out-of-the-box functionality by creating themes and modules, respectively.

With Daisy's Gone, I was once again presented with the dilemma of whether to build a site from scratch or to use a CMS. Fortunately, I remembered how I had seen a then just-released Orchard CMS used at the NYC Give Camp a few months earlier. Give Camps match developers and designers with charities who have some unmet technical need, often a web presence. One of the developer groups built its charity a new website using Orchard.

I remember being quite impressed by how much this team accomplished in one week-end using this new CMS. So when it came time to start building the website for my band, I made the choice to use Orchard.

The original Daisy's Gone website was not much more than a home page. It was primarily a sandbox for me to learn Orchard. As I write this book, I'm going to create a new online presence for the band. I'll walk through the steps of creating a custom look and feel (themes) and extending Orchard with new functionality (modules).

Whether you are building a new corporate site for your company or a site for the local youth soccer league, you are likely to have many of the same needs of your CMS. Building a site for a band is no different. You may need to schedule events, manage user comments, support OAuth, or have a site map. The content will vary by your domain, the features much less so.

About the Orchard Project

Microsoft released Orchard in January 2011, along with ASP.NET MVC 3, IIS Express, SQL CE 4, Web Farm Framework, and WebMatrix. What all of these technologies have in common is that they, in some way, aim to make web development on Windows more accessible. PHP owes its wide adoption to its perceived ease of use. With this new tool chain, Microsoft is courting the community who wants a simplified development experience.

The Orchard Project belongs to the Microsoft supported Outercurve Foundation (formerly known as the CodePlex Foundation). The Outercurve Foundation is a nonprofit organization whose mission is to foster support for open source projects, such as Orchard and the popular ASP.NET toolkit, MVC Contrib. Microsoft does not officially support Orchard, but its employees are currently among those leading the development efforts for the project.

There are three stated goals of the Orchard Project. The first is to provide a set of shared components to be used in ASP.NET applications. The second is to create a set of reference applications built using these components. The third is to build a community to support these components and applications. At the time of this writing, Orchard is primarily a platform for creating content-driven websites. Though in Orchard, content is certainly not limited to blog posts or simple text-heavy pages.

Orchard is developed with a full open source stack. It uses ASP.NET MVC 3.0 with the Razor view engine. There are also numerous dependencies on other open source projects, most notably NHibernate for data access and Autofac for dependency injection. Orchard is licensed under the BSD license.

Not much more than a year after its 1.0 release, Orchard has been downloaded nearly 1,000,000 times. More than 300 modules and themes have been created and submitted to the Orchard Gallery. A series of minor releases have taken Orchard to its current 1.4 version. The Orchard community is strong and growing.

Why Another .NET CMS?

There have been several commercial and open source .NET CMS products over the years. DotNetNuke (DNN) is arguably the most notable and most popular. However, it was written in VB.NET and remained that way until earlier this year when it was ported to C#. VB.NET was a deal-breaker for me, as it was for many developers.

Even though DNN is a C# project now—as is another popular open source .NET CMS, Umbraco—both are WebForms projects. Like VB.NET, WebForms is also a deal-breaker for me. While the underlying web framework or programming language used by a CMS is of little consequence to an end user, to a programmer it will likely be important. As an MVC developer, I've wanted a CMS that is built on ASP.NET MVC and uses metaphors that are familiar to MVC development. I believe the .NET

developer community will gravitate towards Orchard, because its development stack is more in line with modern ASP.NET development practices.

That said, it is entirely possible to build content and functionality-rich sites using Orchard without having to write a single line of code. Even though Orchard is a relatively new product, it has a remarkably rich set of extensions. Like other modern CMS products, Orchard marries great programmatic extensibility with rich, out-of-the-box functionality.

Audience

This book is being written for web developers who want to create content-heavy websites without starting from a blank slate. I assume readers have some familiarity with basic CMS concepts and some level of web development skill. Experience with Orchard is neither assumed nor required. Programming custom modules for Orchard does require knowledge of C# and to a lesser extent ASP.NET MVC. If you are unfamiliar with MVC, I suggest reading *20 Recipes for Programming MVC 3* or *Programming ASP.NET MVC 4* (O'Reilly) or *Programming Microsoft ASP.NET MVC* (Microsoft Press).

Many Orchard users will be able to accomplish their goals without having to write any custom code. This book will fully cover how to get a site up and running with Orchard and will walk readers through customization with existing themes and sites from the Orchard Gallery. However, even users who will not be building custom modules might benefit from the chapters describing module and theme architecture. Knowing how Orchard works will help users debug problems that might arise in production.

Contents of This Book

This book introduces Orchard development by walking a reader through the process of creating a customized Orchard site.

[Chapter 1](#) covers getting an Orchard site set up for development and provides a quick tour of the Orchard experience.

[Chapter 2](#) details the process of creating content from existing types of content and covers how to create new types of content.

[Chapter 3](#) describes how to change the way Orchard displays content by default.

[Chapter 4](#) explores the process of customizing an Orchard site's look and feel by building a theme.

[Chapter 5](#) provides a walkthrough of creating and configuring a custom module.

[Chapter 6](#) provides a walkthrough of creating and configuring a custom widget.

[Chapter 7](#) explores options for adding multi-language support to an Orchard site.

[Chapter 8](#) discusses considerations for maintaining Orchard sites.

[Chapter 9](#) contains the final thoughts on what is covered in this book.

Companion Material

All code samples in this book are available on Bitbucket in my OrchardCMS repository located at <https://bitbucket.org/johnzablocki/orchardcms>. I use Mercurial for version control. Any general purpose module created in this book will also be available for download in the Orchard Gallery. And of course, you will be able to view the finished project at <http://www.daisysgone.com>.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

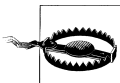
Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require

permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Orchard CMS: Up and Running* by John Zablocki (O'Reilly). Copyright 2012 John Zablocki, 978-1-449-32021-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



Safari Books Online (www.safaribooksonline.com) is an on-demand digital library that delivers expert [content](#) in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of [product mixes](#) and pricing programs for [organizations](#), [government agencies](#), and [individuals](#). Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens [more](#). For more information about Safari Books Online, please visit us [online](#).

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreil.ly/orchardcms_upandrunning

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

While a team of developers is able to build Orchard full-time, the rest of the Orchard ecosystem comes from the work of the community. I thank these volunteers who leave blog posts and code for Bing to find and the rest of us to follow. Without their efforts, there would not be an Orchard about which to write this book.

I'd like to thank my editor, Rachel Roumeliotis, for giving me the opportunity to write this book. Rachel has been great to work with from the beginning. She understood when a demanding new job slowed my writing pace and when a new release of Orchard invalidated nearly half of my work. I hope to be able to work again with Rachel.

I'd also like to thank O'Reilly Media and its employees who helped get this book published. I realize I am just one piece of many in the puzzle that is turning a bunch of Word documents in a Dropbox folder into a printed work.

Finally, I am forever grateful and thankful to my technical reviewer, Mark Freedman. Mark has long been a colleague and a mentor. He's helped me not only with this book, but with so many aspects of my career. I consider myself fortunate to have started my career many years ago under Mark's management. Twelve years later, Mark's guidance has again proved invaluable. I couldn't have written this book without him.

Getting Started with Orchard

We're about to start building an Orchard website. We'll create some content. We'll manage some content. We'll change the way our site looks and behaves. We'll write some code to extend the functionality that's available out of the box. Though we could perform all of these tasks without ever looking at the Orchard source, we're .NET developers. We're most comfortable in Visual Studio, so why wouldn't we start there?

Prerequisites

Though it's not entirely necessary, it's my preference to build modules, create themes, and manage my Orchard sites all within the context of the full Visual Studio 2010 Orchard solution. Aside from being able to debug the site with Visual Studio, having the source handy also provides a great reference when creating your own Orchard extensions. We'll learn how to develop extensions in the chapters ahead.

Development Environment

Orchard extensions are known as modules. Creating a module requires writing code, typically C#, but any .NET language will work. You could write that code in Notepad or any text editor of your choice, but that wouldn't be the most efficient way to work. In this chapter and those that follow, I assume that you'll be working with Visual Studio 2010 Professional or higher.

The Orchard documentation contains tutorials on how to perform tasks, such as creating modules, without Visual Studio. While it certainly is possible to do so, it's impractical to consider this approach for all but the most basic of Orchard workflows. You could use the Express editions of Visual C# and Visual Web Developer. However, you won't be able to open the master solution. Express editions don't support mixed project type (web and class library) solutions.



A \$500 IDE might seem to be an expensive barrier to entry for an open source project such as Orchard. To lower the bar, the Orchard team has provided documentation and tooling that supports non-Visual Studio workflows. The open source IDE MonoDevelop is probably the best alternative, as you can work within the master Orchard solution.

Obtaining the Orchard Solution

There are two ways to get the Orchard solution onto your local development environment. The simplest way is to go to the Orchard downloads at [CodePlex.com](http://codeplex.com), which is Microsoft's open source project hosting site. Though I sometimes use the source control option described below for my Orchard development, I'm instead going to use a packaged release for this book. That will guarantee we're all using the same source. The solution I'm using for this book can be found at <http://orchard.codeplex.com/releases/view/74491>.

Packaged Releases

With each release (major or minor), Orchard zip files are made available on this page. One is a precompiled version of Orchard, which is optimized for users who want to deploy Orchard and won't be coding extensions for the site. This package is labeled *Orchard.Web.1.4.0.zip*. At the time of this writing, 1.4.0 is the current version. This ZIP file appears under the heading "Recommended Download."



The URL <http://orchard.codeplex.com/releases/view/74491> will always point to the 1.4.0 release. Click the "Downloads" tab on the page to find downloads for the latest version as it may have changed by the time this book is published.

A second ZIP file download is a snapshot of the Orchard source code as it existed for the release. This package is labeled *Orchard.Source.1.4.0.zip*. Download this zip file and extract it to work with the most recent and stable release of Orchard. This ZIP file appears under the heading "Other Available Downloads." Each release will have different zip files available, but the web and source downloads should be consistently available.

Using Source Control

As an alternative, you could get the Orchard source by cloning the source control repository. This is the method I prefer for module development, as it guarantees that I will be developing against the most recent committed changes. Of course, there is a risk that a feature in the current codebase will not make the final cut. For this reason, to follow along with the examples in this book you should download the source as

described previously in this chapter. However, I'll describe how to clone the source later in case you wish to try that option instead.

The Orchard team uses the *distributed version control system* (DVCS) Mercurial. You don't need to understand much about Mercurial (Hg) in order to use it to get the Orchard source. I won't cover anything beyond where to download Mercurial and the two commands you'll use with Mercurial and Orchard. If you would like more information on Mercurial, I suggest reading *Mercurial: The Definitive Guide* by Bryan O'Sullivan (O'Reilly).

Mercurial is an open source project and is freely available at <http://mercurial.selenic.com/>. There is a Windows shell extension called TortoiseHg, which allows you to access common commands by right-clicking on folders in Windows Explorer. When you download and install TortoiseHg, the command line client is also installed. After running the installer, open up a command prompt and enter the following command:

```
hg clone https://hg01.codeplex.com/orchard Orchard
```

The clone command in Mercurial is loosely analogous to a checkout command in Subversion or Team Foundation Server. One significant difference is that cloning copies the entire repository locally. With a DVCS such as Mercurial, there is no centralized repository.

The first argument passed to the clone command is the URL where the repository may be found. The second is the name of the local directory to where your repository will be copied. This path is relative to the directory from which you ran the hg command. In other words, if the command prompt opened into `C:\users\John`, you'd have a new folder: `C:\users\John\Orchard`.



If the hg command was not recognized by Windows, it is likely that the path to `hg.exe` was not added to your system's path environment variable. By default this path is `C:\Program Files (x86)\Mercurial`.

If you choose to obtain the source by way of cloning the repository, it's a good practice to get the latest changesets, or commits, to the repository. Open a command prompt to the directory into which you cloned Orchard and run the following command:

```
hg pull
```

The **pull** command will get the latest version of the Orchard sources. However, Mercurial doesn't by default add updates into your working copy. To update your local repository with any changes found during a pull, run the following command:

```
hg update
```

The Contents of the Solution

If you navigate to the directory where you either unzipped the source package or cloned the Mercurial repository, you should see two directories *src* and *lib*. Open up the *src* directory and locate the file *Orchard.sln*, which is the Orchard solution. Open this solution file and build it.

Setting Up the Website

After you’ve successfully compiled the Orchard source, you’ll need to create a virtual directory in *Internet Information Services* (IIS). Name this virtual directory “Orchard.” The physical path of the virtual directory should be set to the location of the Orchard web project. If you copied the Orchard source to a directory *c:\dev\Orchard*, then the physical path would be *c:\dev\Orchard\src\Orchard.Web*.

Alternatively, you could simply use the ASP.NET Development Server that ships with Visual Studio. In fact, the default settings of the Orchard solution are to run the *Orchard.Web* project using this server. While, generally speaking, the development server should be adequate for the purpose of Orchard site development, I prefer to work with IIS when possible. IIS will be what you use in production, so you’re more likely to catch certain issues in development with a local IIS setup.

After you’ve created the virtual directory, open up a web browser and navigate to <http://localhost/orchard>. If you are using the ASP.NET Development Server, simply run the *Orchard.Web* project using Ctrl+F5 to start the site without debugging (or just F5 to start with debugging). You should see the screen in [Figure 1-1](#).



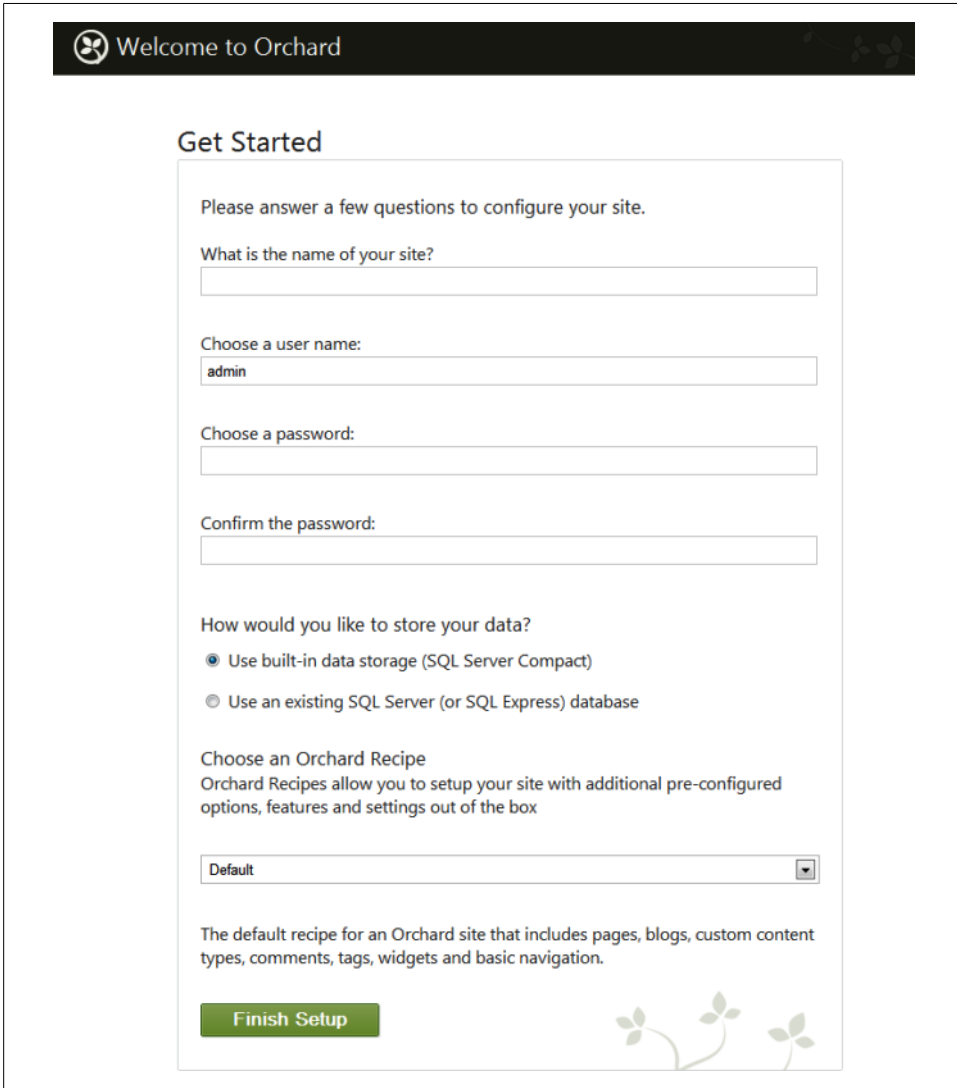
The very first run of an Orchard site will probably seem slow as some of the dynamic components are compiled and cached.

Configuring Orchard for the First Time

The Get Started page ([Figure 1-1](#)) requires you to provide a few quick details to get your Orchard site up and running. The first three questions ask you to name your site and provide an administrative username and password. These values should be straightforward. You then have the option to select a SQL Server Compact or full SQL Server (or SQL Express) database. Typically you’ll choose SQL Server Compact for development only, but a low traffic site might make do with a SQL Server Compact database.

Create your new site with the name “Daisy’s Gone.” Revisit the Preface to remind yourself of the purpose of our site. You should also change the default admin name to something identifiable, like “jzablocki.” Make this change so that when content, such

as blog posts or event listings, is added to the site, the name of the author isn't "admin." We'll also opt to use SQL Server Compact for storage since we're setting up this site for development purposes.



Welcome to Orchard

Get Started

Please answer a few questions to configure your site.

What is the name of your site?

Choose a user name:

Choose a password:

Confirm the password:

How would you like to store your data?

☒ Use built-in data storage (SQL Server Compact)

☐ Use an existing SQL Server (or SQL Express) database

Choose an Orchard Recipe

Orchard Recipes allow you to setup your site with additional pre-configured options, features and settings out of the box

Default

The default recipe for an Orchard site that includes pages, blogs, custom content types, comments, tags, widgets and basic navigation.

Finish Setup

Figure 1-1. The Get Started page of Orchard CMS



An advantage of using SQL Server Compact is that you're easily able to reset an Orchard site back to its starting state (the Get Started page) by deleting the App_Data folder containing the site's local database. On my system, this path is `C:\dev\Orchard\src\Orchard.Web\App_Data\Sites`. Resetting the site is useful when you want to ensure a clean slate for new site development.

The final question on the Get Started page involves selecting an Orchard recipe. Recipes are preset site configurations and are simply XML files stored in a well-known path. It's also possible to create your own. The default Orchard setup includes three:

Default

Includes frequently used Orchard features. This recipe is suitable as a starting point for most sites.

Blog

Creates an Orchard site to be used as a personal blog.

Core

Configures Orchard to have only the minimum required features. This recipe is much better for Orchard development than for site creation.

Though the “Core” recipe is suitable for module development, we're going to use “Default” since we're going to be building a fully functional site. After you've answered these five questions, click the “Finish Setup” button. You should see a modal progress bar appear with the message “Cooking Orchard Recipe” (see [Figure 1-2](#)).

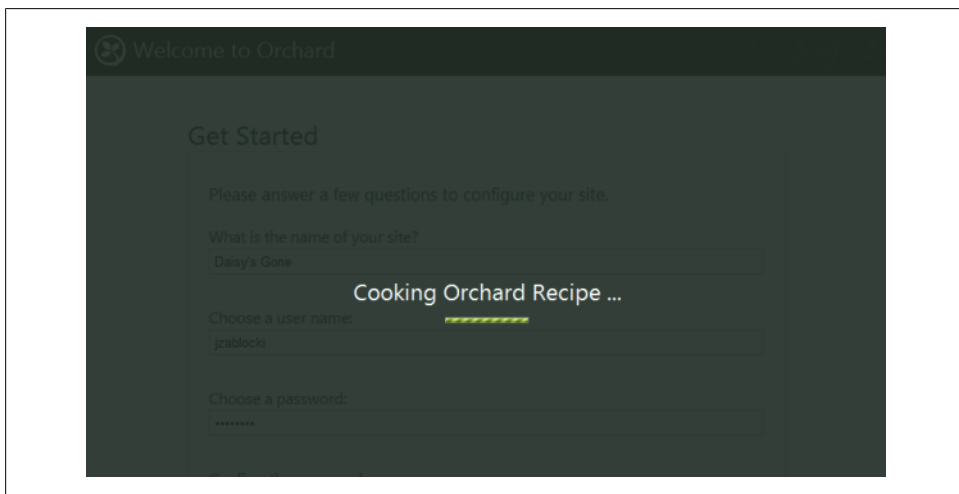


Figure 1-2. Cooking an Orchard recipe

After the recipe has finished cooking, you'll be redirected to your site's home page (Figure 1-3). You'll see the site name that we entered on the "Get Started" page in the upper-left corner and a series of text sections, each with a title and some content.

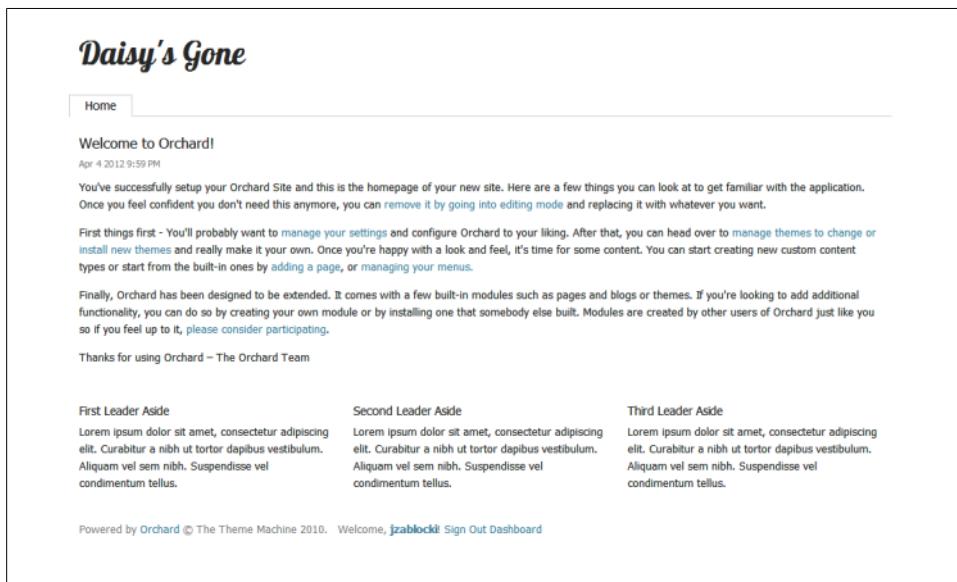


Figure 1-3. The default Orchard home page

The Orchard Dashboard

At the very bottom of the home page, you'll see the default template's footer with a few links. The last link is labeled "Dashboard." Click that link to get to the admin pages for your site. When you click through, you'll land on the admin home page. This page simply has links to finding more information on Orchard. The functionality for managing content is accessible via the menu on the left side of the screen. We're going to explore the admin pages in more detail in the chapters ahead. For now, we'll just take a quick tour of the basics.

Creating Content

Start by clicking the "Content" link on the menu. This is the admin page where you'll manage content, such as your site's home page (Figure 1-4). There are three tabs on this page, which are described here. (Don't worry about the new terms in this chapter; we're going to revisit them over the next few chapters.)

Content Items

Single piece of content, such as a blog post or an event listing.

Content Types

Blueprints for content items. Defines the attributes and behavior of the content items that you'll create using the admin tool.

Content Parts

Reusable pieces of functionality that may be used to compose content types.

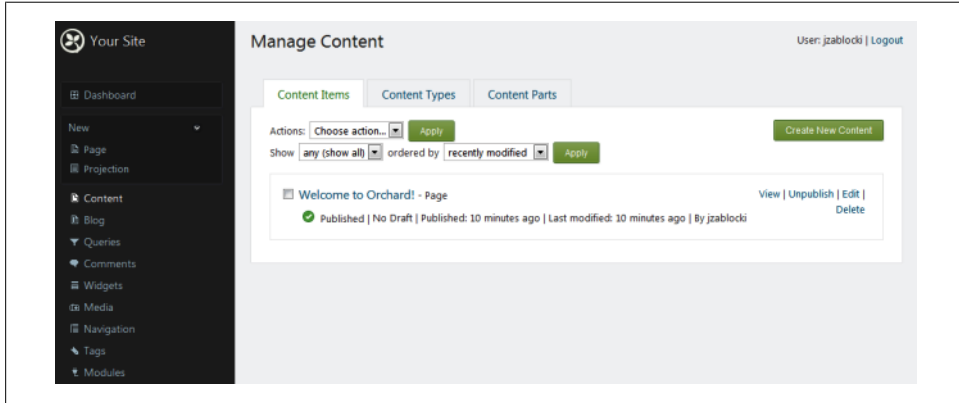


Figure 1-4. The content management admin page

To get a sense of how content is created using these tools, click the Create New Content button in the upper-right corner of the page. You'll be taken to a page where you are shown two links: Page and Projection. Select Page. You'll then arrive at the page shown in [Figure 1-5](#).

Right now, our site has only the home page. We're going to use the New Page form to add an About page. For the title, enter "About Daisy's Gone." Then add some content describing the band. We'll also check the option to "Show on main menu," which will add a tab to the site's main navigation. Checking this option enables an additional textbox for setting the menu text. Enter "About."

You'll then have the option to set a "Created On" timestamp. We'll let this default to the current date and time. We'll also choose to "Publish Now" rather than select a future date via the "Publish Later" option. The Save button will allow you to save your work without having it appear on the site.

After you publish your page, click the Your Site link in the upper-left corner of the admin pages. You'll be taken to the home page. Notice the new "About" link that has been added to the menu. Your new page is accessible via that tab ([Figure 1-6](#)).

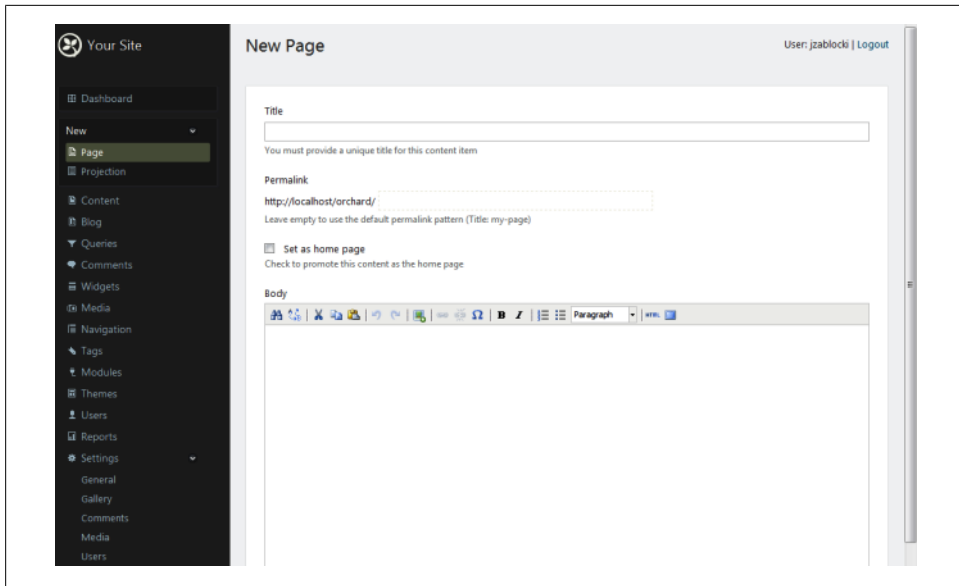


Figure 1-5. The new page admin screen

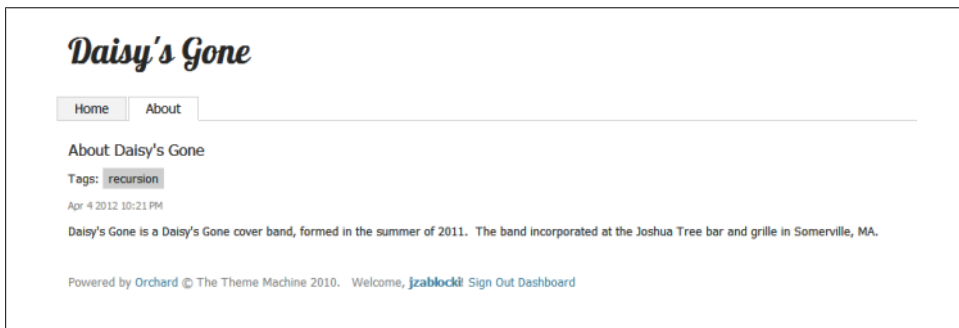


Figure 1-6. The new About page



When working with a CMS such as Orchard, it's useful to keep one browser tab open to the Dashboard and a second open to your site. This practice will allow you to Ctrl+Tab between tabs to see your changes on the live site without navigating away from your current admin location.

Adding Widgets

Widgets are UI components that may be added to some or all pages of an Orchard site. Click on the “Widgets” menu item to manage these components. On this page, you’ll see a listing of layers and zones.

Layer

Set of rules that define when widgets will appear.

Zone

A placeholder into which widgets may be inserted.

In [Chapter 4](#), we’ll learn about zones, layers, and layouts in detail. For now, know that a theme defines a site’s layout and a layout defines which zones are available. The listing of zones you see on the “Widgets” admin page were defined in the default theme, which is called “The Theme Machine.” Layers define rules for which zones will be active.

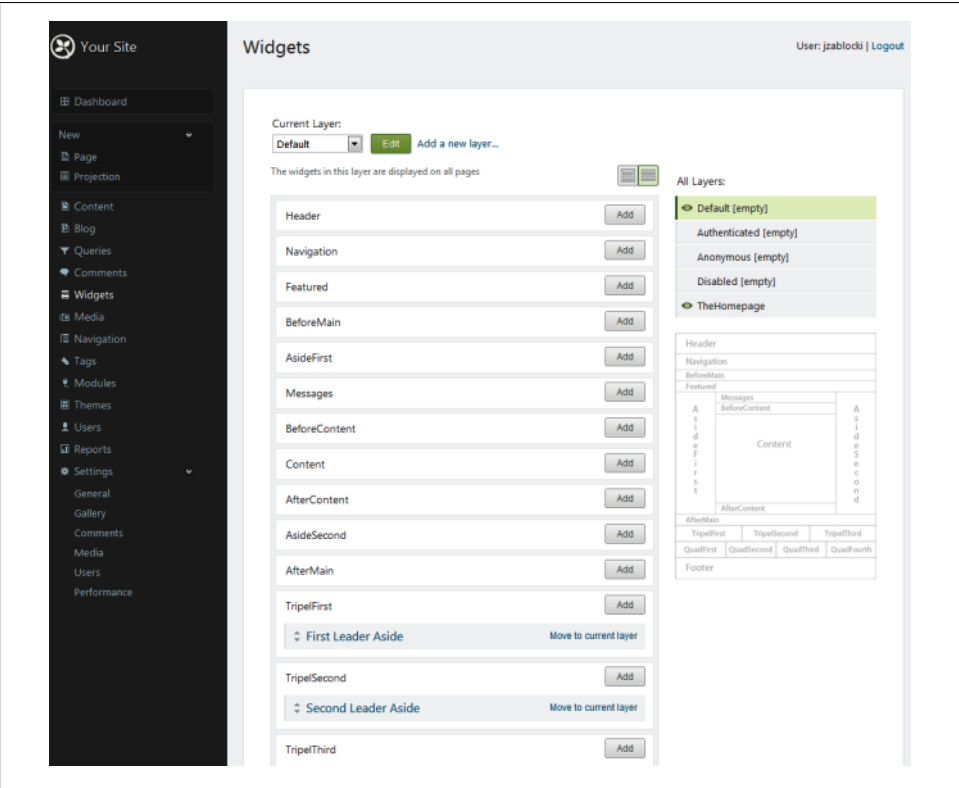


Figure 1-7. Managing widgets

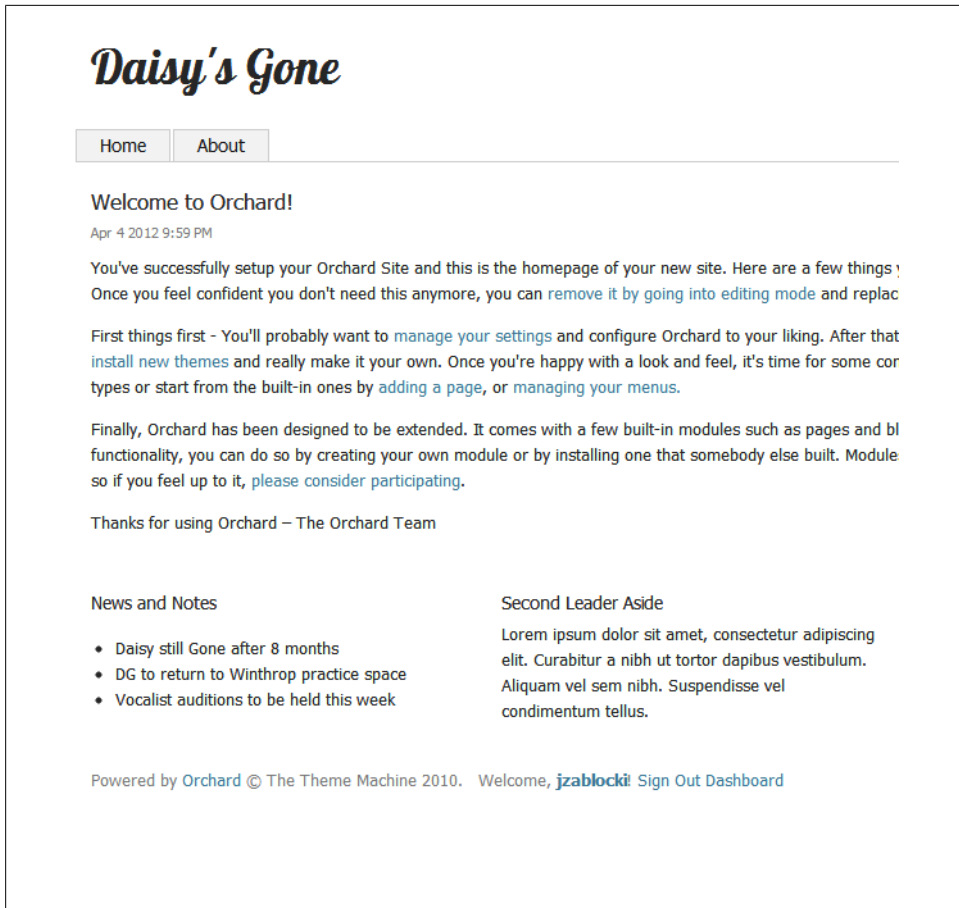


Figure 1-8. HTML widget updates

Notice that the zones `TripelFirst`, `TripelSecond`, and `TripelThird` have links with the text “First Leader Aside,” “Second Leader Aside,” and “Third Leader Aside,” respectively. If you click over to your site’s home page, you’ll see that these are headings on the three zones at the bottom of the page (Figure 1-3). Click through “First Leader Aside” and you’ll find yourself on a page where you can edit the HTML page of this widget. Appropriately, this is called an HTML Widget. If you click the “Add” button next to any zone, you’ll see “HTML Widget” listed as an option for each.

As we learn to create and manage content, we’ll go over these options in more detail. For now we’ll take a quick look at changing widgets. Click on the HTML widget under “TripelFirst.” Enter the title “News and Notes.” Click on the bulleted list in the HTML editor and enter some news or some notes. Save the changes and visit your site’s home page again. You should see your new content reflected on the home page (Figure 1-8).



You might be wondering why the third set of zones are named “Tripel” and not “Triple.” It’s not a misspelling. The zones are a sort of inside joke and a tribute to Tripels, which are a particular style of high alcohol, strong pale ales.

Orchard Modules

A great deal of functionality is available for an Orchard site by downloading and installing Orchard modules. Clicking on the “Modules” admin menu option at first reveals the set of installed modules under the “Features” tab. The “Default” recipe we chose to setup our site has influenced this listing, as recipes can instruct the Orchard software to include different modules. We’ll explore the “Features” tab in more detail when we write our first custom module.

We want visitors to know where to find the band on rehearsal nights, so we’re going to include an embedded Bing map that will display a push pin at this location. We’ll place this map on the home page only. Bing Maps aren’t out of the box Orchard functionality, so we’re either going to have to code a solution or find one that’s already coded for us. Fortunately, the work has already been done.

Click on the “Gallery” tab; you’ll see a listing of modules available for download and installation. Search for “Bing Maps.” There will be a few results. The one we want is a module named “Bing.Maps” that was authored by Orchard project lead Bertrand Le Roy. Click “Install” to download and install the module. After it is installed, you’ll be prompted to enable it. After you enable it, you will be able to add it to a zone.



At the time of this writing, there is a bug that might lead to a “Package installation failed” error when installing a theme or module from the Gallery. If you get that message along with a note about permissions errors, click Settings→Gallery and set the gallery feed URL to <http://packages.orchardproject.net/FeedService.svc/>.

Return to the “Widgets” admin page. Change the “Current Layer” drop-down box from “Default” to “TheHomepage.” Locate the zone “TripelThird.” Click “Remove” to delete the existing HTML Widget from that zone. Click “Add” and select the “Bing Map Widget” from the list of possible widgets. Enter the title “Where Daisy’s Gone Practices”; a latitude and longitude of 42.375,-70.983; and width and height of 300×200. Set the “Zoom” level to 10 and click “Save.” If you navigate to the site home page, you should now see a Bing Maps widget where before there was placeholder text (Figure 1-9).

s the homepage of your new site. Here are a few things you can look at to get familiar with the application. you can [remove it by going into editing mode](#) and replacing it with whatever you want.

[in settings](#) and configure Orchard to your liking. After that, you can head over to [manage themes to change or](#) you're happy with a look and feel, it's time for some content. You can start creating new custom content , or [managing your menus](#).

comes with a few built-in modules such as pages and blogs or themes. If you're looking to add additional module or by installing one that somebody else built. Modules are created by other users of Orchard just like you

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.

Welcome, [jzablocki](#) [Sign Out](#) [Dashboard](#)

Where Daisy's Gone Practices

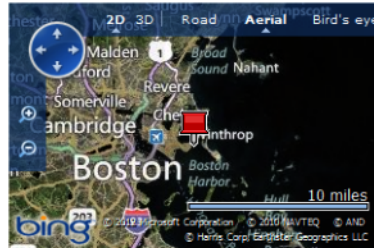


Figure 1-9. A Bing Maps widget

Orchard Security

We've seen *how* to manage content. Now we'll look briefly at *who* can manage content. The "Users" admin feature contains forms for managing users and roles. Creating a user is a relatively straightforward process. Click "Add New User" and enter a user-name, email, and password. You are also able to add users to roles. There are a number of predefined roles that are part of a standard Orchard installation.

Administrator

Has full control over a site and its content

Editor

Can edit and publish, but not create content

Moderator

Can validate user-generated content, such as comments

Author

Can create and publish content

Contributor

Can write, but might not be able to publish content

There are also roles defined for Anonymous and Authenticated users, but membership is based on whether a user is logged into the site. These roles are not assignable. New roles with custom permissions may also be created. The “Add a role” button is accessible via the “Roles” tab. When creating a new role, you select the permissions you want to be available for users in this role.

We’re going to create two users in addition to our administrator, who in my case is named “jzablocki.” Click “Add new user” and enter the username “gcocca.” Check the “Editor” and “Author” roles and save the new user. Create another new user with the username “nsilvia” and check the role “Moderator.” Save this user. Greg—gcocca—will now be able to create, edit and publish content. Nino—nsylvia—will be able to moderate user generated comments.

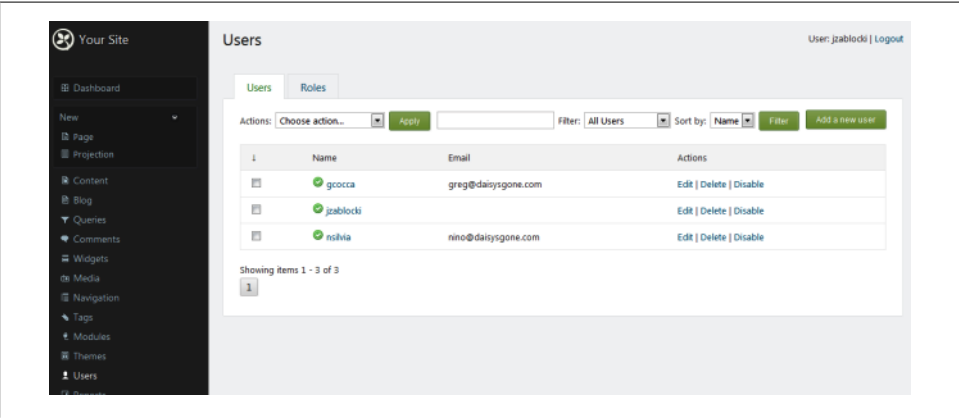


Figure 1-10. Orchard user management

One additional user setting to be aware of is under the “Settings” section of the admin menu. Click Settings→Users and you’ll find additional user management settings. You can allow users to self-register, reset their own passwords, verify email addresses, and go through an approval process before having authenticated access to the site. We’ll stick with unauthenticated comments for our content and leave all options unchecked.

Summary

We’ve now seen how to get an Orchard site up and running from the Orchard source. We’ve created and edited content. We’ve downloaded a module from the Orchard Gallery and added it to our home page. A simple site could be built and maintained using what little we’ve learned so far. Our site won’t be too complicated, but it will require some basic customization. In the chapters ahead, we’re going to learn how to unleash the power and extensibility of Orchard.

Creating and Managing Content

In [Chapter 1](#), we briefly explored how to work with the content management tools of Orchard. In this chapter, we'll dig a little deeper as we build out a baseline to our site, which will then be extended with custom modules and a custom theme in later chapters. This baseline will include a handful of simple pages as described here. These pages will provide enough variety that we'll be able to showcase Orchard's content management capabilities.

Home Page

The same home page from the previous chapter

About

The About page from the previous chapter

Contact

A page with a form to use to contact the band

Events

A page listing upcoming band-related events

Bios

A page with band biographies

Blog

A standard blog

Gallery

A page with images, videos, and audio

Bio Items

In its simplest form, a biography on the Daisy's Gone Orchard site is just a web page. It could easily be added to the site using the same approach we used to add the "About" page in the previous chapter. However, we're going to have a bio page for each band member. Though we could cram all bios into a single page, it would be more readable

to have a unique page for each musician (and drummer). That requirement means that I'll effectively be creating the same page at least three times, just with different content.



An old joke asks, “What do you call the guy who hangs out with the musicians in the band?” and answers “The drummer.”

The process of creating a bio page is somewhat analogous to that of creating a blog post. Each blog post has common attributes, such as a title, a body, tags, and comments. When we manage blogs in a CMS, we typically have a special form for creating posts; we don't just create a new page and lay it out a certain way. Similarly, each bio will have a musician's name, a date of birth, and a body. We want to create bio pages just as we create blog posts—with a special admin form.

Content Types

As we saw in the previous chapter, the “Content” admin menu has three tabs. “Content Items” lists the two pages we've created so far. “Content Types” lists the different types of content that we can create on our Orchard site. “Content Parts” lists some reusable components that may be added to content types. Click into the “Content Types” tab and click the “Create new type” button.

The new content type is going to be a **Bio** type. Create the new type with this name and leave the content type id as “Bio,” which is what Orchard automatically set the value to when we typed in the name. Then click the “Create” button.

Content Parts

From the screen that followed, we're going to add content parts to the **Bio** content type. By adding these parts, **Bio** content will automatically gain behavior and attributes. To the **Bio** type, add the content parts that follow by checking the appropriate boxes and clicking “Save.”

Autoroute

Makes a bio item accessible by a URL matching a defined route pattern

Body

Adds an HTML content section to the content item

Containable

Allows a content item to be contained within another content type

Menu

Allows a content item to appear on the main navigation

Publish Later

Allows a content item to be edited and saved, but activated at a future date

Title

Allows a page title to be captured

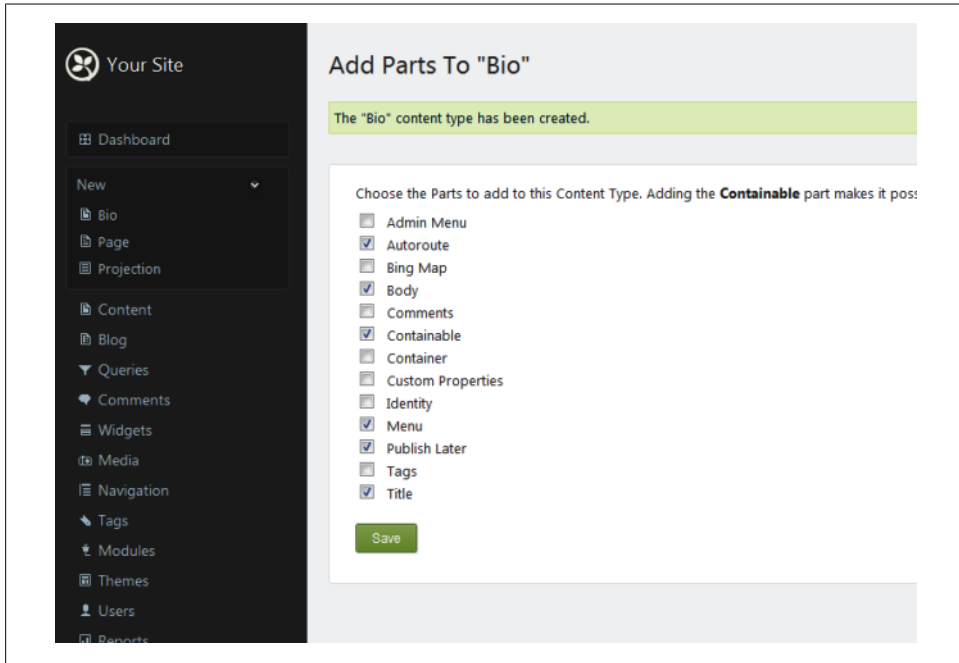


Figure 2-1. Adding content parts to a content type

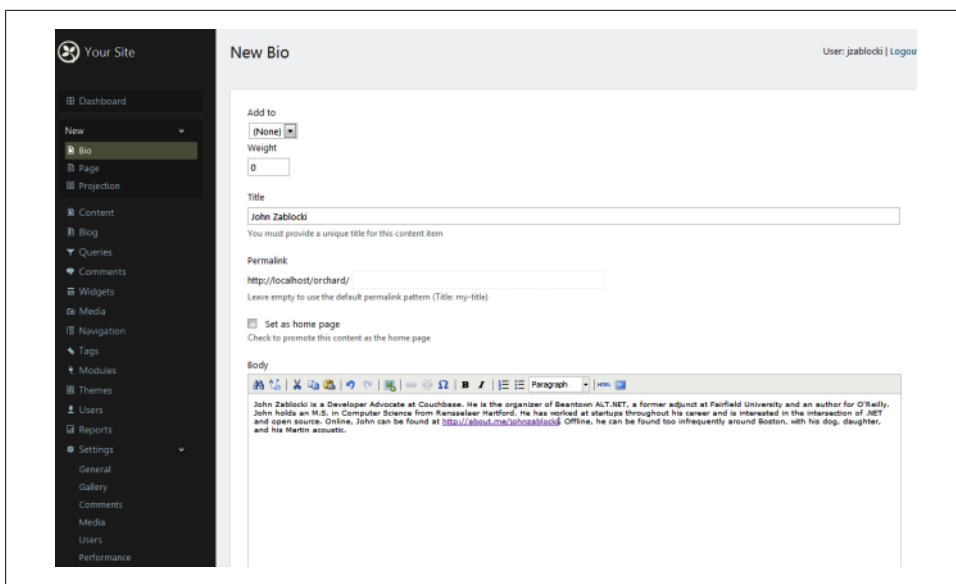
After saving parts to the content type, you'll be taken to a page where you can choose to add fields or additional parts. For now, just click "Save" at the bottom of the page. We'll revisit these options later in this chapter. After saving, we now have an option under the "New" menu item called "Bio."

Clicking this option opens a new form where we'll create new content items for our **Bio** content type. At this point, our **Bio** type has only title and body fields where we can enter bio content. We'll add to these fields shortly. For now, enter the name of the person whose bio is being created for the title and add some biographical content to the body.

Before clicking "Publish Now," we'll want to make note of the "Permalink" field that's displayed after the "Title" field. The permalink contains the URL by which our **Bio** item may be reached. Recall that we added the **Autoroute** content part to our content type, which will take care of mapping the permalink to the right content item.

We didn't change the defaults for **Autoroute**, so the title value will be used to complete the URL for this item. With the title "John Zablocki," our **Bio** item may be reached at <http://localhost/orchard/john-zablocki>. Now you can click "Publish Now." After publishing, browse to the item at the permalink URL.

Note that we also included the **Menu** part and could have chosen to add the bio page to our main menu. We’re going to create a few **Bio** items, so we don’t want to inflate the navigation. But having the part gives us that option in the future.



The screenshot shows the 'New Bio' admin page. On the left is a dark sidebar with a 'Your Site' menu. The 'New' section is expanded, and 'Bio' is selected. The main area has a header 'New Bio' and a user profile 'User: jzablodski | Logout'. The form contains the following fields:

- Add to:** A dropdown menu with '(None)' selected.
- Weight:** A text input field with '0' entered.
- Title:** A text input field with 'John Zablodski' entered. Below it is a note: 'You must provide a unique title for this content item.'
- Permalink:** A text input field with 'http://localhost/orchard/'. Below it is a note: 'Leave empty to use the default permalink pattern (Title: my-title)'. There is also a checkbox labeled 'Set as home page' with the text 'Check to promote this content as the home page'.
- Body:** A rich text editor with a paragraph of text about John Zablodski.

Figure 2-2. The New Bio admin page

Content Fields

At this point, the **Bio** template is a little thin in that it’s lacking fields for capturing information such as instruments played and bio pictures. Let’s first add the field for instruments played by editing the **Bio** template. Click the “Content” menu option and then click the “Content Types” tab. Clicking the “Edit” option on the “Bio” row opens up the “Edit Content Type” page.

Clicking “Add Field” will open a form for creating a new field for the **Bio** template. Several types of fields are available by default. A text field is sufficient for the instruments field, so select that value for “Field Type.” Set the “Display Name”—used for labels—of this field to “Instruments.” Let the “Technical Name”—used in code—default to “Instruments.” Click “Save.” Add two additional text fields with display names of “Birthplace” and “Lives In.” Accept the respective technical name defaults of “Birthplace” and “Lives In.”

Clicking New→Bio reveals that the **Bio** template now has the three additional fields. Similarly, clicking Content→Edit on the **Bio** content item also shows that these new fields are available for existing **Bio** content items. Enter “Guitar and Vocals” under “Instruments”; “Wethersfield, Connecticut” under “Birthplace”; and “Cambridge,

Massachusetts” under “Lives In.” After clicking “Publish Now” return to the page at the permalink we used earlier to view the first version of our **Bio** item.



Remember that “Publish Now,” not “Save” will make a change visible.

The screenshot shows a web form for defining a 'Bio' content type. It has a vertical sidebar on the left. The main form area contains several sections: 'Instruments' with a text input 'Guitar and Vocals'; 'Birthplace' with a text input 'Wethersfield, Connecticut'; 'Lives In' with a text input 'Cambridge, Massachusetts'; a checkbox labeled 'Show on main menu'; and 'Owner' with a text input 'jzablocki'. At the bottom of the form are five buttons: 'Save' (green), 'Publish Now' (green), 'Date' (light gray), 'Time' (light gray), and 'Publish Later' (green).

Figure 2-3. New fields on the Bio type definition

To add the field for uploading a profile image, an extra step is required. Since a text field obviously isn’t appropriate for an image, we’ll instead need to find a field with the functionality for uploading and inserting an image into our content item. However, Orchard doesn’t provide this field type by default.

Fortunately, the Orchard Gallery offers just such a field. Clicking back to “Modules” and then the “Gallery” tab, we’ll add the image field by first searching for “image field.” From the results, select the **Image Field** module by Sebastien Ros (of the Orchard team). If you find multiple versions, install the latest version (1.1.3 as of this writing). After installing and enabling this module, we can now use it in our bio template.

Returning back to the “Content Type” tab in the “Content” menu, again edit the **Bio** type. Click “Add Field” and select the new “Image Field” option. Set the display name to “Headshot,” accept the default technical name, and click “Save.” Now when we click back on “Content” and then click to edit the existing **Bio** item, we’ll see the addition of the new image field. Adding a new image simply involves clicking “Choose File,” selecting a file from your PC, and again publishing the page.



Orchard does include a “Media Picker” field. This field does support uploading new images to content items. However, it won’t automatically display the image on the published item.

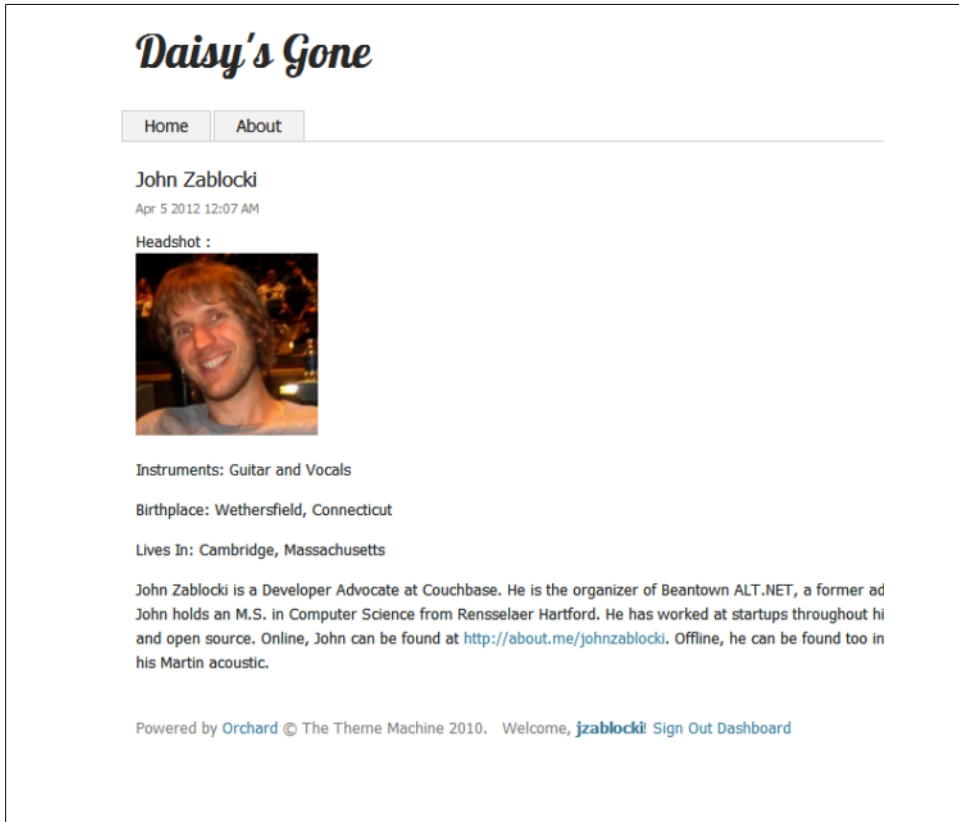


Figure 2-4. A bio page with a headshot

Projections and Queries

As we’ve seen, we haven’t actually made our first bio page accessible from the site via a link. To do so, we have a couple of options. The simplest case would be to create a standard page (via New→Page) and to add links or a summary section for each bio page that we create. While this approach certainly would work, it has a flaw. Each time we create a bio, we’d have to go update this page and add the new bio links and summary section. Fortunately—as you might have guessed—Orchard has a better approach.

A **Projection** is a content type that will display the results of a query in either grid or list form. We won’t actually write a *SQL* query to get the data for our projections.

Instead, we'll use the dashboard to select a filter, sort criteria, and a layout to compose an Orchard query. The items that match the conditions of our filter will be displayed in the projection page that we create.

Since our projection page will be uninteresting with only a single bio, let's take a quick detour to create two more **Bio** items. Return to New→Bio and create an item for drummer Greg Cocca and guitarist Nino Silvia. The specific content for these items isn't as important as having content.

We'll start by creating a new query. In the admin menu on the dashboard, click "Queries." On the "Manage Queries" page, click "Add new query." Set the title to "All Bios" and save the query. When you're redirected to the "Manage Queries" page, click "All Bios" and you'll see three options for our query. Click "Add a new filter" to set the data source for our new query.

Filters

We have several filter options from which to choose. For example, we could select "Body Part Text" to show only **Bio** items that have certain content (which we'd specify). We could select content with a certain title value or creation date. For our purposes though, keeping things simple is sufficient.

We'll just include all items that are **Bio** types. Under the "Content" section (of the filter page, not admin menu) click "Content Types" and select "Bio" from the next screen and save the form. If you click "Preview" after saving, you'll see a preview page with the single **Bio** we've created so far.

Sort Criteria

Next, on the "Edit Query" page, click "Add a new Sort Criteria." If you clicked to preview, just click "Back" in your browser as there's no navigation out of the preview page. We'll sort our query by name, which in our case is found in the **Title** part. Under "Title Part Record" click "Title Part Title" and select ascending from the next screen and save.

Layouts

The Layout option includes options for rendering our **Bio** items in a grid or an HTML list. In either case, we have a great deal of flexibility for managing the layout of these items. In fact, we are able to create multiple layouts so that we can have different views for the same query. Since we won't be displaying our bio list in more than one place, a single layout will be sufficient.

Click "Add a new layout" and select "Grid." The HTML List offers similar options, but the grid has a little more flexibility, so we'll choose that. The description section is used

to disambiguate our layout when we're creating projection pages. We're creating only the one layout, so we can leave that blank. The "Display Mode" allows us to choose from all item values or individual properties. Select "Content" to accept the default rendering.

Leave the alignment as "Horizontal" and set the "Number of Columns" to 1 (we'll just stack our **Bio** items). If we wanted to be able to use CSS or JavaScript with these items, we could specify an ID for the grid, along with CSS class names for the grid and rows. We won't be doing anything with the grid other than rendering it as is, so we can leave those fields blank for now. Save the layout.

Now that we have a query, we are able to create our actual projection page. Click New→Projection from the dashboard menu. Add a title of "Bios" and let the Permalink default to the title. Under "For Query," select the new query we just created—"All Bios (1 columns grid)."

Leave the other defaults in place, except for "Show on main menu." Check that option, and specify the "Menu Text" of "Bios." We want "Bios" to appear alongside "Home" and "About." If you save the projection and return to the live site, you'll see the new "Bios" option on the menu. Click on it to see the queried item.

Event Items

Adding events to the Daisy's Gone site will require a process similar to adding bio pages. Event will be a new content type that will be contained in a page that has a list of events (by way of a query and projection).

Content Types and Fields Continued

Let's start by navigating to Content→Content Types and selecting "Create new type." Name the new type "Event," and like our **Bio** type, select the **Autoroute**, **Body**, **Containable**, **Menu**, **Publish Later**, and **Title** parts. Also include the **Comments** part to allow users to add comments to the events. Click "Save" on the confirmation after saving the parts.

Events will also need locations and dates. For location, add a text field named "Location" to the fields on the **Event** content type (click "Add Field"). For the event date, we'll add a new date time field with a display name of "Event Date." Accept the default technical name of "EventDate." Click "Save" to make our new content type available with the fields we just added.

When clicking New→Event, we can now see that the location and event date fields are present on our new content type. Create an event titled "Unplugged in Cambridge." Recall that this title will be used as part of the URL that will be used to navigate to this page. This is the default behavior of the **Autoroute** content part.

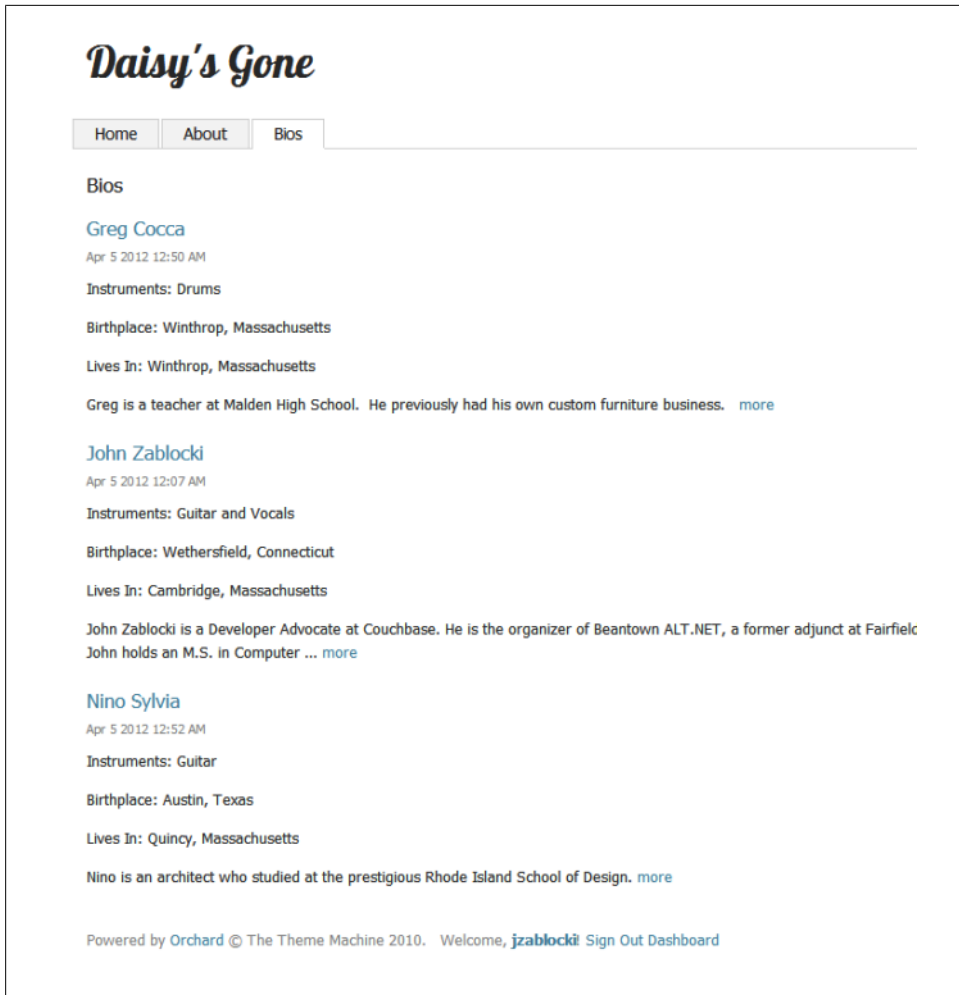


Figure 2-5. The Bio projection page

To see the other possible URL routing patterns we could select, click Content→Content Types and then click “Edit” on the Event type listing. Click on the greater than sign (>) next to the Autoroute part. Notice the “Default” pattern in the list. Its value is “{Content.Slug}.” This pattern will simply create a slug (URL pattern) from the title you enter (basically, spaces are replaced by dashes).

We’re not going to deviate from this pattern in this book, but you should click on the box in the “Pattern” field to see some of the other options you could use to create a route for custom content types. Also note that the previously created bio pages that we navigated to by adding names to the root site URL also worked via Autoroute (i.e., <http://localhost/orchard/john-zablocki>).

Since we just abandoned the content type that we were creating, return to New→Event. Again, add the title “Unplugged in Cambridge.” Next add a brief description and the location “Cambridge, Massachusetts.” Also select a date for the event and check to allow comments to be added (this should be checked by default). Publishing the event makes it available to the site, but we still need to create a projection page before it will be visible via a menu item.

Projections and Queries Continued

Before we can create a new projection page for our events listing, we’re going to need to create a query (as we did for our **Bio** projection). Click “Queries,” then click “Add new query.” Enter the title “All Events” and save the query. Click into the query from the list that shows after saving. We’ll start this query simply by adding a new filter that contains all events. On the “Add a Filter” screen, click “Content Types” and select “Event” from the list that appears. After saving, click “Preview” to see that the query does in fact return our single **Event** item.

We’re going to want to limit the events to only those that are in the future. Click again to add a new filter, but this time select “Event Date:Value” under “Event Content Fields.” We want only events that are occurring today or in the future, so select “Is greater than” in the “Operator” list. Since our source date will change daily, select “An offset from the current time” and enter “-1” for a value and “Day” for a unit. This combination will limit the query to events that are greater than yesterday.

Since we’re going to want to see the next event at the top of the list, add sort criteria to sort on the “Event Date” field ascending. For the layout, add a simple one-column grid as we did when we created the layout for our **Bio** items. See the “All Bios” query earlier in this chapter for details. After adding the layout, save the query.

We haven’t used the “Properties” display mode for our queries. This mode is far more powerful than the “Content” mode we’ve been using. However, “Content” is sufficient for our needs.

Though we won’t cover “Properties” mode in any detail, it certainly merits a brief discussion as to when it would be useful. We would use “Properties” mode if we wanted to be able to customize everything from date formatting (for date time fields) to wrapping HTML elements around content item property values. You can also specify functionality, such as what to do when there are no results for a field and how to handle HTML tags and whitespace for a value. Most of the options you can choose from are straightforward.

Now that we’ve created the “All Events” query, we’re able to create a new projection page that will display a list of our **Event** pages. Navigate to New→Projection. Enter the title “Events” and leave the permalink field blank to use the default behavior, which again will use the title.

Select the “All Events (1 columns grid)” query and check the “Show on main menu” option. Set the “Menu Title” to “Events.” Otherwise, leave the defaults in place. Save the projection and return to the live site to see the Event listing on our new projection page. Click the event title to see the details with user comments enabled (Figure 2-6).

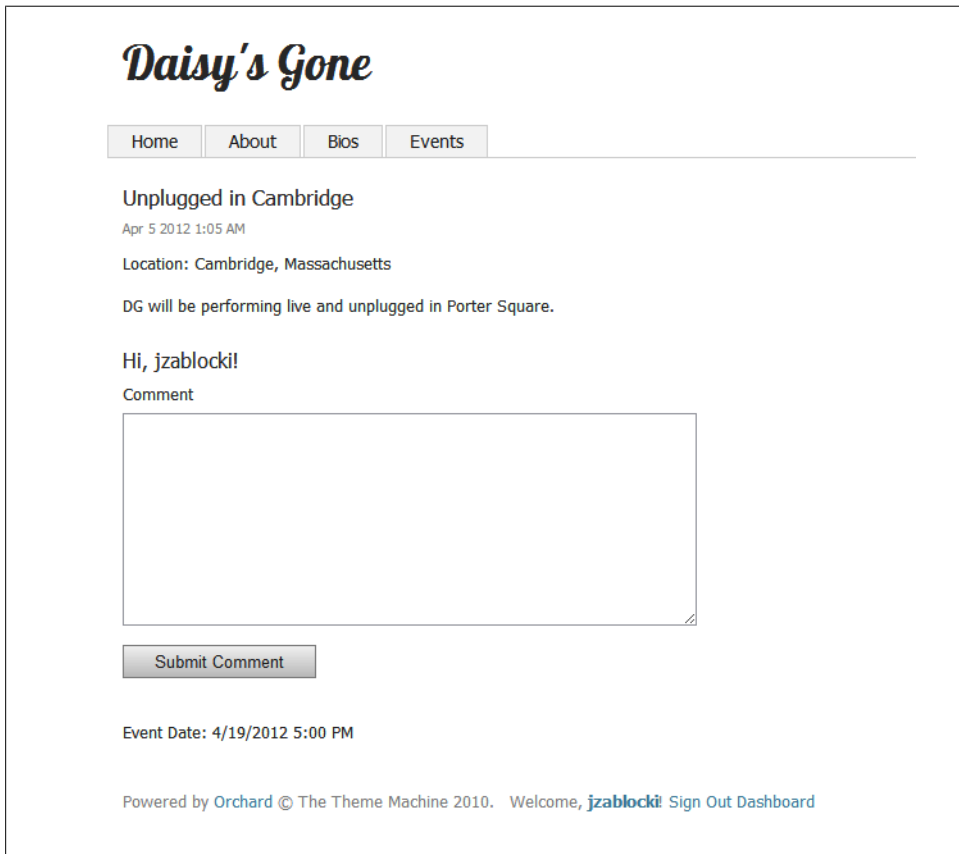


Figure 2-6. An event page with commenting

Daisy's Blog

Perhaps one of the most common use cases for a CMS such as Orchard is to create a blog. Blogging clearly helped WordPress to become what is arguably the most popular CMS on the market. For blog support, Orchard includes the `Orchard.Blogs` module, which provides standard blogging functionality, such as creating and maintaining posts and allowing authenticated comments.

Many blogs are built as standalone websites. Like the site we're building, these blogs have About and Contact pages. They also often include widgets for tag clouds and blog

rolls (links to other blogs). Creating such a standalone blog in Orchard is as simple as creating a new Orchard site and selecting the “Blog” recipe from the “Get Started” page. We’re instead going to include a blog as another section of our site.

The Blog Module

Click “Blog” to get to the “New Blog” form. Enter the title “Daisy’s Blog” and set the permalink to “blog.” If we had plans to create another blog for our site, we would leave the default permalink behavior of using the `Title` part’s value. Check the “Show on main menu” option and provide the “Menu text” value of “Blog.”

Save the form and revisit the site to see the new “Blog” tab has been added to the main navigation. After saving, you’ll also notice that the “Blog” admin menu has been updated to include new options. “Manage Blog” is where you’ll edit existing blog posts and find the link to update a blog’s properties. “New Post” and “New Blog” should be self-explanatory.

Before we create our first post, we’re going to update the permalink defaults used when blog posts are created. As was the case with the other content types we’ve seen, the default is simply to create a slug from the `Title` part of our post. We’re going to set our blog to use a traditional blog permalink format, which includes the date as pieces of the path. An example of such a URL is <http://daisysgone.com/blog/2012/4/1/welcome-to-daisys-blog>.

Navigate to Content→Content Type and click “Edit” in the row for the `Blog Post` type. Click the greater than symbol (“>”) next to “Autoroute.” Change the “Name” field from “Blog and Title” to “Date and Title”; change the pattern to “`{Content.Container}/{Content.Date.Format:yyyy/MM/dd}/{Content.Slug}`”; and change the description to “published-date/my-post.” The value “`Content.Container`” in this pattern will be replaced with “blog,” which was the permalink value we set for this blog.

It’s important to understand that the `Autoroute` part won’t recreate the permalink as you update your posts (or any other content types that use the `Autoroute` part). If you want that behavior, then you need to check the option to “Automatically regenerate when editing content,” which appears right above the “Patterns” section of the `Auto route` part settings for a particular content type.



Keep in mind that if you update a permalink that has been previously published, incoming links at the old URL will no longer be valid.

After saving these changes, click “New Post” and in the form that follows, enter the title “Welcome to Daisy’s Blog!” Since we just went through the effort of creating a new pattern for our permalink, we’ll obviously leave that new default in place. Enter some content in the “Body” field. We’ll leave the “Tags” field blank and allow

comments. Click “Publish Now” and then click the “Blog” tab on the site to see that this post is now listed ([Figure 2-7](#)). You can also see that its URL uses the pattern for its permalink.



Figure 2-7. A blog post

About Page

The About page is pretty straightforward and doesn't necessarily require any special Orchard functionality or custom content types. However, we'll take this opportunity to upgrade the HTML and text editor that comes standard with Orchard, which is the **TinyMce** editor.

TinyMce is probably sufficient for a most users, but I prefer **CKEditor**, which has richer editing capabilities. In the module gallery (Modules→Gallery) search for “editor” to find the “CKEditor” module, which we'll install to add this popular web-based HTML and text editor.

After enabling the **CKEditor** module, it's necessary to disable the standard HTML and text editor, which takes precedence. Disable the default editor by clicking Modules→Features, navigating to the “Input Editor” section, and disabling the **TinyMce** editor. If you don't disable the default editor, you'll still be able to edit, just not with our updated HTML editor.

After disabling **TinyMce**, we can click New→Page to see that the **CKEditor** is now the HTML and text editor for all content items that have a **Body** content part. Clicking Content→Content Items and then clicking on “About Daisy's Gone” also shows the new **CKEditor** is used for existing content items ([Figure 2-8](#)).

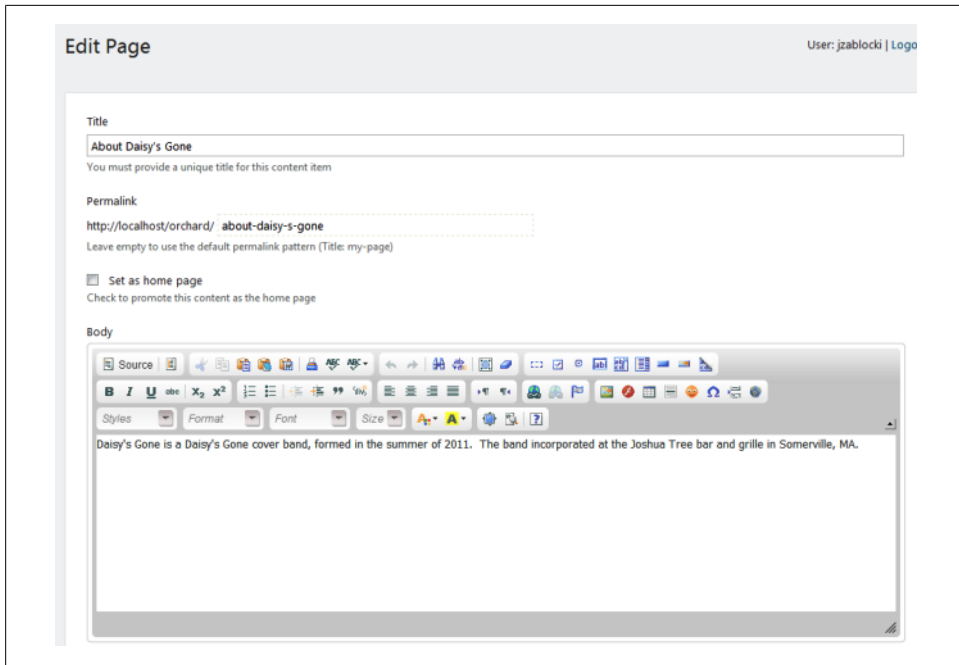


Figure 2-8. The About page with the CKEditor

Contact Page

The easiest way to add a contact page to an Orchard site is to install one of the contact modules from the Orchard gallery. A search for “contact” in the gallery will yield a few different options. We’ll install “Contact Form” by CyberStride. After enabling this module, a new content type of **Contact Page** is automatically added to our site. Clicking New→Contact Page will open a form for creating **Contact Page** items.

One field that is noticeably absent from this form is the ability to add a contact page to the site’s main menu. Fortunately, as we saw when creating the **Bio** and **Event** content types, this functionality is easy to add. By clicking Content→Content Type→Edit, we can add the content part.

Click “Add Parts” and check off the **Autoroute**, **Menu**, and **Title** parts and click “Save.” Return to New→Contact Page. Creating the contact page involves inputting some straightforward values for title and body. The “Email notification to” field needs to be set to the username of a valid user in the system. After saving the form and refreshing the site, we can click over to the “Contact” tab on the main navigation to see the new page (Figure 2-10).

The contact form includes fields for collecting name, email, and a message, all of which are required. If we fill out the form and click Submit, our message is sent. Back on the

New Contact Page User: jzablocki | Logou

Title

You must provide a unique title for this content item

Permalink

Leave empty to use the default permalink pattern (Title: my-title)

☐ **Set as home page**

Check to promote this content as the home page

Body

Figure 2-9. The new contact page form

admin pages, we can browse to “Contact Requests” and see a listing of attempted contact form submissions, both successful and failed (validation failures).

For an email to be sent when a contact request is made, we need to enable the **Email Messaging** module in the “Modules” admin page. When we enable the Email module, Orchard will display a yellow warning message that we need to configure SMTP settings. To set these, we can either click the link in the warning or navigate to Settings→Email. You’ll need a valid SMTP server through which you can relay messages in order to send email in Orchard.



If you get an error while clicking the “Contact Requests” page in the admin dashboard, you’ll need to edit the view file *index.cshtml* in the *Admin* views directory of the module. At the time of this writing, this page uses the `DateTime HtmlHelper` extension method, which is no longer present in the current version of Orchard. Simply remove the method call and set the line to `<td>@contact.ContactDateUtc.GetValueOrDefault()</td>`.

Daisy's Gone

[Home](#)[About](#)[Bios](#)[Events](#)[Blog](#)[Contact](#)

Contact Daisy's Gone

Apr 5 2012 1:45 AM

Contact Us

Name

Email

Phone

Current Website

Topic of Inquiry

Details

Send

Powered by Orchard © The Theme Machine 2010. Welcome, **jzablocki** Sign Out Dashboard

Figure 2-10. The Daisy's Gone contact page

Home Page

For the home page, we'll start by making a couple of minor content changes. Under "Content" we'll click on the listing currently titled "Welcome to Orchard!" and update its title to "Daisy's Online." Clicking "Publish Now" makes these changes available on the live site. Note that because this page has been set to the home page by way of the "Set as home page" field, we can get to this page at the root site URL <http://localhost/orchard>.

Zones and Layers

At this point, our home page hasn't been customized much beyond the small tweaks we made in the previous chapter. To add additional customization, we're now going to add a couple of new modules from the Gallery that we'll then add to the `TripelSecond` zone.

We want to give visitors the ability to "Like" the band on Facebook right from the home page. Searching for "Facebook" in the Gallery yields more than a handful of results. We're going to install the `Facebook.Like` module by David Fekke. After installing and enabling this module, we'll add it to the home page. We'll also add a Twitter follow button by installing the `Twitter Follow Button` module by Nicholas Mayne (search for "twitter").

To add the Facebook widget to the home page, click over to "Widgets" and set the current layer to "TheHomepage." Remove the "Second Leader Aside" widget from the `TripelSecond` zone. Then click "Add" on "TripelSecond" and select the `Facebook Like` widget. In the form for this widget, enter the title "Like Daisy's Gone on Facebook." Leave the other defaults and save.

Repeat these steps to add the `Twitter Follow` widget also to the `TripelSecond` zone. Use the title "Follow" and provide the Twitter username "@daisysgone". After saving both widgets, refresh the home page to see that both widgets have been added.



Figure 2-11. Social widgets on the home page

Clicking to other pages on the site, you'll notice that these widgets don't appear anywhere but the home page. The reason for this behavior is that the rule that determines whether to display these widgets is defined in the layer "TheHomepage." Clicking on "Widgets," selecting the layer "TheHomepage," and clicking "Edit" displays a form for editing the layer.

The "Layer Rule" field defines a **Boolean** expression that determines whether a widget is displayed. "TheHomepage" has a rule `url '~/'` that is true when the URL is the root site URL.

Layer rules have a terse syntax, using the logical operators **not**, **and**, and **or** along with the `url` and `authenticated` functions. To demonstrate how to use layers, we're going to add a new **Projection** widget to the zone **AsideSecond** on all pages except the events page.



On the Widgets admin page, you probably noticed the image that shows where zones are placed in the layout. Note that this image was created by the theme designer and will only be accurate if the CSS rules render zones in a way that matches the zones as shown in the image. If you modify the theme's CSS rules, there's no guarantee this image will remain accurate.

Click **Widgets**→**Add a new layer**. Name this layer "NotTheEventsPage". Add a rule `not url '~/events'` and then save. After saving, Orchard returns to the "Widgets" admin page with the new layer selected. Click "Add" on **AsideSecond** and select "Projection Widget." Enter a title of "Upcoming Events." Select the query "All Events (1 columns grid)", and limit the number of items to display to 3. After saving and refreshing the site, we can see that the new events projection widget appears on all pages except for the events page.

Gallery

For the gallery page, we're going to install and use the **Image Gallery** module by Gabriel Eduardo Chites de Mello. After installing and enabling this module, a new admin menu option is added. Click **Image Gallery**→**Add new Image Gallery**. Name the new gallery "Gallery Page." After saving, click the new gallery's name in the listing and then upload new images by clicking "Add Images" and using the upload form.

Again, click **Widgets**→**Add New Layer...** and name that layer "Gallery." Set the rule to `url '~/gallery'` so that we can add the **Image Gallery** widget to a page at this URL only. After saving the layer add the **Image Gallery** widget to the **AfterContent** zone. Leave the defaults, other than the required title.

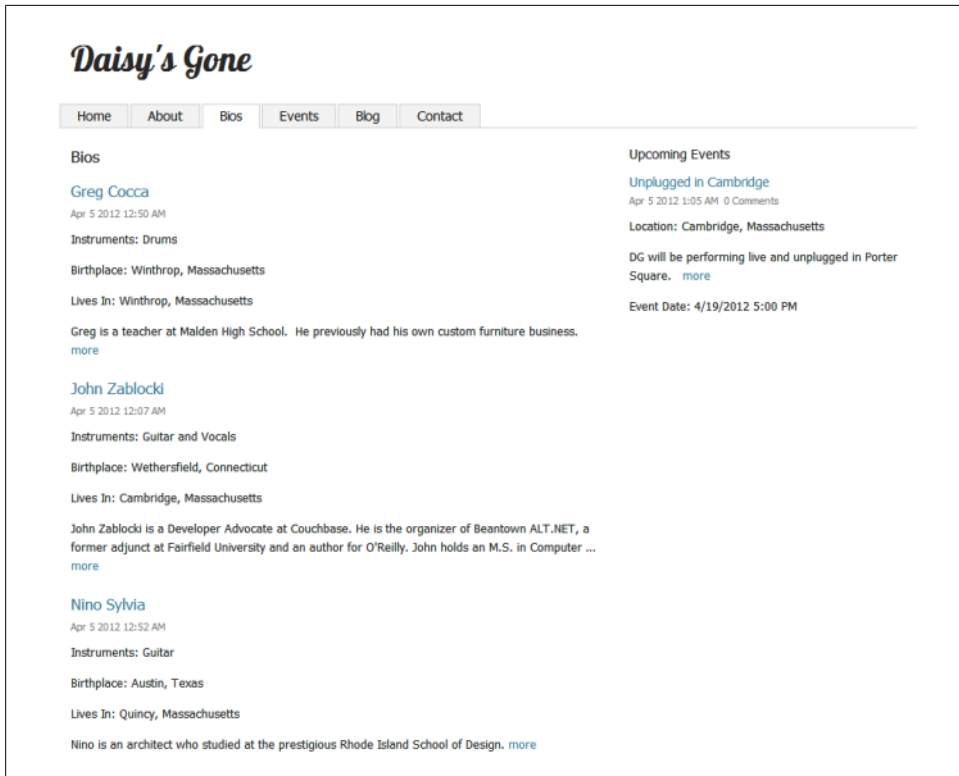


Figure 2-12. The new NotTheEvents layer

We'll next create the actual gallery page via New→Page. Add some content, add the title "Daisy's Gallery," set the permalink to "gallery" to match our rule, add the page to the main menu with the text "Gallery," and publish. After publishing, you can browse to this page by clicking the "Gallery" tab on the main menu (Figure 2-13).



I originally planned to name the "Gallery" page "Media." However, Media is already a special directory in Orchard and the URL route collided resulting in a 404 Not Found error by IIS.

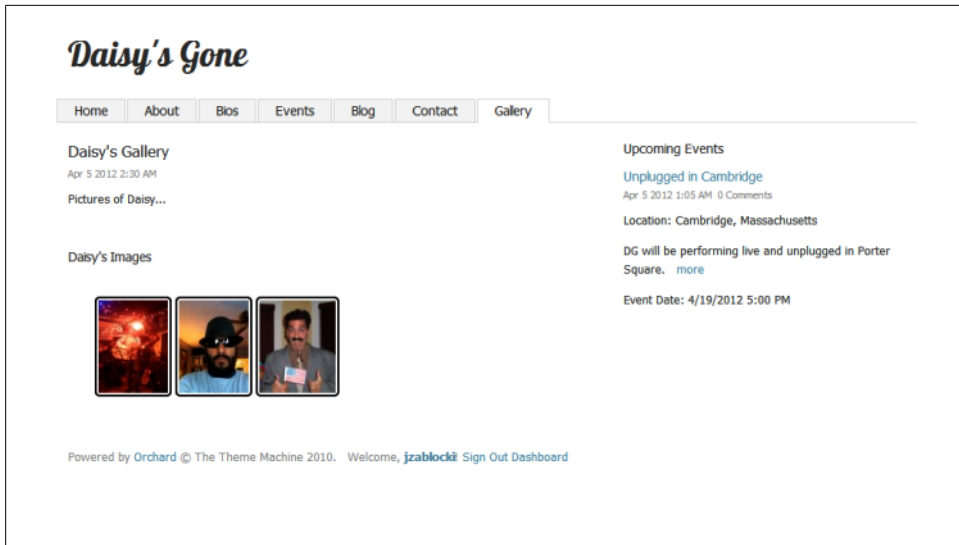


Figure 2-13. The Image Gallery

Summary

At this point, we've touched upon most of the core content management features of Orchard. We've seen how easy it can be to create new content types. We've also seen how we can augment existing types. In the next chapter, we're going to look at customizing how the content we create is displayed by Orchard.

Displaying Content

As we saw in the previous chapter, getting a basic Orchard site up and running is a pretty low friction process. Without having to write a single line of code, we were able to create new pages with dynamic content and reasonably advanced functionality. However, we also saw that the content we displayed has a default rendering, which we can't fully customize through the admin tools.

Fortunately, Orchard does provide functionality to allow us to alter this rendering. In fact, with minimal effort Orchard gives us great control over how we display content. Virtually every piece of data that gets rendered by Orchard (collectively known as shapes) may be customized by overriding its default template. In this chapter, we'll learn about the conventions and tools that make this process relatively straightforward.

Customizing Biography Content

As it exists now, the bio projection page simply takes each field in our **Bio** content type and renders its label and value one after the other with each on its own line (see [Figure 3-1](#)). It would be better if our content read a little more naturally, such as “John Zablocki was born in Wethersfield, Connecticut, and plays guitar for Daisy's Gone.” The label-value rendering might be sufficient for a product listing, but is less suited for our biography listing.

Ultimately, we'll create a template for rendering **Bio** content items in a projection page (or any other container). However, to demonstrate how Orchard alternates work, we'll start at a higher level and change the way all content is displayed when it's rendered in a container (a projection in our case).

Orchard content is rendered differently based on its usage. For example, when content items such as those created from our **Bio** type are added to a projection page, their summary display type is used. When a full content item is rendered (i.e., an individual biography), then the detail display type is used.

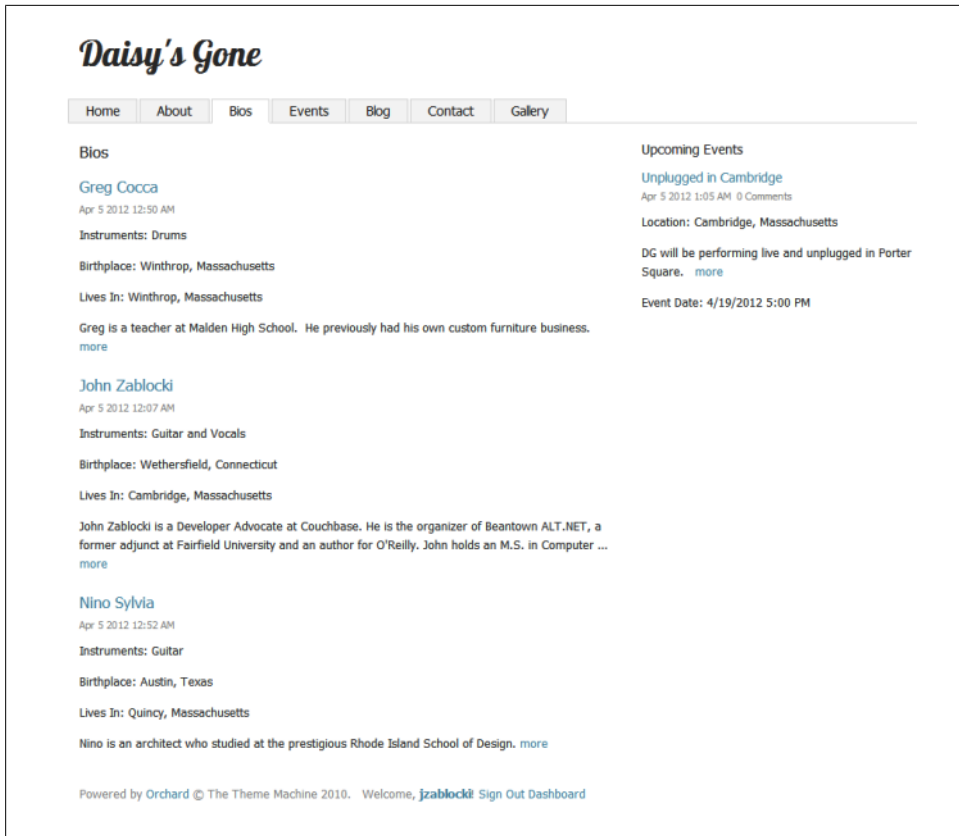


Figure 3-1. The default bio projection page

Content Templates

If you open up the **Themes** project in your Orchard solution (in the “Themes” solution folder) you’ll find a directory named *TheThemeMachine*, which is the default Orchard theme. The styles associated with this theme are kept intentionally simple, because this theme is generally used as a starting point for building new themes. In the next chapter, we’ll create a new theme from scratch, but for now we’re going to work within this one.

The first thing to do is expand the *Views* directory. There you’ll see six Razor (*.cshtml*) files already in this folder. These files are all used by the “TheThemeMachine” theme. We won’t explore these files now as we’ll discuss them in the next chapter.

Add a file to this directory, named *Content.Summary.cshtml*. If you clear the contents of this new Razor file, save it and then refresh the “Bios” projection page, what you’ll see (Figure 3-2) is that the **Bio** item summaries have disappeared. Also notice that the “Upcoming Events” are no longer appearing in the projection widget that we added to the **AsideSecond** zone.



The Themes project was not created using an ASP.NET MVC Visual Studio project template, therefore the usual MVC context menu short-cuts are absent from this project (i.e., “Add View”). Instead, I use Add→New Item and select “HTML Page” from the list of web templates. This action requires manually setting the *.cshtml* extension.

As you might suspect, Orchard found our new alternate template and used that file to render the summary display for our **Bio** items. You might also have realized that like our “Upcoming Events,” the **Event** and **Blog Post** items have also disappeared. You can verify this by clicking on the “Events” or “Blog” tabs. What this change is starting to demonstrate is how Orchard performs the task of “shape rendering.”



Figure 3-2. The effects of a blank *Content.Summary.cshtml* file

Alternate Templates

There are several file and directory naming conventions used by Orchard to determine which template should be used to render a particular piece of content. In our example, creating a view named *Content.Summary* instructed Orchard to use this template for all content shapes rendered with the summary display type. These views are known as “alternate templates.”

Again, content items in a projection page are rendered using the “summary” display type. We could edit our alternate template to add the header (title) and content (everything else) back in. To do so, we need to add only two lines of code. For illustrative purposes only, we’ll also change the background color of the content:

```
<div style="background-color:#c0c0c0;">
  @Display(Model.Header)
  @Display(Model.Content)
</div>
```

If you save the template and refresh the bio, event, and blog projection pages, you’ll see that the content is again being rendered as before, but now with a gray background. You’ll also see our “Upcoming Events” widget being rendered the same way.

To change this alternate template so that only Bio content items are affected by the customization, you simply have to rename the file to *Content-Bio.Summary.cshtml*. Orchard will parse the token after the hyphen and match it to a content type id. In this case “Bio” was the default content type id associated with the Bio content type when we created it. With this change, Event and Blog Post items are no longer rendered with a gray background within their respective containers.

You could also rename the file to *Content-17.Summary.cshtml*, where “17” is the unique identifier of a particular piece of content (my bio in my case). After renaming the file, if you refresh the “Bios” page you’ll see that only the bio summary specified in the filename has been grayed (Figure 3-3). This item level customization might be useful for calling out a particular item in a list, such as a special event.

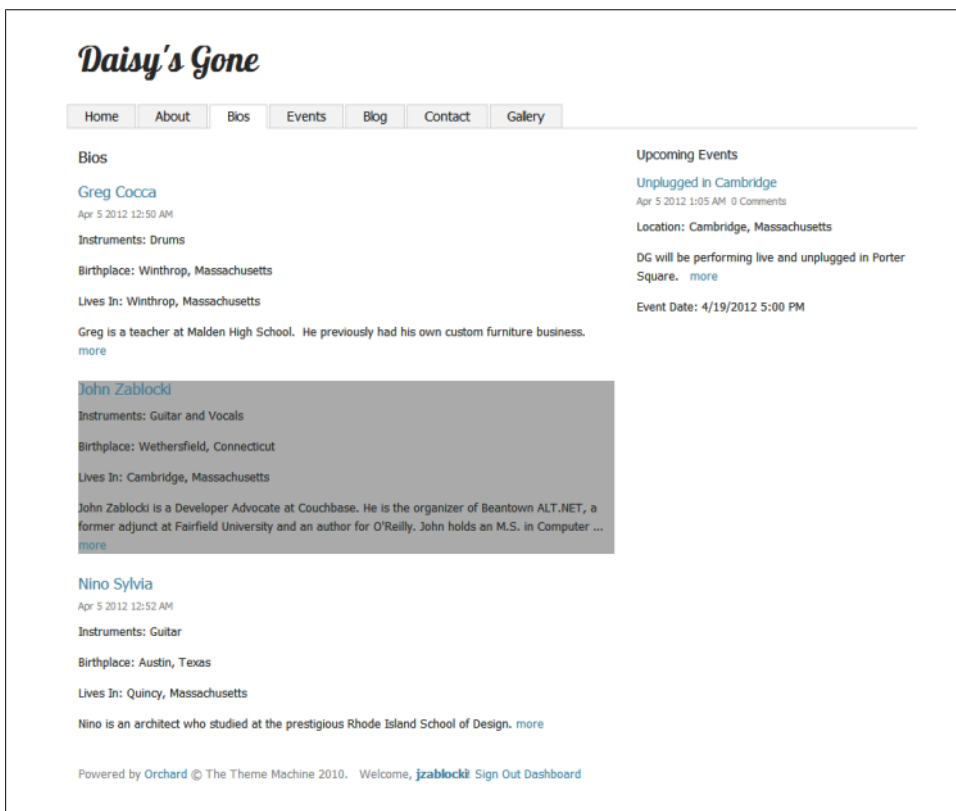


Figure 3-3. A template for a single content item

Let’s rename our alternate template back to *Content-Bio.Summary.cshtml* so that we can return to our original goal, which was to render our bio listings with more natural language. To achieve this goal, we’re going to create a simple template that will display values from our content type in an order that we define.

Our template first defines a simple Razor function `splitName` for convenience. We're going to take the shortcut of parsing our first name out of the title, as opposed to creating a separate field for this value in our type definition. We'll then check whether a profile picture has been set. If it has, we'll display it:

```
@{
    Func<string, string[]> splitName = (name) => name.Split(' ');
}
<div style="float:left;margin-bottom:10px;">

@{ var fileName = Model.ContentItem.Bio.Headshot.FileName;
  if (! string.IsNullOrEmpty(@fileName)) {
    
  }
}
<a href="@Model.ContentItem.AutoroutePart.DisplayAlias">
  @Model.ContentItem.TitlePart.Title
</a> plays
@Model.ContentItem.Bio.Instruments.Value.ToLower() for Daisy's Gone.
<p />
@splitName(Model.ContentItem.TitlePart.Title)[0] is originally from
@Model.ContentItem.Bio.Birthplace.Value and currently lives in
@Model.ContentItem.Bio.LivesIn.Value.
</div>
```

Next we display a link to the full bio page. We set the `href` property of the anchor tag by accessing the `Bio` content item's `Autoroute` content part, which has the URL slug for permalinks. We then construct a sentence that summarizes a little information about each musician. In this sentence, we'll access both the `Bio` type's fields and its `Title` part to fill in the template values.

The important thing to note is that if you want to reference fields from a content item in your templates, you'll need to use a dynamic expression that is generally of the form `Model.ContentItem.{ContentType}.{FieldName}.Value`. Other elements such as the title require first accessing its content part. Later on in this chapter, we'll see how to discover these values using the Orchard tools.

We just fixed our bio summary templates so that the content wouldn't just dump out labels and values for each property. However, when we click through to an individual bio, we can see that the detail view for our page looks like our summary did before our customization.

To get our bio details and summary views more in line, we'll add a new alternate template. Start by creating a new Razor file named *Content-Bio.cshtml* in the *Views* directory in "TheThemeMachine." This new template will be only a slight variation of the `Bio` item summary template we just created:

```
<h1>@Model.ContentItem.TitlePart.Title</h1>

@{ var fileName = Model.ContentItem.Bio.Headshot.FileName;
  if (!string.IsNullOrEmpty(fileName)) {
```

```


}
}
Plays @Model.ContentItem.Bio.Instruments.Value.ToLower()
for Daisy's Gone.<p/>
Originally from
@Display(Model.ContentItem.Bio.Birthplace.Value).<p />
Currently lives
in @Model.ContentItem.Bio.LivesIn.Value.<p />
@Html.Raw(Model.ContentItem.BodyPart.Text)

```

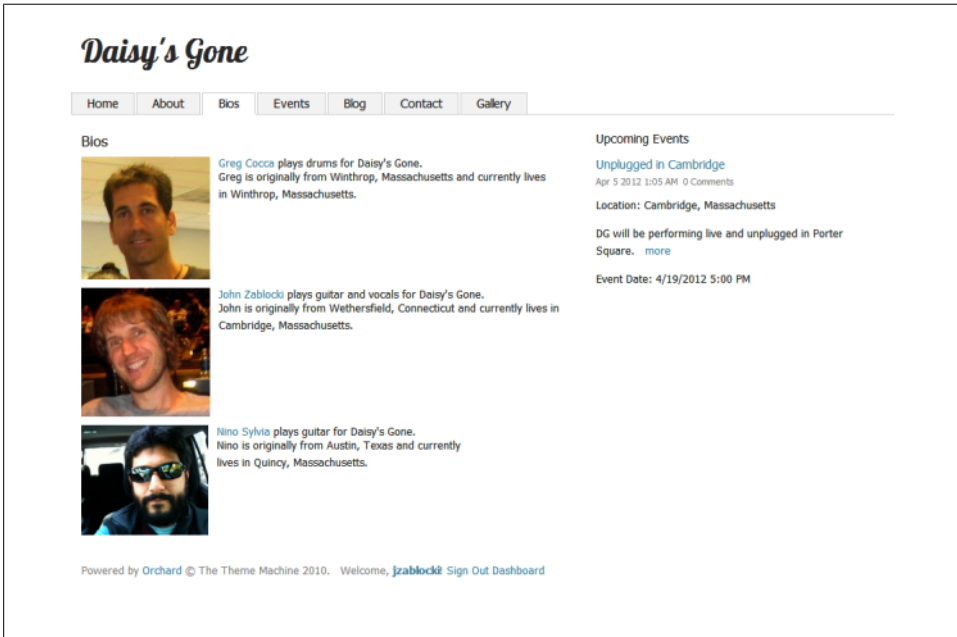


Figure 3-4. The customized bio listing

It's also worth noting that if we removed the summary alternate template, this new template would then be used to render summary listings as well since the name of this file doesn't specify a display type (i.e., *Summary* or *Detail*). We could also have (and probably should have) named this file *Content-Bio.Detail.cshtml* to remove the potential impact on summary displays.



Since shapes are **dynamic** types, it's occasionally helpful to attach the Visual Studio debugger to your Orchard application to inspect the object being bound to your view. You can set a break point in a Razor file in a code block, which is any executable line between a `@{ }`.

Customizing Events

Like our bios in our bio listing, events in our events listing are rendered one property at a time with labels and values occupying one line each. Arguably, this display is appropriate for events, so we won't create an alternate template for event summaries. However, we do want to adjust the order in which fields are rendering. [Figure 3-5](#) shows that the location appears before the body field, while the event date is after. We'll move both of these fields ahead of the HTML body field.

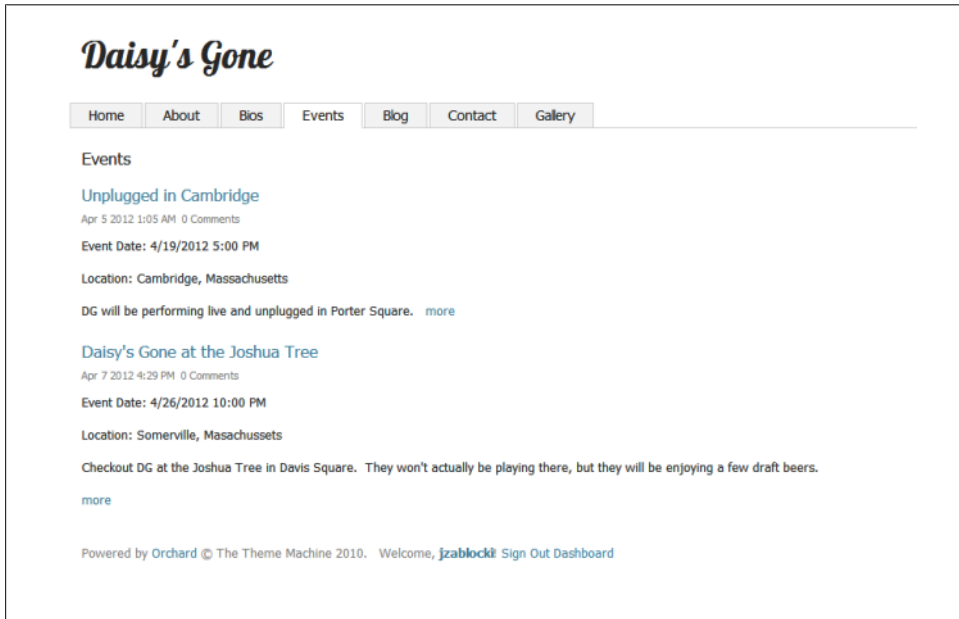


Figure 3-5. Default event listing

Placement Files

To rearrange the fields, we'll use a special XML file named *Placement.info* that lives in the root of a template directory. This file defines rules for rendering content items, parts, and fields. “Place rules” allow module developers and template designers to provide preferred layout ordering for content pieces. We'll modify the *Placement.info* file in the root directory of the “TheThemeMachine” theme by replacing its content entirely with this XML:

```
<Placement>
  <Match ContentType="Event">
    <Match DisplayType="Summary">
      <Place Fields_Text="Content:before">
      </Place>
      <Place Fields_DateTime="Content:before">
      </Place>
```

```

    </Match>
  </Match>
</Placement>

```

The `Match` element will limit the placement rule so that it affects only items of type `Event` that are shown with the summary display type. The `Place` rules that follow the `Match` specify that the text and date time fields will appear before the `Content` zone, where the HTML body is placed. If we want to force the event date to render after the location, we could provide relative numeric values instead of “before” or “after”:

```

<Place Fields_Text="Content:2" />
<Place Fields_DateTime="Content:3" />

```

Field Templates

We also want to jazz up our location field so that it renders a link to Bing Maps for that particular location. To do so, we’ll create an alternate template for just that field. This is different from what we did with `Bio` content, where the alternate template affected an entire item’s rendering.

Create a new directory named *Fields* under the *Views* directory in the “TheThemeMachine” theme. To that new directory, add a file named *Common.Text-Event-Location.cshtml* (prefixing this file with *Fields* would make the new directory unnecessary). The body of this new template will simply supply the value of our event’s location to the “q” parameter used by Bing Maps when it performs a search:

```

@Model.Name:
<a href="http://www.bing.com/maps/?q=@Model.Value">
  @Model.Value
</a><p />

```

If you refresh the event listing, you’ll now see the location field is wrapped in a link to Bing Maps. If you click through to view the actual event, you’ll see that the location field is also linking to Bing Maps (Figure 3-6). Our field alternate applies to both summary and detail displays.

There is still one last problem we want to fix. The ordering of the event fields in the details display type is not the way we want it. The event date is rendering below the content as was previously the case with event summary displays (Figure 3-6). To fix this, we’ll simply remove the `Match` tag from *Placement.info* where we’d previously restricted the display rule to summary only:

```

<Placement>
  <Match ContentType="Event">
    <Place Fields_DateTime="Content:1" />
    <Place Fields_Text="Content:2" />
  </Match>
</Placement>

```

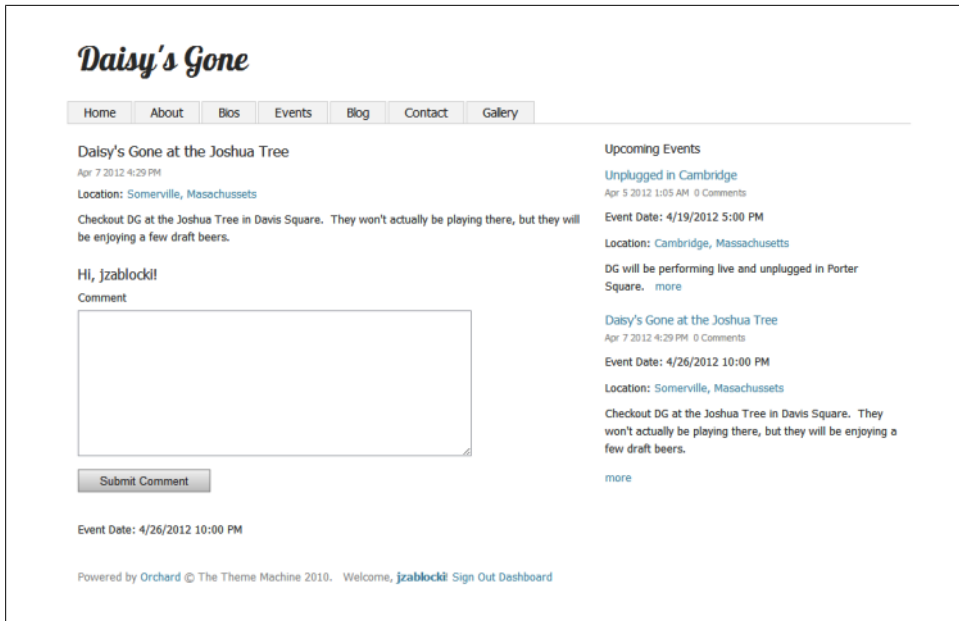


Figure 3-6. The event location linking to Bing Maps

Shape Tracing

While for some shapes it's easy enough to figure out the correct naming for alternate templates based on the documented rules, it's not always obvious. Fortunately, there's the **Shape Tracing** module to help navigate the complex hierarchy of shapes that compose a page. **Shape Tracing** is installed but disabled when you create a site with the "Default" recipe. It can be enabled on the admin dashboard under Modules→Features. It's listed under the "Designer" category.

After you enable the **Shape Tracing** module, across the bottom of each page of your site you'll see a small bar with a small square icon all the way to the right. Clicking it will bring up the tools for outlining the shapes on a page.

Shape Tracing is a JavaScript-based utility that provides information about how zones and the shapes contained in these zones are rendered. As you move your mouse around the page, the **Shape Tracing** tool highlights content pieces such as zones, widgets, content parts, and fields.

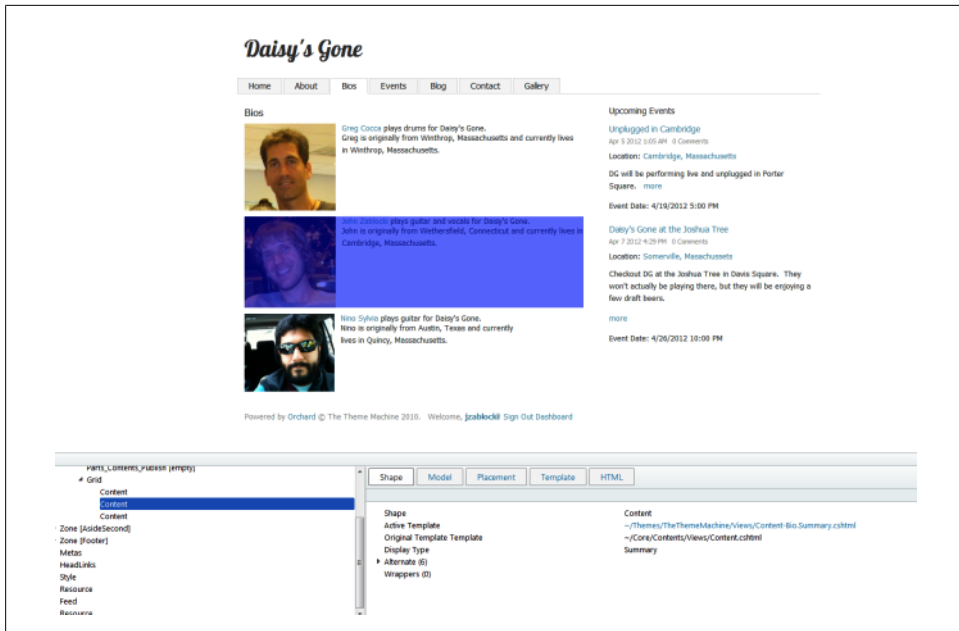


Figure 3-7. The Shape Tracing developer tools

Finding alternate templates

To demonstrate the utility of the **Shape Tracing** module, we'll create an alternate template for the content of the "Body" field on our home page. We're going to add an image to the content and float it to the left of the text. We could use the HTML editor to add an image, but this template will give us a little more control over layout.

With the **Shape Tracing** module activated (click the little square), mouse-over the **Content** zone. Click when one of the paragraphs is highlighted. The shape tracing console will display information under the "Shape" panel (Figure 3-7).

If you expand the "Alternate" tree under the "Shape" panel, you'll see three possible options for creating an alternate template. Next to each option is a "Create" button that you can click to have the tool generate an alternate template for you in your current theme. Click "Create" to generate the *Parts.Common.Body-11.cshtml*. Recall that the number in the filename is the unique content ID of a content item and yours will vary.



In order to navigate your site normally again, you must click the dash icon in the upper-right corner of the **Shape Tracing** tool, to turn off its functionality.

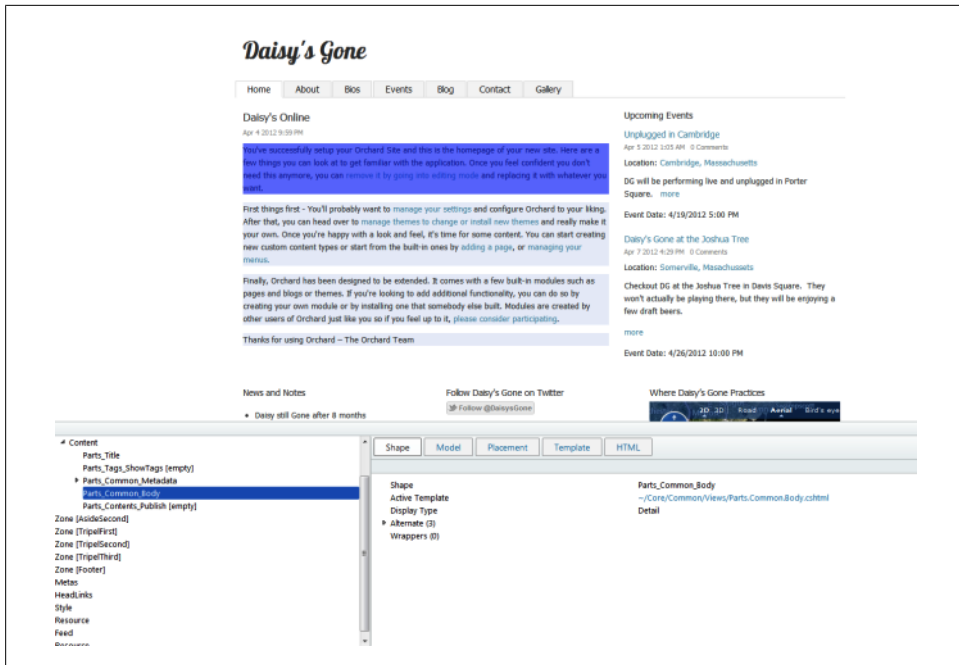


Figure 3-8. The Shape Tracing tool after clicking on a shape



If you use the Shape Tracing tool to create new alternate templates, these files won't be added to the project automatically. You'll need to click "Show All Files" in Visual Studio's Solution Explorer to see them. You can then select "Include in Project" from the context menu.

After you click to create this new alternate template, back in the *Views* directory of the "TheThemeMachine" theme, you'll find the new file. By default its code just renders the HTML property of the *Model* property of the page. The Shape Tracing module creates alternate templates based on current template for that piece of content.

Now we'll add the new image for our home page. Create a new media folder by clicking Media→Add a Folder. Name the new folder "Art" and click "Save." Click into "Art" and then click "Add Media" and upload an image.

After it's been uploaded, click the filename to get its path (copy the "Embed" value). Then back in Visual Studio, open *Parts.Common.Body-11.cshtml* and add an HTML *img* tag with the source set to this new image. We'll render the saved HTML (which is simply the "Body" content) next to our Daisy's Gone logo:

```

@Model.Html
```

We also could have named the template *Parts.Common.Body-Page.cshtml*. This naming would have resulted in all pages with a **Body** part being affected. For our site, that would have meant the home, contact, gallery, and about pages would all include the new image. Even the “News and Notes” HTML widget would be affected.

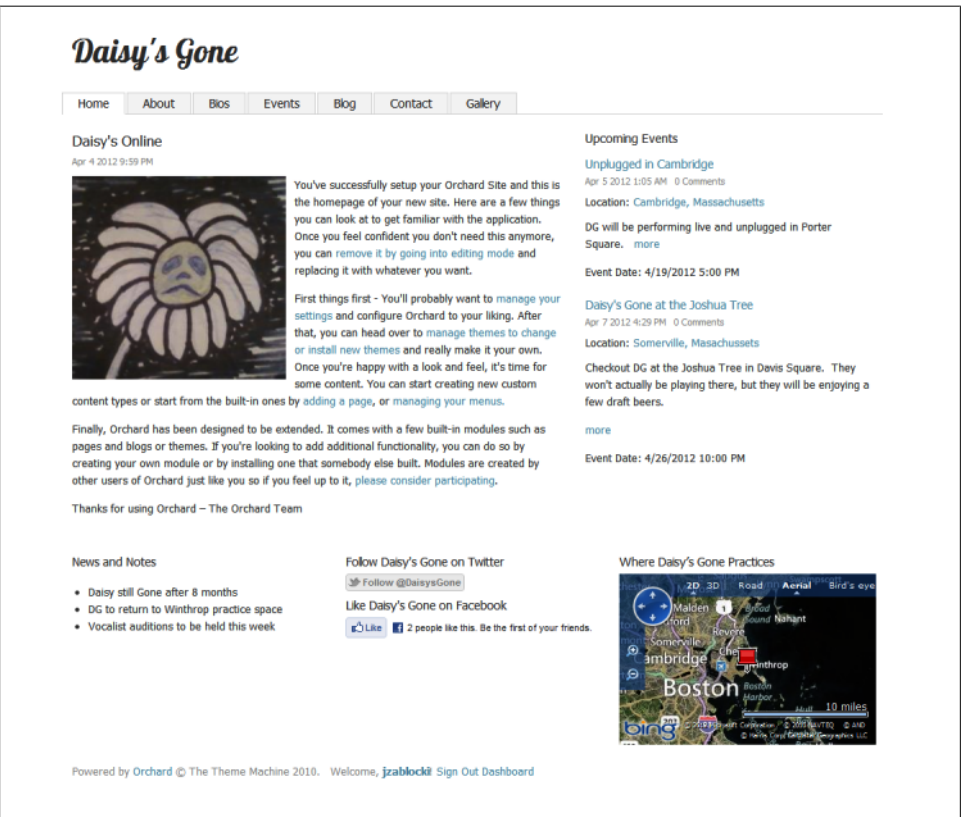


Figure 3-9. The alternate Body template

Finding shape properties

The Shape Tracing module also provides us with a way to figure out object hierarchies for our template alternates. If you wondered how the expression `Model.ContentItem.AuthoroutePart.DisplayAlias` was discovered for use in our bio templates, it was the Shape Tracing module.

Hover again over a piece of content while the Shape Tracing module is enabled, but this time click the “Model” tab in the console. You’ll see a tree (Figure 3-10) that contains information about the `ContentItem` property of the view’s `Model` property.

You’ll see details of content parts and fields. As you click through the tree, the expression to access these properties appears just below the tabs. Simply copy that expression

into your alternate template. For example, you could highlight an event in our “Upcoming Events” widget and click through the model to find the location expression `@Model.ContentItem.Event.Location`.

The **Shape Tracing** module can also be used to see the placement rules for a piece of content, the current template used to display that content and the actual HTML rendered for that content. It’s an incredibly powerful tool to have at your disposal. You’ll use it frequently while customizing the display of content on your site. Just don’t forget to disable the module if you’ve enabled it on your live site for testing.

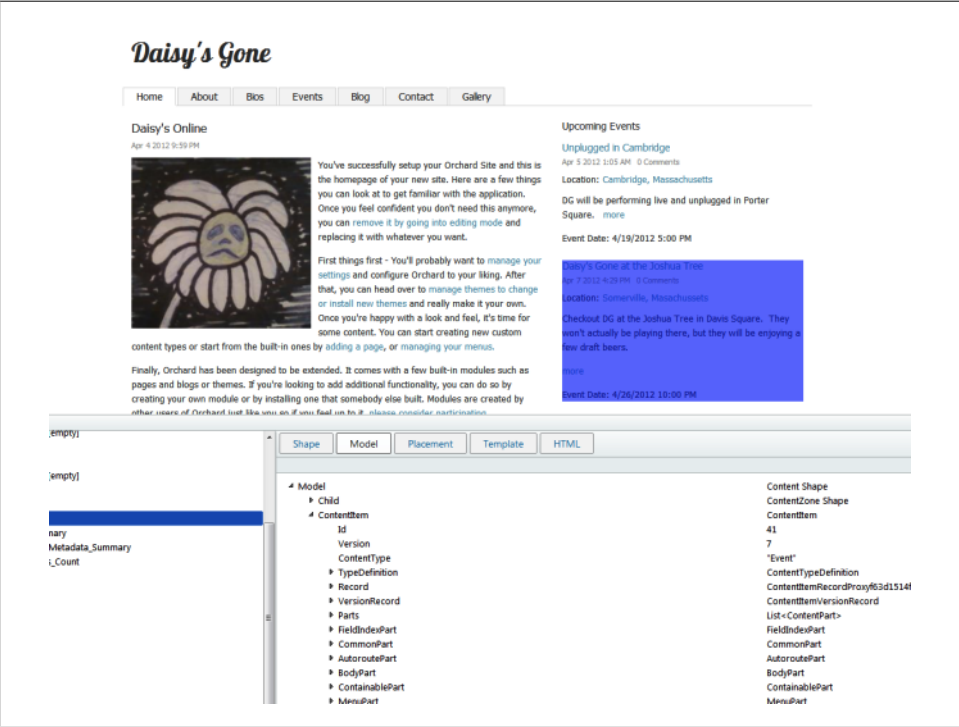


Figure 3-10. The Shape Tracing model panel

Summary

We’ve seen in this chapter that Orchard provides extensive ways to customize content. We can provide alternate renderings for full content items or individual fields within our content. Our alternates were somewhat utilitarian in that they offered improved and new functionality or changed the way something rendered. The second part of customizing content in Orchard is to create striking visual changes to the way content is rendered. We’ll learn how to make such changes in [Chapter 4](#).

Creating Themes

In the previous chapter, we saw that it's possible to customize the look and feel of an Orchard site by creating alternate templates for pieces of content. While this feature does provide some flexibility over how content is rendered on pages, it doesn't easily allow for wholesale changes to the way a site looks.

In order to achieve this broader goal, we'll look at creating our own themes. We've already taken a brief tour of the default theme—"TheThemeMachine"—that is part of a standard Orchard installation. In this chapter, we're going to take a look inside of that theme to understand how to create our own.

At this point, our Orchard development has been limited to editing a couple of view files. We wrote some C# code in those Razor templates and learned a little about how the content is modeled and displayed in a view. However, we haven't really been exposed to the Orchard development experience. To gain this exposure, we first need to learn about a couple of tools that make Orchard development easier.

The Orchard Command-Line Interface

It probably seems strange to consider using a command-line interface (CLI) with a web-based CMS. However, the `Orchard` CLI offers quick access to many common admin functions without the need to open up a browser and navigate to different property pages. Assuming you've been working in the Orchard solution, the CLI has been ready for use since you first compiled your app and set up your recipe.

Getting Help

To get started with Orchard's CLI, open up a command window (or PowerShell) and navigate to the *bin* directory of your Orchard site. There you'll find file *Orchard.exe*. Execute that file. After a few moments, you'll see an Orchard prompt.

```
PS C:\dev\Orchard> cd .\src\Orchard.Web\bin
PS C:\dev\Orchard\src\Orchard.Web\bin> .\Orchard.exe
```

```
Initializing Orchard Session. (This might take a few seconds...)
Type "?" for help, "exit" to exit, cls to clear screen
orchard>
```

There are a number of commands you could execute. To get CLI help, there are two commands you should know. The first simply tells you how to perform tasks like quitting, clearing the screen, and getting more help:

```
orchard> help
```

One of the items listed when you execute the `help` command explains how to get help for the commands that allow you to work with your Orchard site:

```
orchard> help commands
```

Executing `help` commands gives you a list of CLI commands ranging from user creation to page creation. You'll also see how to go one level deeper for command help:

```
orchard> help page create
```

To test your session, enter the command that simply lists the site cultures, which are used for the internationalization of your site:

```
orchard> cultures list
Listing Cultures:
en-US
```

Code Generation Tools

If you ran the help commands, you probably saw that the `Orchard` CLI has commands for many of the common tasks you'd normally perform in the admin pages, including enabling and disabling features. We're going to use this command to enable a feature that will give us additional command line tools:

```
orchard> feature enable Orchard.CodeGeneration
Enabling features Orchard.CodeGeneration
Code Generation was enabled
```

Once the code generation tools are enabled, they will provide useful shortcuts for creating and managing themes and modules. `Orchard.CodeGeneration` is an example of the Orchard's extensibility. In fact, you can build your own command-line tools for Orchard. It's as simple as creating a class that extends the base class `DefaultOrchardCommandHandler`.



The code generation module is not installed by default with Orchard, but it was found in the source when you downloaded the zip or cloned the repository. If you are not working with the solution, you might need to install this module from the Orchard Gallery.

Code Generating a Theme

In the previous chapter, we saw that a theme is a collection of files contained in a directory in the **Themes** project in the Orchard solution. There's nothing special about a theme other than it follows a set of conventions and is stored in the *Themes* directory under the *Orchard.Web* project.

To start building a new theme, you could simply copy the directory structure for the “TheThemeMachine” theme and paste it into a new sibling directory. We'll instead use the command line code generation options to create our theme. Later we'll learn how these tools can help us avoid starting from scratch.



The Orchard solution is organized using solution folders, so it appears that the **Themes** project lives outside of the web project's directory structure. However, the **Themes** project and theme files are actually nested in the filesystem under the *Orchard.Web* directory.

Return to the CLI and your Orchard prompt. We're going to create a new theme named “DaisysTheme.” There are three options we'll want to consider before we create our new theme:

CreateProject

Whether to create new project for this theme. The default is false.

IncludeInSolution

Include this theme in the solution. The default is true.

BasedOn

Inherit default templates from an existing theme.

For our new theme, we're neither going to inherit from an existing theme nor create a new project. We'll simply run the theme code generation without any arguments:

```
orchard> codegen theme DaisysTheme
Creating Theme DaisysTheme
Theme DaisysTheme created successfully
```

Return to Visual Studio where you'll be prompted to reload the solution. The *codegen* utility modified the **Themes** project and forced a reload of the solution. After reloading, we can start to inspect the anatomy of our new theme.

The Structure of a Theme

The structure might look somewhat familiar to an experienced ASP.NET MVC developer. As is the case with the standard Visual Studio ASP.NET MVC project template, there are directories for *Scripts*, *Styles*, and *Views*. The purpose of each of these and the other directories is as follows:

Scripts

Directory for JavaScript files

Styles

Directory for CSS files

Views

Directory for Razor template (*.cshtml) files

Content

Directory for images and other static content

Zones

Directory for templates that wrap zones

Additionally, the generated theme template includes a *Placement.info* file. Recall that this is the XML file that instructs Orchard as to how or whether to layout fields, parts, and items. There are also numerous *Web.config* files that are used by ASP.NET to set up some configuration plumbing for ASP.NET and ASP.NET MVC. There are two other files worth noting, namely *Theme.txt* and *Theme.png*. Both of these files are used by Orchard to describe your theme to the admin pages.

Back in the admin dashboard, select Themes→Installed. You'll see three themes listed (Figure 4-1). The current theme will be the default "TheThemeMachine." There's a second theme called "The Journalist" and our new theme named "DaisysTheme." However, some things don't look quite right with our new theme.

Notice that the name is "DaisysTheme" without a space. Authorship is attributed to "The Orchard Team." The version is already set to 1.0 and the description and URL are also wrong. As we'll see shortly, the preview image also fails to accurately represent our new theme at this stage. You might have guessed that Orchard is using *Theme.txt* and *Theme.png* to determine what values to plug into this admin page.

Before returning to Visual Studio to look at these files, let's first make our new template the current template by clicking "Set Current." Once the switch is complete, refresh your site. What you'll see is that your site is suddenly without any discernible structure or styling (Figure 4-2). Also notice that our alternates are gone. We'll add some of those pieces before we modify the metadata that's used by the Dashboard.

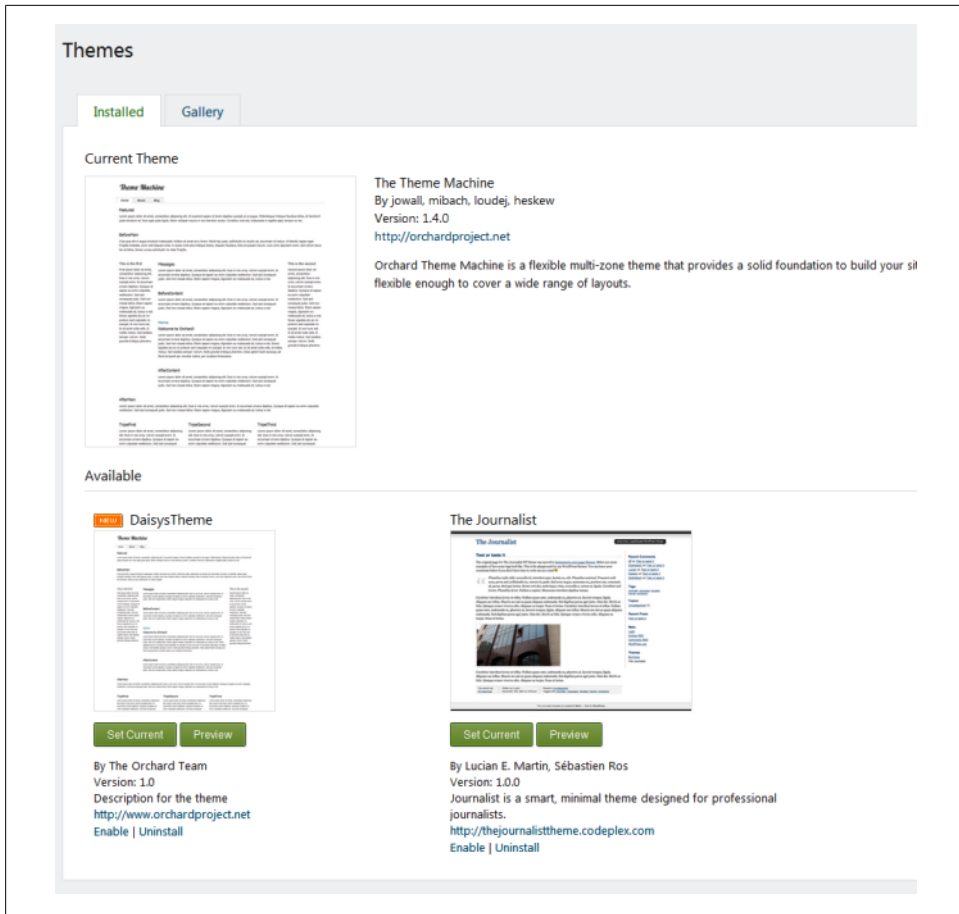


Figure 4-1. Installed themes



The purpose of this chapter is to introduce theme development. As I am not a designer, I am intentionally keeping the theme we will build simple, including all graphical treatments, styles, and HTML.

Default Content Templates

If you view the source for the home page, you'll notice that there is a full HTML document wrapping the content. This might seem strange, since we haven't actually defined any master page or layout files. These default template files do exist, though. Orchard includes them in the `Orchard.Core` project.

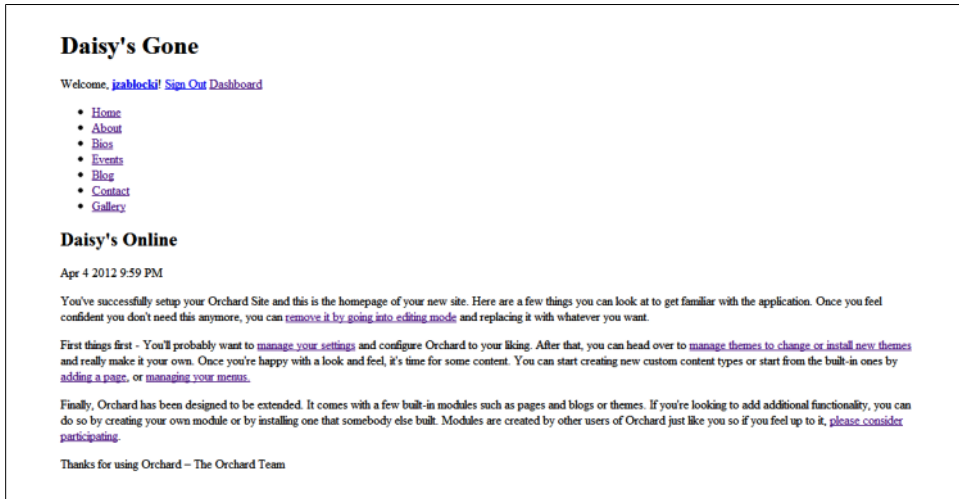


Figure 4-2. A new theme without any styles or structure

If you expand that project in Visual Studio's **Solution Explorer**, you'll see a *Views* directory nested under a *Shapes* directory. The view files inside this directory are used by Orchard as safe defaults for displaying content when no suitable template alternates are found in a theme for a given piece of content.

Open the files *Document.cshtml* and *Layout.cshtml* to see the HTML that is wrapping the content in our home page. *Document.cshtml* defines the basic structure of an HTML document, including the `html`, `head`, `title`, and `body` tags. *Layout.cshtml* defines a very basic arrangement of `div` elements on the page. Notice that the template for the title of the page is actually found in the *Parts.Title.cshtml* file in the *Views* directory under *Title* in the *Orchard.Core* project.

As is hopefully now clear, Orchard uses a hierarchy of templates when determining how to render content. In the previous chapter, when we defined alternate templates, we simply added files to the current theme and those files took precedence over those found in the default templates directories. When creating a theme or module (in the next chapter), you could choose to inherit a template or override it at any level (item, type, part, or field).

Working with Views

In [Chapter 3](#), we created alternate templates in the form of Razor template files. However, we didn't explore these templates in any detail. Before we build a theme and continue our Razor efforts, it's worth a quick look at some of the functionality that Orchard adds to the standard base class used by Razor views. For more on Razor, see *Programming Razor* by Jess Chadwick (O'Reilly).

Orchard extends the `System.Web.Mvc.WebViewPage` base class used by Razor views with its own `WebViewPage`. We've already seen a method from this subclass. When we created our zones, we used its `Display` method. There are other useful helper methods in this base class.

The `Style` property of `WebViewPage` has an `Include` method that will render a link tag that points to the filename provided as its argument. It assumes that your file is in the `Styles` directory of your theme. Similarly, there is a `Script` property with methods related to including JavaScript blocks and files.

Both the `Script` and `Style` properties are of type `ResourceRegister`, which provides an additional method named `Require`. Given a resource defined in a manifest class (we'll learn more about this class in *Creating Widgets*), `Require` will find a script or style and ensure that it's included only once.

`WebViewPage` also includes some convenience methods, such as the null and whitespace checking `HasText` method. Orchard also provides a `StringExtensions` class in the `Orchard.Utilities.Extensions` namespace. This class has methods such as `Camel Friendly`, `Ellipsize`, and `HtmlClassify`, all of which may be useful in views.

Layouts

For our theme, we'll consider this document wrapper sufficient and won't override it with a new `Document.cshtml`. We'll instead start our theme with a new layout file. Add a new Razor file named `Layout.cshtml` to the `Views` folder in the "DaisysTheme" theme. If you save this file with no content (an empty HTML file) and refresh any page on your site, you'll see that the content has disappeared.

By including a `Layout.cshtml` file in our template, we've instructed Orchard not to use its default layout template. Instead we've instructed Orchard to use this new, empty file. Notice though that the page title still appears. As mentioned, the title was included in `Document.cshtml`, which we chose not to override.



If you've viewed the source of any of your site's pages as we've been making changes to the theme, you may have noticed a great deal of JavaScript content. The script is from the `Shape Tracing` module that we enabled in the previous chapter. It would not appear on production sites unless you left that module enabled.

We'll add some code to `Layout.cshtml` to get some content back on our site. Start by adding the following snippet:

```
<div id="main">
  @if (Model.Content != null) {
    @Display(Model.Content)
  }
</div>
```

This simple chunk of Razor code demonstrates a couple of key patterns for building layouts in Orchard. We're going to explore this pattern in more detail later in this chapter. For now, recognize that we null-check a property on our view's `Model` and call `Display` on that property when it's not null.

Recall that the data bound to our views are dynamic types known as shapes. The `Model` property of our view is a representation of these shapes. When you call the `Display` method, it's going to check the runtime type of the argument you provided. In this case, the type will contain metadata indicating that it's a "zone," which will allow the `Display` method to properly render content in that given zone.



The `Display` method is actually a read-only dynamic property that is defined in Orchard's `WebViewPage`, which is the base page type for Razor views in Orchard. This property returns an instance of a callable dynamic object, which is why the method-call syntax works.

Saving the layout and refreshing the home page, we now see that the main content has been added. However, we've lost our navigation and other zones. Let's add the navigation by placing the block of code that follows above the content snippet we previously entered:

```
@if (Model.Navigation != null) {  
  <div id="layout-navigation" class="group">  
    @Display(Model.Navigation)  
  </div>  
}
```

Once navigation has been added, we can now refresh the home page and click through each of the pages to see that content is in fact displaying on each page. While most pages look like their "TheThemeMachine" equivalents without any CSS, the home page is noticeably missing the widgets we'd previously added.

If we want the Bing Maps widget to show up in our new theme, we need to include a zone named `TripelThird` and a block of Razor/HTML as follows:

```
@if (Model.TripelThird != null) {  
  <div id="tripel-third">  
    @Display(Model.TripelThird)  
  </div>  
}
```

Zones and Layers

At this point we can see that zones are simply sections defined in our layout templates. We use the `Display` method of the view to create them. Adding widgets to zones in the admin tool creates the relationship that allows the null-checks that surround the `Display` calls to evaluate properly. Had we not added a widget to the zone `TripelThird`, then `Model.TripelThird` would evaluate to null.

Let's take a quick detour from Daisy's Theme to explore zones a little deeper. Start by opening up *Theme.txt* in our theme's root directory. The "Zones" entry in this file is used by Orchard to display the list of zones that appear when you click "Widgets" on the admin menu. That list is currently populated by zones defined in other installed themes and Orchard will tell you as much when you visit that page.

Add a "Zones" section to your *Theme.txt* named "MoreContent":

```
Zones: MoreContent
```

Next return to *Layout.cshtml* and add a new zone:

```
@if (Model.MoreContent != null) {  
    <div id="more-content">  
        @Display(Model.MoreContent)  
    </div>  
}
```

If you refresh the "Widgets" admin page, you'll now see a zone named "MoreContent" above the zones defined in "TheThemeMachine." Click Add→Html Widget, add some content, and save. Next, refresh the home page (or any other page). You'll now see that the zone is displaying on each page (Figure 4-5).

Let's limit this new widget so that it appears only on the home page. Click on the "More Daisy's Content" link (the name of the HTML widget) listed with the "MoreContent" zone. On the property page for that widget, choose the layer named "TheHomepage" and click Save. Click through to each of the pages to see that the layer rule has enabled this zone only for the home page.

Alternate Templates

As you navigate around the site, you'll notice that we've lost the customization built in [Chapter 3](#) for rendering bios and events. If we want to get these templates back, all we need to do is add those Razor files into our new template.

If you move or copy *Content-Bio.Summary.cshtml* to the *Views* directory of our "DaisyTheme" theme and refresh the bio page, you'll see the listed bios are displaying content as they were previously. Of course, without any CSS in our theme you'll notice that the rendering lacks any style (Figure 4-4).

Theme Inheritance

At this point, our theme isn't particularly stylish or interesting. Our layout is pretty limiting as well. What we really want is some HTML that's easily styled by a skilled designer. Fortunately, the work for that has already been done.

The theme "TheThemeMachine" defines a very flexible layout file. We could simply copy that into our theme, but instead we're going to inherit it into our theme. In

Daisy's Online

Apr 4 2012 9:59 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 this anymore, you can [remove it by going into editing mode](#) and replacing it with whatever you want.

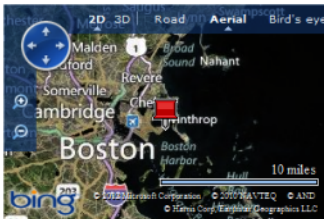
First things first - You'll probably want to [manage your settings](#) and configure Orchard to your liking. After that, you can head over to your own. Once you're happy with a look and feel, it's time for some content. You can start creating new custom content types or s

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 your own module or by installing one that somebody else built. Modules are created by other users of Orchard just like you so if yo

Thanks for using Orchard – The Orchard Team

- [Home](#)
- [About](#)
- [Bios](#)
- [Events](#)
- [Blog](#)
- [Contact](#)
- [Gallery](#)

Where Daisy's Gone Practices



More Daisy's Content

This is some more content about Daisy's Gone...

Figure 4-3. The MoreContent zone

“DaisysTheme” open *Theme.txt* and add the line that follows. Then delete *Layout.cshhtml* (the one we created):

```
BaseTheme: TheThemeMachine
```

After you refresh the home page, you'll see that our site has returned to its “TheThemeMachine” roots. However, we obviously want to customize our look and feel a bit. To deviate from the inherited theme, we need to override the default styles found in “TheThemeMachine.”

Create a new file *Site.css* in the *Styles* directory of the “DaisysTheme” theme. After you create the empty stylesheet, you'll see after refreshing your site that we've again lost our styling, but maintained our layout and alternate templates (Figure 4-5).

Bios



[Greg Cocca](#) plays drums for Daisy's Gone.

Greg is originally from Winthrop, Massachusetts and currently lives in Winthrop, Massachusetts.



[John Zablocki](#) plays guitar and vocals for Daisy's Gone.

John is originally from Wethersfield, Connecticut and currently lives in Cambridge, Massachusetts.



[Nino Sylwia](#) plays guitar for Daisy's Gone.

Nino is originally from Austin, Texas and currently lives in Quincy, Massachusetts.

- [Home](#)
- [About](#)
- [Bios](#)
- [Events](#)
- [Blog](#)
- [Contact](#)
- [Gallery](#)

Figure 4-4. An alternate template in the *DaisysTheme* theme

Unfortunately, there's no way for our theme to inherit both the layout and stylesheet from the "TheThemeMachine" theme. *Layout.cshtml* explicitly includes only a single stylesheet named *Site.css*. If we want to inherit the entire layout file *and* customize the style, we have to copy the contents of *Site.css* from the *Styles* directory of "TheThemeMachine" into our new file. Otherwise, we have to modify the layout file to look for an additional stylesheet.

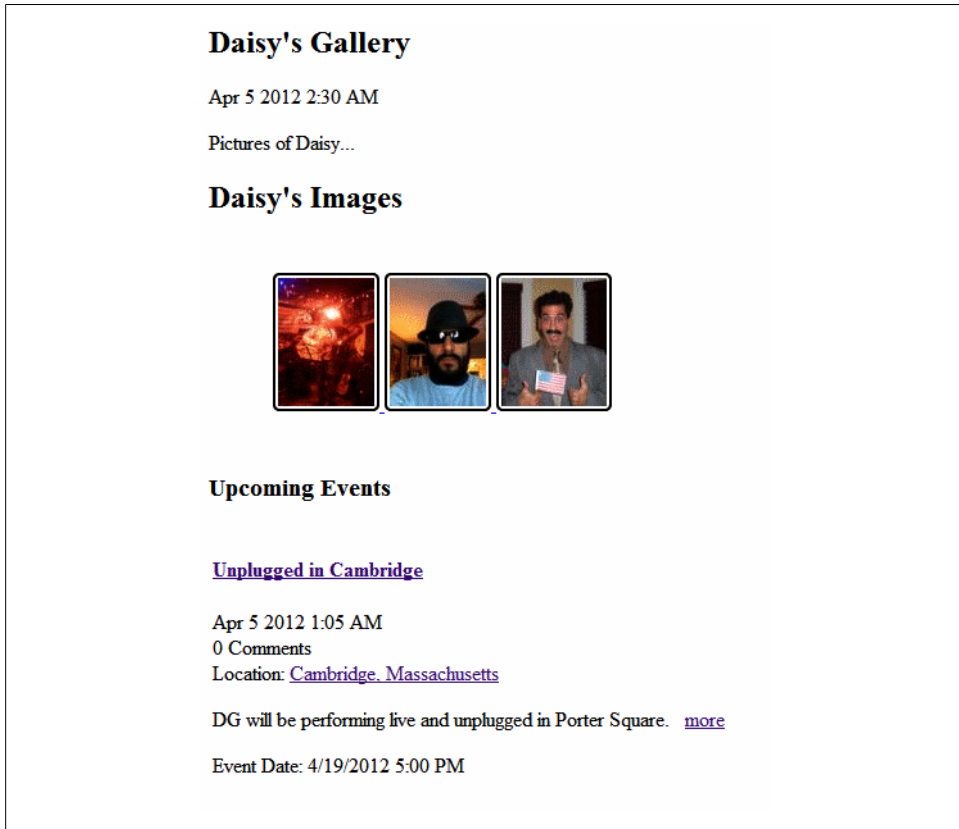


Figure 4-5. *DaisysTheme* inheriting layout from *TheThemeMachine*

Basic Styling

Copy the stylesheet content over into our new theme (*Site.css*) and save the stylesheet file. Refresh the site to see that we're now back to the "TheThemeMachine." Again, we'll leave the design lessons for the designers, but we'll modify some of the basic UI to make our theme a little more unique.

Since we're designing a site for a rock band, we'll change the background color to a blackish color. Locate the body selector in the stylesheet and we'll go from a light theme to dark simply by changing the background color:

```
body {
  line-height: 1;
  font-size: 81.3%;
  color: #434343;
  background: #303030;
  font-family: Tahoma, "Helvetica Neue", Arial, Helvetica, sans-serif;
}
```


Of course, it's a bit hard to read the gray text on the dark-gray background. So let's lighten up our content areas. We could go into the individual page sections and set each to have a white background, but there's an easier way. We can take advantage of the fact that the layout from which we're inheriting wraps various groups of zones in `div` elements with a class name of "group":

```
.group {  
    background: #fff;  
}
```

Let's also update the font that's used for the header of the site. By default it uses a font named "Lobster." You're probably thinking that you don't have "Lobster" installed, yet you've somehow been seeing the correct cursive font on the header. Orchard assumes modern web standards by default, so our theme is able to make use of the `@font-face` directive in CSS3. More specifically, it uses the Google Web Fonts API:

```
Style.Include("http://fonts.googleapis.com/css?family=Lobster");
```

If we want to change this font, we have a few options. We're again faced with the dilemma of whether to modify or copy *Layout.cshtml* in "TheThemeMachine" to include our desired change. Since we're just changing styles, we're going to keep the layout in place and take a different approach.

We'll simply import a new web font in our stylesheet and then set the branding element's font-family to our new font. Start by adding a new `@import` directive to the top of our stylesheet:

```
@import url(http://fonts.googleapis.com/css?family=Frijole);
```

In our template, the header text is rendered in an `h1` element named "branding." We'll simply set style for that element to use our newly imported font:

```
#branding a  
{  
    text-decoration:none;  
    color: #434343;  
    font-family: 'Frijole';  
}
```

Styling Projections

Next we're going to modify the event listing so that the event titles have a background color and text that's in all caps. We could write our CSS selector expression to affect all `a` tags that follow `h1` tags as that's the way events are rendered, but such a selector would not be limited to events matching that pattern. Instead we're going to inject a class name into each event row.

In the admin dashboard, navigate to "Queries" and select the row for "All Events." Click to edit the "1 columns grid" that we created previously. In the section with the heading "Html properties," enter a value "event-row" under the "Row class" and save.



Figure 4-6. A styled projection

We could also have chosen from predefined dynamic expressions (in the drop-down menu for that field), but a static class is sufficient for our purposes.

After you save the new row class, add a new CSS rule to affect a elements that follow h1 elements that follow a tr element with a class name of “event-row.” We’ll style the anchors to be displayed as “block” so that we have equal length backgrounds:

```
tr.event-row h1 a {
    background-color:#BACEFF;
    padding:3px;
    color:#000;
    width:300px;
    display:block;
}
```

Figure 4-6 shows the template with our new styling.

Shape Wrapping

We’re going to add another template to our theme, but first we have to bring back the rest of our content customization. Copy or move *Content-Bio.cshtml*, *Parts.Common.Body-11.cshtml*, and *Placement.info* from “TheThemeMachine” to “DaisysTheme.”

After moving those files, add a new template named *NewsAndNotes.Wrapper.cshtml*. Unlike our other templates, this one will surround its target with HTML and won't actually modify the shape template itself. To call attention to band activity, the code for this wrapper will simply add a `div` element with a yellowish background to our "News and Notes" HTML widget:

```
<div style="background:#FFE8A5;padding:2px;">
    @Model.Html
</div>
```

An additional step is required for this wrapper to be used on our site. We need to update *Placement.info* to instruct Orchard to use this template. The match constraint will cause this rule to apply to the HTML widget on our home page:

```
<Match ContentType="Widget" Path=~/">
    <Place Parts_Common_Body="Content:5;Wrapper=NewsAndNotes_Wrapper" />
</Match>
```

This scenario is admittedly slightly contrived, as we could have used an alternate template for our zone to accomplish the same thing. However, it does illustrate an additional layer of customization available to theme designers.

Theme Metadata

We'll consider our theme sufficiently styled at this point (at least by developer standards). Now we're ready to update the metadata used by the admin tool. Return to Visual Studio and open *Theme.txt* in the root of the theme. Most of the values are pretty obvious; Name, Author, Website, Description, and Version are included by default and shouldn't merit any further description. If you personalize these values and return to Themes→Installed in the Dashboard, you'll see these updated values.

Theme Previews

There are two final files you need to know about when developing themes: *Theme.png* and *ThemeZonePreview.png*. These files both live at the root of a theme. The former is typically a screen-grab of your theme's homepage that will be used in the Dashboard and the gallery to provide a preview of your theme. The latter is an image used on the Widget admin page to provide a preview of where zones are conceptually placed in layout files.

Theme Credits

The last update we'll want to make is to modify the chunk of HTML on the bottom of the page that gives credit to "TheThemeMachine" as this site's theme. While that statement is partially true, we'll instead claim credit for our "Daisy'sTheme" theme. We'll need to override a file that's in "TheThemeMachine" named *BadgeOfHonor.cshtml*.

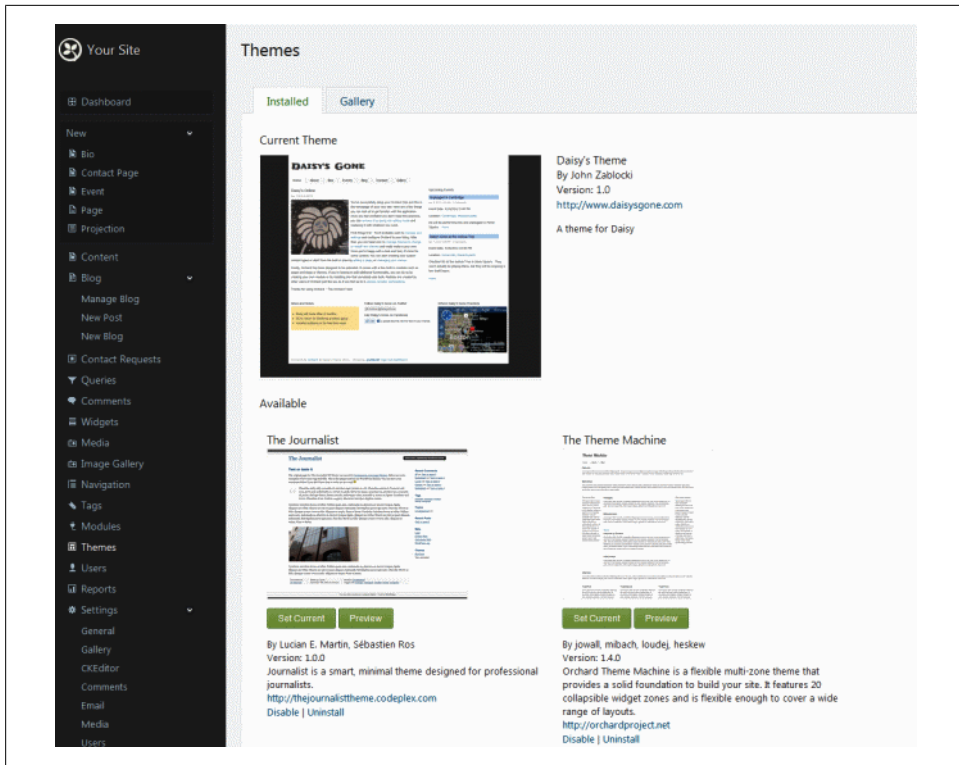


Figure 4-7. Daisy's Theme with updated metadata

Copy it from the `Views` directory of “TheThemeMachine” into our theme’s `Views` directory. Modify the content so that the `span` with the “copyright” class has the content as follows:

```
<span class="copyright">@T("&#169; Daisy's Theme 2012.")</span>
```

Summary

In our exploration of developing themes, we’ve seen that we don’t have to start from scratch when developing our site’s look and feel. In fact, this is a common way to develop themes. The “TheThemeMachine” theme provides a flexible, barebones layout that could easily be styled by a skilled designer.

There is little reason to create new layout files and complex zone schemes when most of what you need may be found in this theme. Simply inherit from it and create a new stylesheet with new graphical treatments. Of course, your needs may require that you copy the entire “TheThemeMachine” theme to get started. Moreover, the same approach we took to modifying “TheThemeMachine” applies to any theme you install into Orchard (theme license permitting).

Creating Modules

We’ve already seen that Orchard may be extended easily and quickly by installing modules from the Gallery. When we wanted to add an image field to our event pages, we just installed the `ImageField` module. When we didn’t want to create our own contact form, we just installed one that someone had already written and shared. So what happens when we want functionality that isn’t readily available in the Gallery? Well, we just have to trade instant gratification for a little elbow grease.

The Places Field

Some of the most interesting data sources available to developers are those that contain places information. These are the databases that allow you to check into a business or some other venue with Facebook or Foursquare. When you arrive somewhere, your phone (or HTML5 capable browser) sends your location to a server-side API where a list of businesses, restaurants, and other public points of interest is then returned to the phone. You select the one where you are and check in.

While it’s unlikely that users will be checking into the Daisy’s Gone site, there’s another use of places data of which we might wish to take advantage. Consider the case of Meetup.com, where users who want to schedule a Meetup for their group are asked to select a location before saving the new event. The location is chosen from an auto-suggested list of places, similar to the check-in features of Foursquare and Facebook.

This is the functionality we’re going to add to our `Event` content type. Currently, our `Location` field is a text field into which event creators enter free-form text. In order to limit the locations to valid venues, we’ll instead use a field that auto-suggests possible places given a center point. To get this functionality, we’ll need a module. Fortunately for us, as of this writing, there is no such module in the Orchard Gallery. I say “fortunately” because we’re going to have the opportunity to create it in this chapter.

Getting Places Data

There are several places where we could get our places data. Facebook and Foursquare provide places data that is somewhat coupled with their respective services. Google provides places data, but doesn't allow that data to be rendered on a Bing Map (or any other non-Google Maps service). SimpleGeo provides places data, but it requires a paid developer key. For our needs, Yelp provides an API that will suit us well.

Yelp offers a REST API that is easily consumed from a .NET application. For our project, we'll use the *YelpSharp* library that's available on Nuget. We're not really concerned with the inner workings of a RESTful API client, so we'll just install the client and use it as a black box.

Module Code Generation

When we created our new theme in [Chapter 4](#), we used the *Orchard.CodeGeneration* module to create the initial directory structure for our theme's files. The new files were added to the *Themes* project that is part of the main Orchard solution. Unlike themes, each module is housed in its own project by default. Within the solution, these projects are organized under the "Modules" solution folder. On disk, these projects are stored in the *Modules* directory beneath the *Orchard.Web* directory.

Running the code generation command to create a new module will result in a new C# project being added to the solution within the *Modules* solution folder. If you're familiar with ASP.NET MVC, you'll probably recognize the structure, with familiar directories such as *Views* and *Controllers*. As we build out our module we'll explore the project in more detail.



Unlike the other modules that typically ship with Orchard, the code generated module projects are not web projects, but rather standard class library projects. What this difference means is that some of the context menus will not be web-centric (i.e., there is no "Add Controller").

As we did when creating our theme, we'll use PowerShell (or the standard command line) to create an Orchard session. If you closed out your session from the previous chapter, navigate to the *Orchard.Web* project's bin directory. Once there, execute *Orchard.exe* to initiate the session:

```
orchard> codegen module Contrib.PlacesField
Creating Module Contrib.PlacesField
Module Contrib.PlacesField created successfully
```

The default behavior of the `codegen` utility is to add the new module project to the solution. So when you return to your solution in Visual Studio, you'll be prompted to reload. Once you do, your new module project will be available. The module project is mostly comprised of empty directories and a few *web.config* files at this point. There's also a *Module.txt* file that will describe our module to the admin Dashboard. Again, we'll explore these project files as we start building out our new module.

We're going to create a content field, which is a little different than creating a more complex content part. Each has its own conventions. We'll examine content part creation in the next chapter on our way to creating a widget. However, some of the steps and concepts we'll use while creating our field will certainly provide us with transferable knowledge when it comes time to create a more complex module.

The Places Field Project

Before we add any code to the project, let's review the goal of our field to make sure we understand its requirements. At a high level, we simply want to limit the list of locations for an event to those that come from the Yelp API. To do so, we'll ask admin users (or content creators) to provide some details that will narrow the places search. After providing those pieces of information, users may then select a place from the list. That place and its details will then be saved and made available for rendering on the event pages (or any other content items where this field is used).

The Yelp API is highly flexible, allowing for searches by category, distance, search terms, and more. For the sake of keeping things simple our field won't support all possible query options, though it should be sufficiently generic to allow for reuse.

Places Field Model

We'll start off by considering the model for our field. Our model will represent the query and its result. To that end, we'll create a model class that has properties for some of the possible search parameters that we'll use when querying the API. This model class will be a subclass of Orchard's `ContentField`, which will provide our class with a few basic services that we'll see as we build our field.

Under the new `Contrib.PlacesField` project, create a new directory named *Fields* and add a class to that directory named *PlacesField.cs*. This class is just a plain old CLR object (POCO) that uses the persistence capabilities of the `Storage` property from its base class to save and retrieve the values set by content creators:

```
using Orchard.ContentManagement;

namespace Contrib.PlacesField.Fields {

    public class PlacesField : ContentField {

        public string PostalCode {
```

```

        get { return Storage.Get<string>("PostalCode"); }
        set { Storage.Set<string>("PostalCode", value); }
    }

    public string Category {
        get { return Storage.Get<string>("Category"); }
        set { Storage.Set<string>("Category", value); }
    }

    public string PlaceName {
        get { return Storage.Get<string>("PlaceName"); }
        set { Storage.Set<string>("PlaceName", value); }
    }

    public string PlaceLatLong {
        get { return Storage.Get<string>("PlaceLatLong"); }
        set { Storage.Set<string>("PlaceLatLong", value); }
    }
}

```

In our admin pages, we're going to collect information that is specific to a content *item* that is using our field, namely a postal code, category, and the selected place. We're also going to collect settings from content *type* creators that will affect how all content items built from a content type definition behave. We'll create the data model for our settings shortly and see how this all ties together when we update our *Event* type.

We've just defined a mixed data model that combines type-level settings with item-level data. The settings and data will both be used to render places by changing the behavior of the view appropriately. To bridge the gap between our disparate models and the denormalized view, we'll create a class that knows about both but more closely resembles the view. This approach is commonly known as the Model-View-ViewModel (MVVM) pattern, and it is frequently used in ASP.NET MVC applications.

Create a new folder named *ViewModels* and add a file named *PlacesFieldViewModel.cs*. This class will look very similar to our actual model class, but it won't concern itself with persistence. It will also have a simple string property for our category IDs and two additional properties for keeping track of our view options (the global settings):

```

namespace Contrib.PlacesField.ViewModels {

    public class PlacesFieldViewModel {

        public string Name { get; set; }

        public string PostalCode { get; set; }

        public string Category { get; set; }

        public string PlaceName { get; set; }

        public string PlaceLatLong { get; set; }
    }
}

```



```

        public bool ShowLink { get; set; }

        public bool ShowMap { get; set; }

    }
}

```

As hinted at, we're also going to include an option that will allow admins to decide which way to render the selected place on content pages. We'll provide options for displaying only the name of the place, displaying the name with a link to Bing Maps, or with an embedded Bing Map (using an `iframe` element, not the widget).

The way we'll represent these options in our model will be to create a settings class. This class is just another POCO and will be used by the admin pages that manage content types that use our field. We'll include a simple enumeration for the options. Create a new folder named *Settings* and add a file *PlacesFieldSettings.cs* with the following content (you can include the `enum` definition in this file):

```

namespace Contrib.PlacesField.Settings {

    public enum PlacesFieldDisplayOptions {
        NameOnly,
        NameAndLinkToMap,
        NameAndEmbeddedMap
    }

    public class PlacesFieldSettings {

        public PlacesFieldDisplayOptions DisplayOptions { get; set; }

    }

}

```

Drivers

Now that we've created our supporting data classes, it's time to write some code to take care of the actual rendering and saving of our field data. The class that has these responsibilities is known as a "driver." We'll also see drivers when we build a content part in [Chapter 6](#).

If you're familiar with ASP.NET MVC, you could think of drivers as being analogous to controller classes that are tightly coupled to a module. Drivers will be responsible for rendering our field's admin (when creating content) and content item (when displaying content) templates.

Create a new directory named *Drivers* and add a file named *PlacesFieldDriver.cs*. This new class will extend `ContentFieldDriver`, which will allow Orchard to recognize that this module is in fact a content field. There are three methods we'll override in this

class, which are discussed next. Though it certainly looks complex, the `Display` and `Editor` methods are simply either passing data to a view or reading data back from a form post:

```
using Orchard.ContentManagement.Drivers;
using Contrib.PlacesField.Settings;
using Orchard.ContentManagement;
using Contrib.PlacesField.ViewModels;

namespace Contrib.PlacesField.Drivers {

    public class PlacesFieldDriver : ContentFieldDriver<Fields.PlacesField> {
        //methods shown below
    }

}
```

The `Display` method begins by retrieving the saved settings for our field (we'll write code to save these shortly). The `ContentField` class that we extended to create our `PlacesField` provides the functionality to retrieve these settings. After getting the settings, the `ContentShape` method is used to return a shape to the view that will be used for display.

Note that this code won't compile until we complete the rest of the dependent code we write later in this chapter, so don't worry if you see several missing references.

Recall that the content that we bind to our views is rendered as a special dynamic type called a shape. The lambda expression provided as our third argument to `ContentShape` is creating the actual shape that will be used as the model for our `PlacesField` view:

```
protected override DriverResult Display(ContentPart part,
                                       Fields.PlacesField field,
                                       string displayType,
                                       dynamic shapeHelper) {

    var settings = field.PartFieldDefinition
        .Settings
        .GetModel<PlacesFieldSettings>();

    return ContentShape("Fields_Contrib_Places",
        field.Name,
        s => s.Name(field.Name)
            .PlaceName(field.PlaceName)
            .PlaceLatLong(field.PlaceLatLong)
            .ShowLink(settings.DisplayOptions ==
                PlacesFieldDisplayOptions.NameAndLinkToMap)
            .ShowMap(settings.DisplayOptions ==
                PlacesFieldDisplayOptions.NameAndEmbeddedMap)
        );
}
```



C# 4.0 introduced the *dynamic* keyword, which when applied to a variable, instructs the compiler not to check for compile-time correctness of properties and methods. In other words, you are letting the compiler know that everything will evaluate correctly at runtime. Orchard makes heavy use of this dynamic feature (e.g., shapes). Without C# 4.0, Orchard could not exist as it's written today.

The first `Editor` method is used to load our field into the dashboard editor forms for content types using our field (i.e., `Create→New→Event`). As was the case with `Display`, `Editor` also begins by retrieving the saved settings. It then composes a `ViewModel` instance from both settings and persisted field values. `ContentShape` is again called, but in this case, the `EditorTemplate` method of the dynamic `shapeHelper` is used to load our editor template and provide our `ViewModel` instance as the view's `Model` property:

```
protected override DriverResult Editor(ContentPart part,
                                     Fields.PlacesField field,
                                     dynamic shapeHelper) {

    var settings = field.PartFieldDefinition
        .Settings.GetModel<PlacesFieldSettings>();
    var viewModel = new PlacesFieldViewModel {
        Name = field.Name,
        Category = field.Category,
        PostalCode = field.PostalCode,
        ShowLink = settings.DisplayOptions
            == PlacesFieldDisplayOptions.NameAndLinkToMap,
        ShowMap = settings.DisplayOptions
            == PlacesFieldDisplayOptions.NameAndEmbeddedMap,
        PlaceName = field.PlaceName,
        PlaceLatLong = field.PlaceLatLong
    };

    return ContentShape("Fields_Contrib_Places_Edit",
        () => shapeHelper.EditorTemplate(
            TemplateName: "Fields/Contrib.Places",
            Model: viewModel,
            Prefix: getPrefix(field, part)
        ));
}
```

The second `Editor` method handles the post-back for our field when a user saves a content item that was defined to use our new field. After saving changes and mapping posted values to our field instance, the edit template is redisplayed (by way of the content item's editor form):

```
protected override DriverResult Editor(ContentPart part,
                                     Fields.PlacesField field,
                                     IUpdateModel updater,
                                     dynamic shapeHelper) {

    var viewModel = new PlacesFieldViewModel();
```

```

        if (updater.TryUpdateModel(viewModel,
                                   getPrefix(field, part), null, null)) {

            var settings = field.PartFieldDefinition
                .Settings.GetModel<PlacesFieldSettings>();

            field.Category = viewModel.Category;
            field.PostalCode = viewModel.PostalCode;
            field.PlaceName = viewModel.PlaceName;
            field.PlaceLatLong = viewModel.PlaceLatLong;

            viewModel.ShowLink = settings.DisplayOptions
                == PlacesFieldDisplayOptions.NameAndLinkToMap;
            viewModel.ShowMap = settings.DisplayOptions
                == PlacesFieldDisplayOptions.NameAndEmbeddedMap;

        }

        return Editor(part, field, shapeHelper);
    }
}

```

The private `getPrefix` method is used to create unique column values in the database for our field. It's a convention to define this method, not a requirement:

```

private static string getPrefix(ContentField field,
                               ContentPart part) {
    return (part.PartDefinition.Name + "." + field.Name)
        .Replace(" ", "_");
}

```

Field Templates

Next we're going to create some templates for our field. We'll need both an editor template and a standard (non-admin) view template. We'll start with the view template, because it is reasonably straightforward. In the *Views* directory, create a new directory named *Fields* and add to it a file named *Contrib.Places.cshtml*:

```

@using Orchard.Utility.Extensions

@{
    string name = Model.Name;

    string mapLink =
        "http://www.bing.com/maps/?v=2&cp=" +
        @Model.PlaceLatLong +
        "&lvl=12&dir=0&sty=r&where1=" +
        @Model.PlaceLatLong.ToString().Replace("~", ",");

    string iframeSource =
        "http://www.bing.com/maps/embed/?v=2&cp=" +
        @Model.PlaceLatLong +
        "&lvl=15&dir=0&sty=r&where1=" +
        @Model.PlaceLatLong.ToString().Replace("~", ",") +

```

```

        "&form=LMLTEW&pp=" + @Model.PlaceLatLong +
        "&emid=30e5aeb2-f963-4089-e9b3-9bd5dfe07759";
    }

    <p class="text-field">
        @name.CamelFriendly():
        @if (!Model.ShowLink && ! Model.ShowMap) {
            <text>@Model.PlaceName</text>
        }
        @if (Model.ShowLink) {
            <a href="@mapLink" target="_blank">@Model.PlaceName</a>
        }
        @if (Model.ShowMap) {
            <text>@Model.PlaceName</text>
            <div id="mapviewer">
                <iframe id="map" scrolling="no"
                    width="400" height="300" frameborder="0"
                    src="@iframeSource">
                </iframe>
            </div>
        }
    </p>

```

After importing the `Orchard.Utilities.Extensions` namespace, we define three variables to be used in the rendering of our field. Next, the display option chosen by the creator of the content type is interrogated to determine how we should display the place on our content item (e.g., an `Event` item). The logic here is fairly straightforward.



The `CamelFriendly` extension method required declaring the `string` name variable since we can't use extension methods with dynamic expressions such as `Model.Name`.

Next we'll create the editor templates. In the *Views* directory, create a new directory named *EditorTemplates* with a subdirectory named *Fields*. Add to that directory a file named *Contrib.Places.cshtml*. This template will provide a simple UI for selecting a category, inputting a postal code and then selecting the place. We'll skip validation for the sake of keeping the form from getting more complicated:

```

@using Orchard.Utility.Extensions
@model Contrib.PlacesField.ViewModels.PlacesFieldViewModel
@{
    Style.Require("jQueryUI_Orchard");

    Script.Require("jQuery");
    Script.Require("jQueryUtils");
    Script.Require("jQueryUI_Core");
    Script.Require("jQueryUI_Widget");
    Script.Require("jQueryUI_Autocomplete");
}
<fieldset>

```

```

<label for="@Html.FieldIdFor(m => Model.PlaceName)">
    @Model.Name.CamelFriendly()</label>
<p />
<label class="forpicker"
    for="@Html.FieldIdFor(m => Model.Category)_CategoriesSelector">Category</label>
<input type="text"
    id="@Html.FieldIdFor(m => Model.Category)_CategoriesSelector"
    class="text"/>
@Html.HiddenFor(m => m.Category)

<p />
<label class="forpicker"
    for="@Html.FieldIdFor(m => Model.PostalCode)">Postal Code</label>
@Html.EditorFor(m => m.PostalCode)

<p />
<label class="forpicker"
    for="@Html.FieldIdFor(m => Model.PlaceName)">Place</label>
@Html.EditorFor(m => m.PlaceName)
@Html.HiddenFor(m => m.PlaceLatLong)

</fieldset>

@using (Script.Foot())
{
<script type="text/javascript">
    function split(val) {
        return val.split(/,\s*/);
    }
    function extractLast(term) {
        return split(term).pop();
    }

    $(function () {

        $("#@Html.FieldIdFor(m => Model.Category)_CategoriesSelector")
            .autocomplete({
                source: '@Url.Content("~/Admin/PlacesField/Yelp/Categories")',
                minLength: 2,
                select: function (event, ui) {
                    $("#@Html.FieldIdFor(m => Model.Category)")
                        .val(ui.item.value);
                    $("#@Html.FieldIdFor(m => Model.Category)_CategoriesSelector")
                        .val(ui.item.label);
                    return false;
                }
            });

        $("#@Html.FieldIdFor(m => Model.PlaceName)").focus(function () {

            $("#@Html.FieldIdFor(m => Model.PlaceName)").autocomplete({
                source: '@Url.Content("~/Admin/PlacesField/Yelp/Places")'
                    + '?categories='
                    + $("#@Html.FieldIdFor(m => Model.Category)").val()

```

```

        + '&postalCode='
        + $("#@Html.FieldIdFor(m => Model.PostalCode)").val(),
        minLength: 1,
        select: function (event, ui) {
            $("#@Html.FieldIdFor(m => Model.PlaceLatLng)")
                .val(ui.item.value);
            $("#@Html.FieldIdFor(m => Model.PlaceName)")
                .val(ui.item.label);
            return false;
        }
    });

});

});
</script>
}

```

It might look like there's a lot happening in this view, but it's relatively straightforward. We're using the jQuery UI Autocomplete plugin to render our short list of Yelp categories. The autocomplete code for categories and places is in the script block at the bottom of our Razor template.

How the Autocomplete plugin works isn't important for understanding how to create a field, so we'll skip a detailed explanation. The important thing to understand is that we've created a form with three fields for collecting information relevant to our places selection. We use hidden fields to save category and places data that is persisted, but not displayed to the user (category ID, latitude, and longitude).

For Orchard to know where to put our field, we'll need to create a *Placement.info* file at the root of our module project. The entries in this file will allow our templates to appear in both admin forms and content displays. We saw how placement files work in Displaying Content. Without this file, your field will not display:

```

<Placement>
  <Place Fields_Contrib_Places_Edit="Content:2.5"/>
  <Place Fields_Contrib_Places="Content:2.5"/>
</Placement>

```

Settings

Next we'll take care of persisting the settings. We're going to need to hook into admin editor events in order to save our settings. This code is mostly boilerplate and you'll find yourself copying, pasting, and modifying it as you create your own fields that require settings. Start by creating a new file named *PlacesFieldEditorEvents.cs* in the *Settings* directory that we created earlier.

In *PartFieldEditor*, we help Orchard load our settings form within the context of a content type editor form (we'll see where this form is rendered before the end of the chapter). *PartFieldEditorUpdate* is used to handle the actual saving of our settings.

These two methods are similar to `Editor` and `EditorTemplate` in our field's driver class:

```
using System.Collections.Generic;
using Contrib.PlacesField.Settings;
using Orchard.ContentManagement;
using Orchard.ContentManagement.Metadata;
using Orchard.ContentManagement.Metadata.Builders;
using Orchard.ContentManagement.Metadata.Models;
using Orchard.ContentManagement.ViewModels;

public class PlacesFieldEditorEvents : ContentDefinitionEditorEventsBase {

    public override IEnumerable<TemplateViewModel>
        PartFieldEditor(ContentPartFieldDefinition definition) {
        if (definition.FieldDefinition.Name == "PlacesField") {
            var model = definition.Settings.GetModel<PlacesFieldSettings>();
            yield return DefinitionTemplate(model);
        }
    }

    public override IEnumerable<TemplateViewModel> PartFieldEditorUpdate(
        ContentPartFieldDefinitionBuilder builder, IUpdateModel updateModel) {
        var model = new PlacesFieldSettings();
        if (builder.FieldType != "PlacesField") {
            yield break;
        }

        if (updateModel.TryUpdateModel(
            model, "PlacesFieldSettings", null, null)) {
            builder.WithSetting("PlacesFieldSettings.DisplayOptions",
                               model.DisplayOptions.ToString());
        }

        yield return DefinitionTemplate(model);
    }
}
```

To create the editor template for our settings, create a new file named *PlacesFieldSettings.cshhtml* in a new directory under *Views* named *DefinitionTemplates*. Note that this template will appear when creating content *types*, not content *items*. It's the creator of the type, not the content item that will set this value. In our template, we'll simply render an HTML select list from the values in our `PlacesFieldDisplayOptions` enumeration:

```
@model Contrib.PlacesField.Settings.PlacesFieldSettings
@using Contrib.PlacesField.Settings;

<fieldset>
    <label for="@Html.FieldIdFor(m => m.DisplayOptions)"
        class="forcheckbox">Display options</label>
    <select id="@Html.FieldIdFor(m => m.DisplayOptions)"
        name="@Html.FieldNameFor(m => m.DisplayOptions)">
        @Html.SelectOption(PlacesFieldDisplayOptions.NameOnly,
            Model.DisplayOptions == PlacesFieldDisplayOptions.NameOnly,
            "Name only")
    </select>
</fieldset>
```



```

        @Html.SelectOption(PlacesFieldDisplayOptions.NameAndLinkToMap,
            Model.DisplayOptions
            == PlacesFieldDisplayOptions.NameAndLinkToMap,
            "Name and Link to Map")
        @Html.SelectOption(PlacesFieldDisplayOptions.NameAndEmbeddedMap,
            Model.DisplayOptions
            == PlacesFieldDisplayOptions.NameAndEmbeddedMap,
            "Name and Embedded Map")
    </select>

    @Html.ValidationMessageFor(m => m.DisplayOptions)

</fieldset>

```

Controllers

One final important detail is the data. So far, we've created a pretty complex field, but it won't actually do anything until we provide it a way to get data from the Yelp API. As I previously mentioned, the API is a simple REST API. However, we can't query it directly from our views, because browser security restrictions won't allow AJAX requests to other servers.

To solve this problem, we'll have to create a solution that runs on our Orchard site. Specifically, we're going to create an ASP.NET MVC controller class that will handle the AJAX requests that are generated by the jQuery Autocomplete plugin. If you revisit the code for our editor view, you'll see the URLs that we're going to call.

URL to retrieve Yelp Categories

/Admin/PlacesField/Yelp/Categories

URL to retrieve Yelp Places

/Admin/PlacesField/Yelp/Places

If you're familiar with MVC, you probably recognize that we're going to create a controller named `YelpController` with two action methods, `Categories` and `Places`. We're going to examine the controller shortly, but for now we need to setup a route to tell Orchard and MVC how to map our requests. Orchard will automatically map all routes by finding classes that implement `IRouteProvider`. Save this file as *Routes.cs* at the root of the module project:

```

using System.Collections.Generic;
using System.Web.Mvc;
using System.Web.Routing;
using Orchard.Mvc.Routes;

public class Routes : IRouteProvider {
    public void GetRoutes(ICollection<RouteDescriptor> routes) {
        foreach (var routeDescriptor in GetRoutes())
            routes.Add(routeDescriptor);
    }

    public IEnumerable<RouteDescriptor> GetRoutes() {

```

```

        return new[] {
            new RouteDescriptor {
                Priority = 5,
                Route = new Route(
                    "Admin/PlacesField/{controller}/{action}",
                    new RouteValueDictionary {
                        {"area", "Contrib.PlacesField"},
                        {"controller", "Yelp"},
                        {"action", "Places"}
                    },
                    new RouteValueDictionary(),
                    new RouteValueDictionary {
                        {"area", "Contrib.PlacesField"}
                    },
                    new MvcRouteHandler())
            };
        }
    }
}

```



The area defined in the `RouteValueDictionary` must be the name of the module and not an area that is defined as a URL part as is expected with typical MVC routing.

The Controller class is a standard MVC controller. Save this file as *YelpController.cs* in the *Controllers* directory that the code generation tools created for the module project:

```

using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using YelpSharp;
using YelpSharp.Data.Options;

public class YelpController : Controller {

    private readonly Yelp _client;

    public YelpController() {
        var options = new Options()
        {
            AccessToken = "<YOUR_ACCESS_TOKEN>",
            AccessTokenSecret = "<YOUR_ACCESS_TOKEN_SECRET>",
            ConsumerKey = "<YOUR_ACCESS_CONSUMER_KEY>",
            ConsumerSecret = "<YOUR_CONSUMER_SECRET>"
        };

        _client = new Yelp(options);
    }

    public YelpController(Yelp client) {
        _client = client;
    }
}

```

```

public ActionResult Places(string postalCode,
                           string category, string term) {

    var so = new SearchOptions() {
        LocationOptions = new LocationOptions() {
            location = postalCode
        },
        GeneralOptions = new GeneralOptions(){
            category_filter = category,
            term = term
        }
    };
    var results = _client.Search(so);

    return Json(results.businesses.Select(
        p => new { value = p.location.coordinate.latitude + "~"
                    + p.location.coordinate.longitude,
                    label = p.name }).ToArray()
        , JsonRequestBehavior.AllowGet);
}

public ActionResult Categories(string postalCode,
                              string categories, string term) {

    var categoryList = new List<string> {
        "Restaurants", "Nightlife", "Arts"
    };

    var results = string.IsNullOrEmpty(term) ?
        categoryList : categoryList
            .Where(c =>
                c.ToLower()
                .Contains(term.ToLower()));

    return Json(results.Select(
        c => new { value = c.ToLower(), label = c })
        .ToArray(), JsonRequestBehavior.AllowGet);
}
}

```

Most of the logic in the controller deals with retrieving data from the Yelp service. For this code to work, you'll need to add the **YelpSharp** Nuget package to your field project:

PM> Install-Package YelpSharp



You'll need to get free API credentials from http://www.yelp.com/developers/getting_started. These API credentials then need to be entered into the constructor of the controller, where the **Yelp** client is instantiated. If you forget this step, your field won't work. Request the 2.0 API on the API access page. It will give you all four keys that you'll require.

Again, this class is a standard MVC controller. There isn't anything Orchard-specific to consider here. Basically, all that's happening in each action (public method that returns an `ActionResult`) is that we're querying the Yelp API method and returning a JSON serialized result of the Yelp data.

The `YelpSharp` client takes care of deserializing the Yelp response to POCO classes that we'll then serialize to JSON formats that our autocomplete textboxes will consume. We use a LINQ projection to create a JSON structure that is friendly to the `Autocomplete` plugin.

Module Metadata

The final step is to set the metadata for our field in *Module.txt*. These settings are straightforward and include author and module description. If we had multiple fields in our project we could list multiple features, which could be enabled or disabled separately. Finally, we'll also declare a set of modules upon which our field is dependent:

```
Name: Places Field
AntiForgery: enabled
Author: John Zablocki
Website: http://dllhell.net
Version: 1.0
OrchardVersion: 1.3
Description: The Places Field allows for location lookups
Features:
  Contrib.PlacesField:
    Name: Places Field
    Description: Places fields.
    Category: Fields
    Dependencies: Orchard.jQuery, Common, Settings
```

Using the Places Field

Now that we've created our field, it's time to compile the project and enable the module. Build your project to make sure you've successfully added all the `using` directives and Nuget references that our code demands. Then return to your Orchard command-line session and enable the new module:

```
PM> feature enable Contrib.PlacesField
Enabling features Contrib.PlacesField
Places Field was enabled
```

Creating Content with the Places Field

Now that our field is coded and enabled, it's time to test it out. Return to the Dashboard and select Content→Content Types and select "Edit" on the listing for "Event." Click "Remove" on the field listing for the existing "Location" field. Then click "Add Field" and select the field type "Places Field." Name the field "Event Location," accepting the

default technical name as usual. Click “Save” and the “Event Location” field will appear in the list of fields.



We could add several `PlacesField` elements to our `Event` content type if we wanted to, which is why we had to prefix our autocomplete client side IDs with field specific IDs. In other words, when you select a place for `PlacesFieldA` it won’t get mixed up with `PlacesFieldB`.

Click “>” next to the field name to be presented with our settings editor. It’s here while defining our content type that we’ll set the display option for this field, affecting all items created from this content type definition. From the select list with our display options (Figure 5-1), choose “Name and Embedded Map” and click Save.

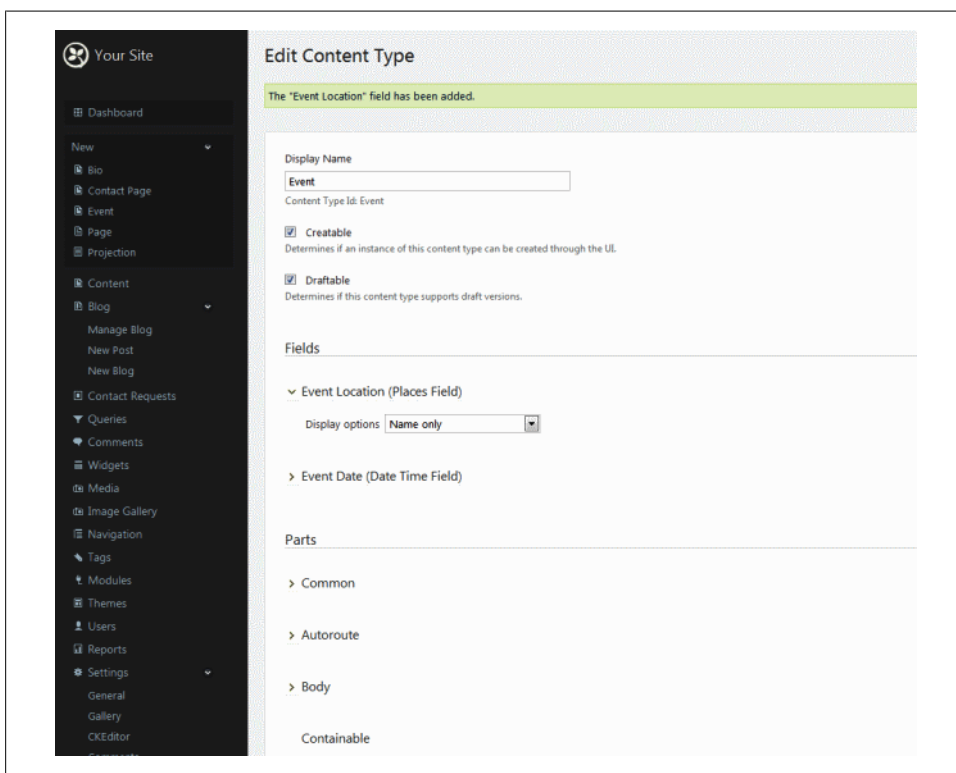


Figure 5-1. The Places Field settings template

If you were to browse to the “Events” listing page on our site, you’d see a Bing Maps error because we haven’t selected a place and we set our field to render a map. You should also see a Bing Maps error on your content item listing (Content→Content Items), because Orchard uses the same field display template to show a preview of your content item in the admin content listing page. That template includes the map code.

To fix these errors, we'll simply select a place. Click Content→Content Items and then click “Edit” on the event listing you wish to edit. As you start typing in the “Category” field, you should see a list of categories that match your input. After selecting a category and entering a zip code, go ahead and search for a place by typing in the “Place” textbox.



Yelp provides a full list of available categories that can be used to filter out results at http://www.yelp.com/developers/documentation/category_list. This module limits searches to only those categories that seem likely to host live music. A more robust solution would also provide some better caching and filtering of results.

Figure 5-2. The Places Field editor template

After you’ve selected a place, click “Publish Now” and then return to the site and refresh the “Events” page. You should now see the event location displayed with both the name of the place you selected and an embedded Bing map (Figure 5-3).

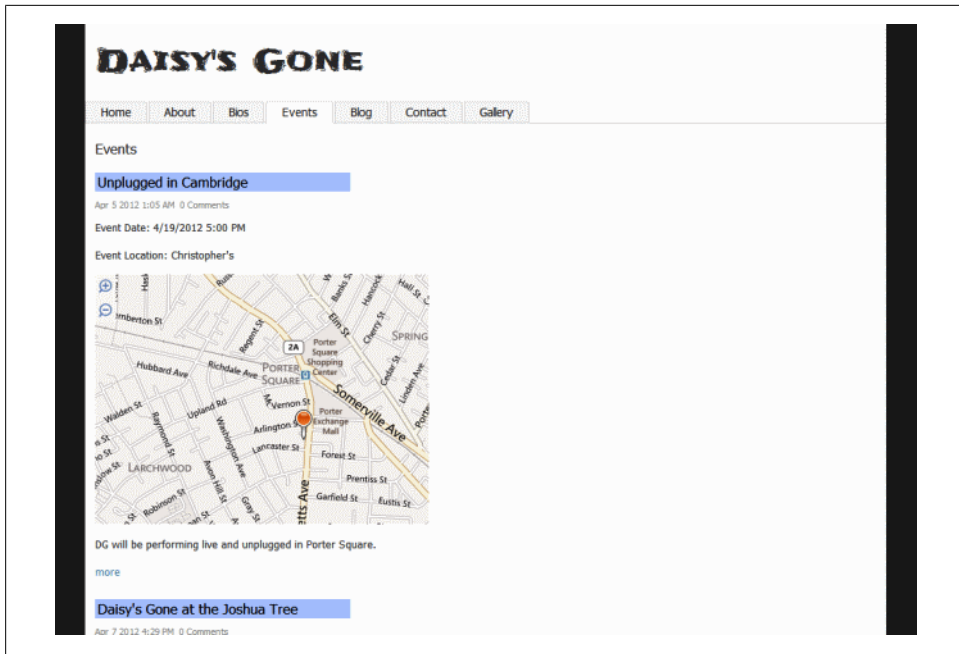


Figure 5-3. The Places Field displayed with a Bing map

Displaying the Places Field

If you click over to the home page (or any page that isn't the "Events" page) you'll see that our event location map is overtaking the space it's been allotted in the `AsideSecond` zone. Our "Upcoming Events" HTML widget is trying to use the embedded map in a smaller space than our event listing or event detail pages.

In [Chapter 3](#), we learned how to customize the display of fields and other types of content. Our own fields may be customized in the same way. Copy `Contrib.Places.cshtml` from the `Fields` directory under the `Views` directory in our module. Paste it in the `Views` directory of our "DaisysTheme" theme and rename it `Fields.Contrib.Places.cshtml`. Change the height and width of the `iframe` that includes the map and refresh the "Events" page. You'll see that the embedded map has changed to reflect your chosen size:

```
<iframe id="map" scrolling="no" width="300" height="200"
  frameborder="0" src="@iframeSource">
```

We also could choose to remove the entire `Contrib.Places` field from any summary listing by updating the `Placement.info` file for our template ([Figure 5-4](#)):

```
<Match DisplayType="Summary">
  <Place Fields_Contrib_Places="-" />
</Match>
```

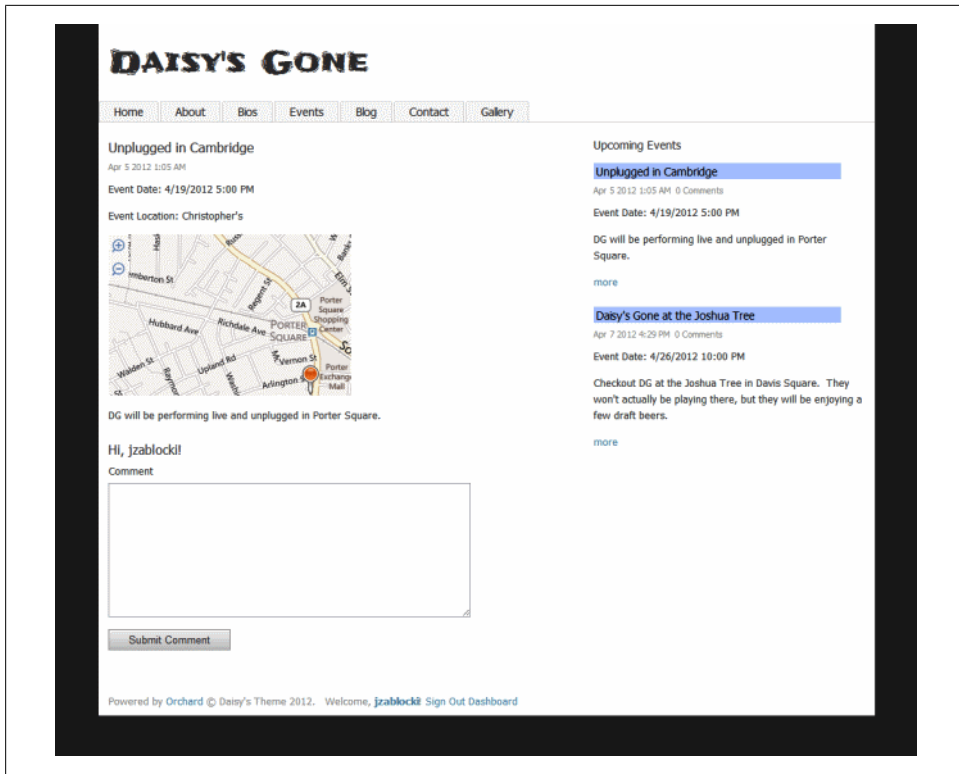


Figure 5-4. Using placement to remove a field

Summary

Though we took a few shortcuts, our `PlacesField` is a reusable Orchard component. We could easily share it in other projects simply by copying the project to another Orchard solution. A better solution, which we'll learn about in a later chapter, would be to package it up using the Orchard tools and submit it to the Orchard gallery (or some private file share).

Content fields are just one type of module we can create in Orchard. In the next chapter, we're going to create a more complex (in terms of behavior) module, namely a content part that we'll turn into a widget. Much of what we learned during our field exercise will still apply, but there will be several new concepts to learn.

Creating Widgets

In [Chapter 5](#), we walked through the process of creating a content field. In order to be used, our field had to be added to a content type. We couldn't just add it to a zone in a template and have it appear. For that behavior, we'll need a widget. Like fields, widgets are a type of Orchard module. In this chapter, we'll walk through the process of creating a widget that we can attach to all or some pages in our site.

Content Parts

Content parts are reusable pieces of functionality (UI or behavior) that are added to content types. When we created the **Event** content type, we added the **Containable** content part to its definition, which allowed events to be contained in projection pages. The **Body** content part allowed us to include an HTML body in our **Bio** content type.

Even though a widget is a content type and not a part, creating a widget is effectively the same process as creating a content part. However, we'll include some additional metadata to make the part behave like a widget and not a part. In other words, once a part becomes a widget we can add it as content in a zone.

In this chapter, we're going to create a module for embedding videos on our site. At the time of this writing, there exists only a single YouTube module in the Orchard Gallery. That module is a field, which again requires a content type to be defined with that field in order for it to be used with content items. What we want instead is a widget that we can attach to different pages, regardless of the content type for those pages.

The JW Player Widget

YouTube offers a very simple interface for embedding its videos into other sites. We could easily create a YouTube video widget that does nothing more than replace HTML `iframe` tag attribute values (i.e., should precede `src`, `height`, `width`) with our video's details. However, that approach would be a bit limiting in that we might want to host our own videos or pull in content from video sites other than YouTube.

Instead of creating a standalone YouTube module, we're going to create a widget that uses the JW Player for Flash and HTML5 for media rendering. The JW Player is an extensible media player that uses Flash and JavaScript to provide support for skinning, plugins, playlists, and customizable controls. Our widget will provide two services for sites that use it. It'll package up the JavaScript and Flash files and build the JavaScript to embed on our pages.

Creating the Module

Once again we'll return to PowerShell (or the standard command line) to create an Orchard session. If you don't have an Orchard session open, navigate to the *Orchard.Web* project's bin directory. Once there, execute *Orchard.exe* to initiate the session. We're going to use the code generation tools to create the skeleton for our module project. Though our widget will be decidedly different than our field, the starting project should look similar to what we saw when we used the `codegen` tools to create our `Contrib.PlacesField` project:

```
orchard> codegen module JWPlayer
Creating Module JWPlayer
Module JWPlayer created successfully
```

As was the case with our places field from [Chapter 5](#), the solution will require a reload after you run this command. Once reloaded, you'll see the skeleton of a module project under the Modules solution folder. If you expand it, you'll again see the placeholder directories that we saw in the last chapter.

The process of creating a widget is not unlike creating a field, but the supporting infrastructure for a widget module is more complex and will require a few additional steps. As we did with our field module, we're going to start out by considering our model classes.

The JW Player supports many customizable features, from simply setting the media file to be played to setting behaviors such as whether to auto-start or repeat. To keep our module from getting too complex, we'll pare down the features to a minimal, yet still interesting set. We'll need the obvious properties for filename, width, and height. We'll also support the behaviors mentioned previously. Beyond that, you could visit <http://www.longtailvideo.com> to see the additional properties that could be included.



The JW Player is an open source project that is free to use for non-commercial sites. If you're interested in using this module or the player on a commercial site, you will need to purchase a commercial license from LongTail Video.

JW Player Model

The codegen utility created a directory named *Models* in our new module project. We're going to add two files to this directory. Start by creating a file named *JWPlayerPartRecord.cs*. This is the class that will be used by Orchard to create the database records for our widget. It's basically a POCO with properties that will map to the columns in our widget's table. Each of these properties represents a piece of data that we'll collect from content creators who use our widget. We'll see how the data model comes together shortly:

```
using Orchard.ContentManagement.Records;

namespace JWPlayer.Models {
    public class JWPlayerPartRecord : ContentPartRecord {
        public virtual string PlayerSource { get; set; }

        public virtual int Height { get; set; }

        public virtual int Width { get; set; }

        public virtual string MediaFile { get; set; }

        public virtual bool AutoStart { get; set; }

        public virtual bool Repeat { get; set; }
    }
}
```

There are two important pieces to this class. The first is that it extends Orchard's *ContentPartRecord* class, which simply provides its subclasses with an *Id* property. The second is that all properties are marked *virtual*. The reason for this modifier is that Orchard uses the *object relational mapper* (ORM) NHibernate for data access. NHibernate will dynamically generate a proxy class based on *ContentPartRecord* classes and requires *virtual* properties to be able to do so.

Next, we're going to create a class that is similar to the *ViewModel* class we created for our field in the previous chapter. It's similar in that it will be the class that's used to bind to admin forms. Unlike our *ViewModel*, though, this class will have a formal requirement of extending Orchard's *ContentPart* class. Add a new file to the *Models* directory named *JWPlayerPart.cs*:

```
using System.ComponentModel.DataAnnotations;
using Orchard.ContentManagement;

namespace JWPlayer.Models {
    public class JWPlayerPart : ContentPart<JWPlayerPartRecord> {

        [Required]
        public string PlayerSource {
            get { return Record.PlayerSource; }
            set { Record.PlayerSource = value; }
        }
    }
}
```

```

    [Required]
    public int Height {
        get { return Record.Height; }
        set { Record.Height = value; }
    }

    [Required]
    public int Width {
        get { return Record.Width; }
        set { Record.Width = value; }
    }

    [Required]
    public string MediaFile {
        get { return Record.MediaFile; }
        set { Record.MediaFile = value; }
    }

    public bool AutoStart {
        get { return Record.AutoStart; }
        set { Record.AutoStart = value; }
    }

    public bool Repeat {
        get { return Record.Repeat; }
        set { Record.Repeat = value; }
    }
}
}

```

It's common, though not required, that this new `ContentPart` class will simply map each of the properties of the `ContentPartRecord` to a property with the same name. In our case, we're simply going to ask for and receive input values that correspond to a column in our widget's table. If our UI was required to be more complex, then our properties might not map directly to the properties. Notice, too, that we're including validation attributes on our properties. Orchard's admin pages will automatically enforce these rules when attempting to save new instances of our widget.

Database Migrations

We've discussed that NHibernate is used for data access and will use our `JWPlayerPartRecord` class to store and retrieve the data for our widget. However, we haven't actually seen how that table gets created. To create our table, we're going to need to create a data migration.

Data migrations are a concept borrowed from frameworks such as Ruby on Rails. The basic idea is that we version our database through a series of migration scripts. Each modification to the database (new table, dropped column, etc.) is a new version of the

database. Each version has a migration associated with it, which is either a SQL script or a migration class that often uses some sort of *domain specific language* (DSL).

In our case, we're going to version modules and not the entire database. You'll define a schema version for your module and move it up to newer versions. This versioning will be done through both convention and a simple DSL implemented by the Orchard framework. Return to your command-line Orchard session and run the following command:

```
orchard> codegen datamigration JWPlayer
Creating Data Migration for JWPlayer
Data migration created successfully in Module JWPlayer
```

As it did when we created the JWPlayer module, the `codegen` utility will force a solution reload. What's been added is a new file named *Migrations.cs* to the root of our JWPlayer module project (for which you should "Include In Project"). This generated file contains a class named `Migrations` with a single method named `Create`. When we eventually enable this module, Orchard will execute this method to create the initial schema for our module:

```
public class Migrations : DataMigrationImpl {

    public int Create() {

        // Creating table JWPlayerPartRecord
        SchemaBuilder.CreateTable("JWPlayerPartRecord", table => table
            .ContentPartRecord()
            .Column("PlayerSource", DbType.String)
            .Column("Height", DbType.Int32)
            .Column("Width", DbType.Int32)
            .Column("MediaFile", DbType.String)
            .Column("AutoStart", DbType.Boolean)
            .Column("Repeat", DbType.Boolean)
        );

        ContentDefinitionManager.AlterPartDefinition("JWPlayerPartRecord",
            builder => builder.Attachable());

        return 1;
    }
}
```

Our migration class extends the base migrations class, which provides access to the `SchemaBuilder` property. `SchemaBuilder`, which is an instance of Orchard's `Schema Builder` class, contains methods for creating our migrations (creating tables, dropping columns, adding foreign keys, etc.).

The `codegen` utility created a table definition for us based on our `JWPlayerPartRecord` property names. We also need to add the `AlterPartDefinition` call to tell Orchard that our content part may be attached to any content type. By convention, `Create` will be the method called when Orchard runs our module's migration for the first time (on

install and enable). `Create` returns 1, which sets the current schema version for our module to 1. We'll see shortly how a second version is created and executed.

Handlers and Drivers

During our exploration of themes and modules we've seen analogies to ASP.NET MVC. When writing content parts, these analogies still hold. In ASP.NET MVC, we can use `Filter` classes to hook into the execution of a request. Filters provide a mechanism for executing code before and after a request. Similarly, with our content parts, we can create a `ContentHandler` subclass that offers us the opportunity to hook into different points in our module's execution.

Create a new directory named `Handlers` to our new module project, and add to it a file named `JWPlayerHandler.cs`. While we could create a handler that overrides lifecycle events such as `OnCreating`, `OnCreated`, `OnActivating`, or `OnActivated`, our needs are simple. We'll include only a constructor with some plumbing code to instruct Orchard to wire up data access for our module using our `ContentPartRecord` implementation:

```
using JWPlayer.Models;
using Orchard.Data;
using Orchard.ContentManagement.Handlers;

namespace JWPlayer.Handler {
    public class JWPlayerHandler : ContentHandler {

        public JWPlayerHandler(IRepository<JWPlayerPartRecord> repository) {
            Filters.Add(StorageFilter.For(repository));
        }

    }
}
```

Like content fields, content parts will use driver classes to take care of the actual rendering of views and processing of admin data collection. Create a new directory named `Drivers` and add to it a class named `JWPlayerDriver.cs`:

```
using Orchard.ContentManagement.Drivers;
using JWPlayer.Models;
using Orchard.ContentManagement;

namespace JWPlayer.Drivers
{
    {
        public class JWPlayerDriver : ContentPartDriver<JWPlayerPart> {
            //methods shown below
        }
    }
}
```

The `Display` method will be used to render our widget in a zone. The `ContentShape` method will construct a shape to be bound to our view file. Our shape will simply have

properties that map one-to-one to our `JWPlayerPart` class. Orchard will call the `Display` method when a content item is rendered that uses our widget:

```
protected override DriverResult Display(JWPlayerPart part,
    string displayType, dynamic shapeHelper) {

    return ContentShape("Parts_JWPlayer",
        () => shapeHelper.Parts_JWPlayer(
            MediaFile: part.MediaFile,
            PlayerSource: part.PlayerSource,
            Width: part.Width,
            Height: part.Height,
            AutoStart: part.AutoStart,
            Repeat: part.Repeat));
}
```

The first `Editor` method is used to load the admin template for collecting the data that we'll use to configure our `JWPlayer` widget. Using the `EditorTemplate` method of the `dynamic shapeHelper`, we instruct Orchard on how to locate our editor template view file and to use a `JWPlayerPart` instance as the model for this view:

```
protected override DriverResult Editor(JWPlayerPart part,
    dynamic shapeHelper) {

    return ContentShape("Parts_JWPlayer_Edit",
        () => shapeHelper.EditorTemplate(
            TemplateName: "Parts/JWPlayer",
            Model: part,
            Prefix: Prefix));
}
```

The second `Editor` method is used to handle the save action when our widget is configured in the Dashboard. The `JWPlayerPart` parameter is bound to the values from the form and saved back to the database. After saving, the editor form is redisplayed, with the updated `JWPlayerPart` instance as its model:

```
protected override DriverResult Editor(JWPlayerPart part,
    IUpdateModel updater,
    dynamic shapeHelper) {

    updater.TryUpdateModel(part, Prefix, null, null);
    return Editor(part, shapeHelper);
}
```

Placement

Next, we're going to need to create our *Placement.info* file in the root of the project. Again, placement files are used by Orchard to determine where to display a widget (or other content types) relative to other widgets, fields, and parts. Review *Creating Content* or *Creating Modules* for more on `Placement.info`:

```

<Placement>
  <Place Parts_JWPlayer="Content:3"/>
  <Place Parts_JWPlayer_Edit="Content:3"/>
</Placement>

```

Enabling Our Module

We haven't finished our widget yet, but we've created enough of our project to test out the basic correctness of our plumbing. Make sure you compile your project and then return to the command line:

```

orchard> feature enable JWPlayer
Enabling features JWPlayer
JWPlayer was enabled

```

By enabling the feature, you've run the data migration. If you open the Dashboard and select Modules→Features, you'll see the JWPlayer enabled and uncategorized ([Figure 6-1](#)). We'll fix the metadata for our module a little later.

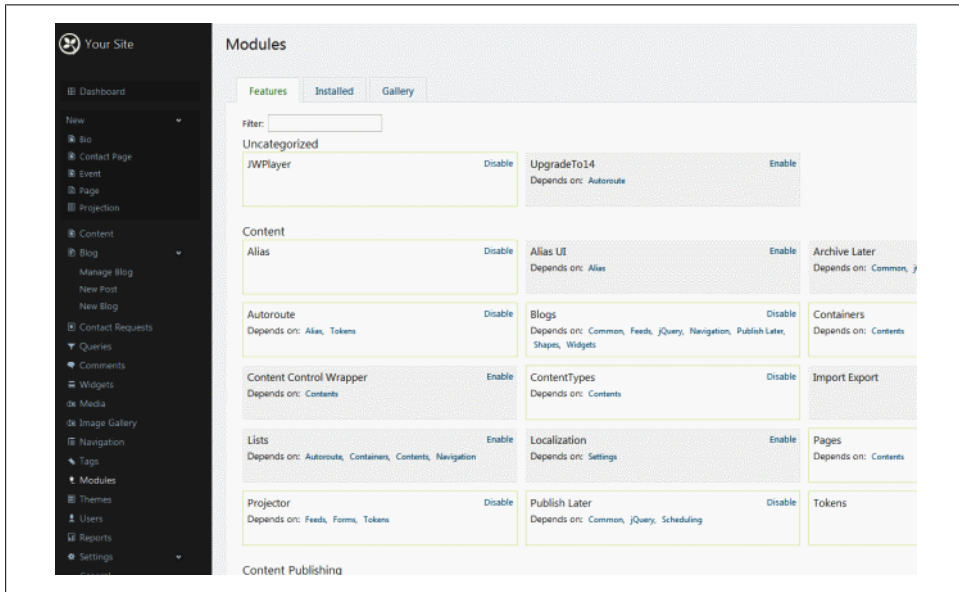


Figure 6-1. The enabled and uncategorized JWPlayer module

A Second Migration

If you click “Widgets” and then click “Add” on any of the zones, you might expect that you'd see our new widget in the list of possible choices, but it doesn't appear. We're going to fix that problem now. We'll need to modify *Migrations.cs* to include an update to our module that will allow it to behave as a widget:


```

public int UpdateFrom1() {
    ContentDefinitionManager.AlterTypeDefinition(
        "JWPlayerWidget", cfg => cfg
        .WithPart("JWPlayerPart")
        .WithPart("WidgetPart")
        .WithPart("CommonPart")
        .WithSetting("Stereotype", "Widget"));

    return 2;
}

```

Notice first, that the method name is `UpdateFrom1`. By convention, Orchard will know to run this method if the current schema version for our module is 1 (which it is after running `Create`). This method creates a new widget type by composing a new type from `JWPlayerPart`, `WidgetPart`, and `CommonPart`. In other words, the definition of our widget has properties from each of these three parts. Conceptually, this is similar to adding parts to our `Event` or `Bio` content types.

Compile your migration and return to the admin dashboard. Navigate to Modules→Features. You won't see any notification that a migration needs to be run for the `JWPlayer` module. That's because when Orchard detects that there's a new migration to run for a module, it runs it automatically. While this might sound a little scary, keep in mind that in all likelihood you would explicitly and manually install a newer version of the module. Orchard is just taking care to make sure the code and data are in sync.

The Widget Views

Before we try to add our widget to a zone, we first have to create the views for it. Our widget is going to encapsulate the third-party `JWPlayer` media player. As you might therefore expect, we're going to need to include some third-party files. Download the latest `JWPlayer` from <http://www.longtailvideo.com/players/jw-flv-player/>.

You're going to need to put a couple of files into your module project from the ZIP file you downloaded. Copy `jwplayer.js` into the *Scripts* directory. Create a new directory named *Flash* and add the `player.swf` file. You should also include both in your project from the Solution Explorer. You'll also need to copy the `web.config` file from the *Scripts* directory into the *Flash* directory so the static `player.swf` file will be properly served. I'm also going to add the `license.txt` file to the root for good measure.

Next, under the *Views* directory, create a *Parts* directory and a new file under that directory named `JWPlayer.cshtml`. This file will first include the `jwplayer.js` JavaScript file and then include a script block that will configure the player. The properties we set in the admin tool are embedded within this JavaScript in order to display the player as our content creators will specify:

```

@using Orchard.UI.Resources
@{ Script.Include("jwplayer.js")
    .AtLocation(ResourceLocation.Head); }

```

```

<div id='mediaspace'>This text will be replaced</div>

<script type='text/javascript'>
    jwplayer('mediaspace').setup({
        'flashplayer': '@Url.Content(Model.PlayerSource)',
        'file': '@Url.Content(@Model.MediaFile)',
        'autostart': '@Model.AutoStart',
        @if (Model.Repeat) {
            <text>'repeat' : 'always',</text>
        }
        'width': '@Model.Width',
        'height': '@Model.Height'
    });
</script>

```

In this view, we included the *jwplayer.js* script by using the `Include` method of our view's `Script` property. If instead we wanted to use the `Require` method to guarantee the script will load only once per page view, we need to create a class that implements `IResourceManifestProvider`. Create a new file named *ResourceManifest.cs* at the root of the module:

```

using Orchard.UI.Resources;

namespace JWPlayer {
    public class ResourceManifest : IResourceManifestProvider {

        public void BuildManifests(ResourceManifestBuilder builder) {
            var manifest = builder.Add();
            manifest.DefineScript("jwplayer").SetUrl("jwplayer.js");
        }
    }
}

```

After creating this file, we can now include our script file using the `Require` method, replacing the `Include` method we originally used previously:

```
@{ Script.Require("jwplayer").AtHead(); }
```

We also need to create our admin UI. Create a new directory named *EditorTemplates* under *Views*, and under that new directory create one named *Parts*. In the new *Parts* directory, create a file named *JWPlayer.cshtml*. This new view will be a simple editor form with fields for each of our `JWPlayerPart` properties. Again, because we decorated these properties with `Required` attributes from `System.ComponentModel.DataAnnotations`, we're able to include validation messages as well:

```

@model JWPlayer.Models.JWPlayerPart

<fieldset>
    <legend>JWPlayer Widget</legend>

    <div class="editor-label">
        Player Source
    </div>

```

```

<div class="editor-field">
    @Html.TextBoxFor(model => model.PlayerSource)
    @Html.ValidationMessageFor(model => model.PlayerSource)
</div>

<div class="editor-label">
    Height
</div>
<div class="editor-field">
    @Html.TextBoxFor(model => model.Height)
    @Html.ValidationMessageFor(model => model.Height)
</div>

<div class="editor-label">
    Width
</div>
<div class="editor-field">
    @Html.TextBoxFor(model => model.Width)
    @Html.ValidationMessageFor(model => model.Width)
</div>

<div class="editor-label">
    Media File
</div>
<div class="editor-field">
    @Html.TextBoxFor(model => model.MediaFile)
    @Html.ValidationMessageFor(model => model.MediaFile)
</div>

<div class="editor-label">
    Auto Start
</div>
<div class="editor-field">
    @Html.CheckBoxFor(model => model.AutoStart)
</div>

<div class="editor-label">
    Repeat
</div>
<div class="editor-field">
    @Html.CheckBoxFor(model => model.Repeat)
</div>

</fieldset>

```

Adding the Widget to a Zone

We're now ready to add this widget to a zone. We'll add this widget to the `AsideSecond` zone, where we currently have our "Upcoming Events" list. We'll also add a layer rule so that it appears only on the events page. We'll use this new widget to display videos from past events.

Return to the Dashboard and click “Widgets.” Click the “Add new layer...” link. Name the new layer “Events” and add the rule url ‘~/events’ and save. With the “Events” layer selected, click “Add” on the `AsideSecond` zone. You should now see the JWPlayer Widget (Orchard attempts to make names more readable, hence the space between the “J” and “W”) as an option (Figure 6-2).

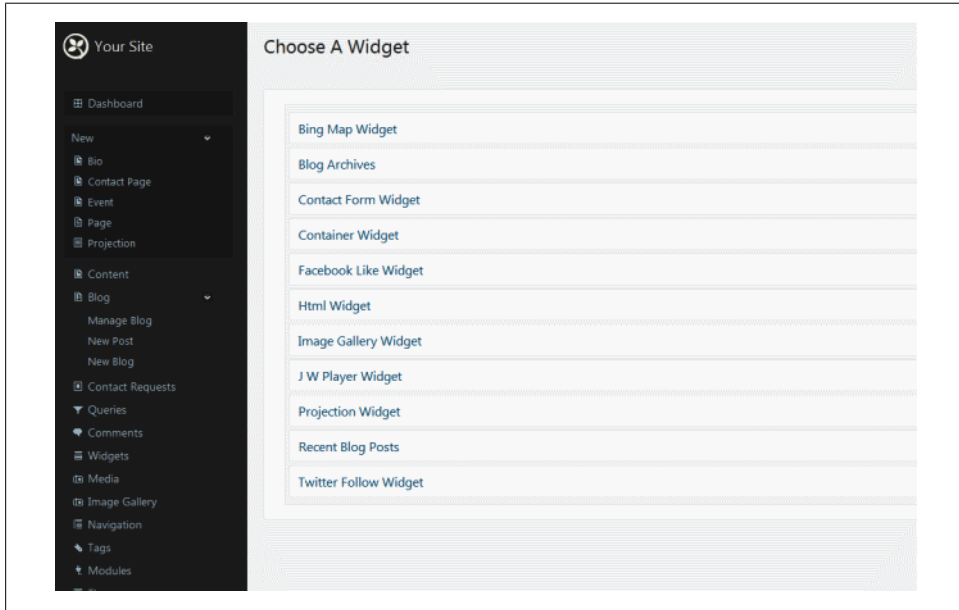


Figure 6-2. The new JWPlayer widget option

Click the “J W Player Widget” option and you’ll see the admin form we just created (Figure 6-3). Enter a height and width of 200 and 300 respectively. Set the “Player Source” to “~/Modules/JWPlayer/Flash/player.swf” and the “Media File” to “<http://youtu.be/tyHwA7IyjuY>” without the quotes. Enter the title “Daisy’s Videos.” Click Save and browse to the Events page (Figure 6-4). If you see an error that it can’t find the player, rebuild your project. You’ll now see a video player embedded into the `AsideSecond` zone.



If you are serving a video through your local IIS and getting a 404 error, the MIME type might not be mapped. To serve video files, such as *.mp4 files via IIS, open up the IIS property pages for MIME Types and add *.mp4 with a mime type of video/mp4.

Widget Metadata

Finally, we’ll want to update our *Module.txt* file so that the metadata used by the Dashboard correctly represents our work:

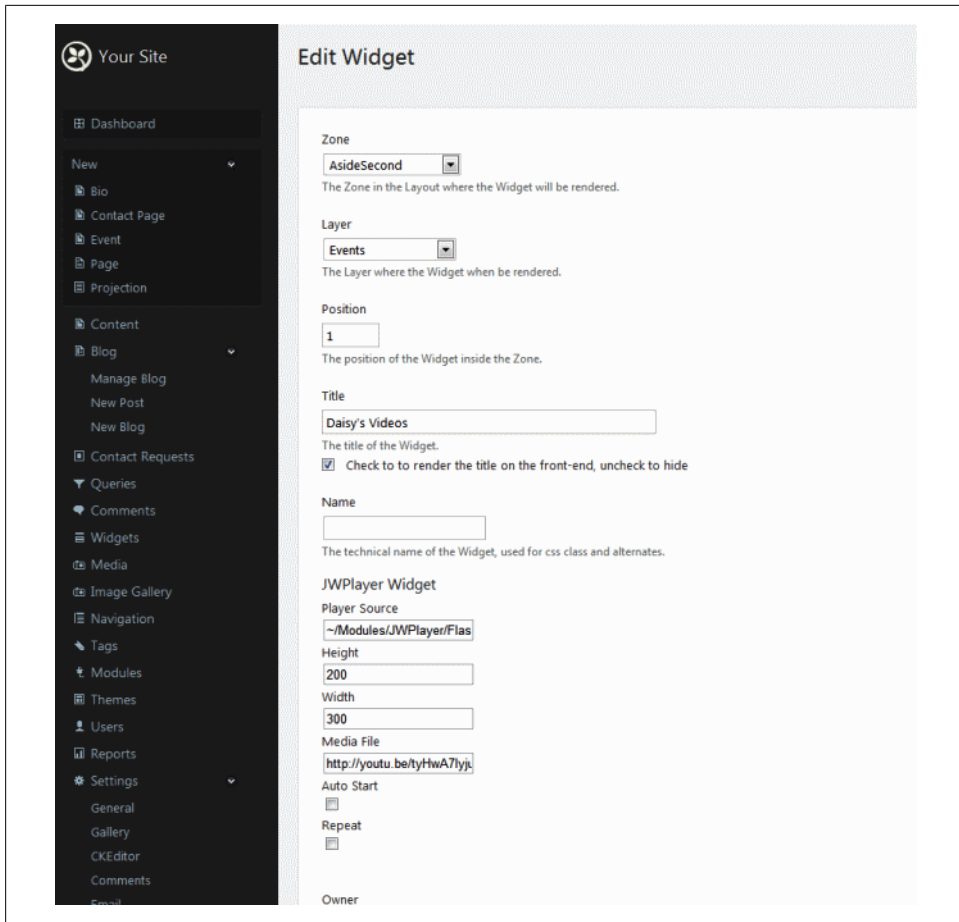


Figure 6-3. The JWPlayer widget admin form

```

Name: JWPlayer
AntiForgery: enabled
Author: John Zablocki
Website: http://dllhell.net
Version: 1.0
OrchardVersion: 1.4
Category: Media
Description: JW Flash Player Widget
Features:
  JWPlayer:
    Description: JW Flash Player Widget
  
```

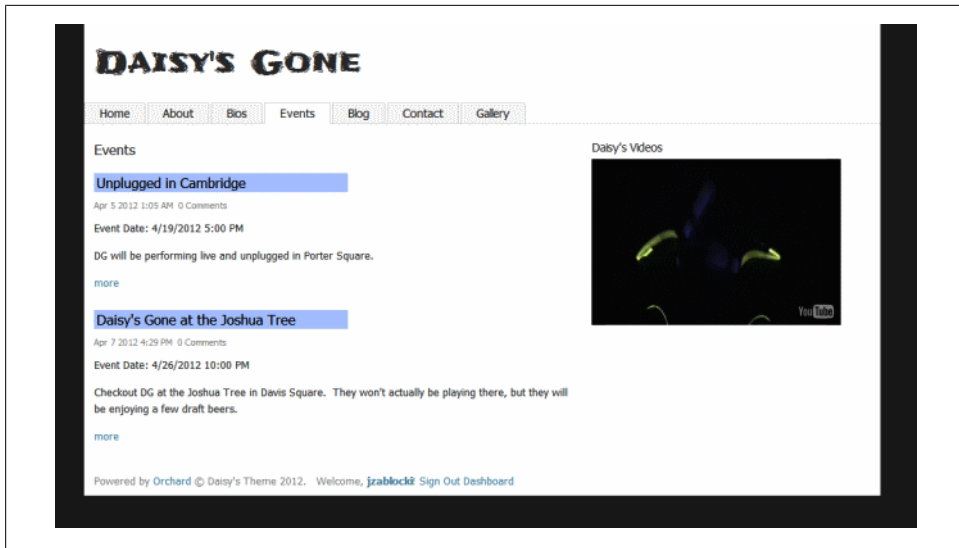


Figure 6-4. The finished JWPlayer widget

Summary

We've now seen how to create two types of modules in Orchard: fields and parts (by way of a widget). Fields offer granular levels of functionality that may be used when creating content types. Parts are similar to fields, but may encapsulate more features, such as the ability to be composed into a widget. We've only touched on some of what modules can do. In fact, there are Orchard modules that are responsible for creating and managing our Orchard modules.

Localization

Though there's no pressing need right now, it seems inevitable that Daisy's Gone will need to support other languages on its website. Given the (partially) Italian blood that flows through the founding members, it seems like Italian language support is a good place to start. *Porteremo Daisy per le persone buone d'Italia!*

Orchard Site Localization

ASP.NET MVC and ASP.NET WebForms both offer localization support by way of resource (.resx) files. Orchard instead takes a different approach that is a better match for its modular architecture. This approach is to use PO files, which are simple text files with a well-defined format. PO files are commonly used by other web frameworks, including many other CMS platforms.

Within an Orchard site, there are three areas you might need to localize. If your content creators are not native English speakers, you'll want to take advantage of the admin and module translation files that have been created by the Orchard community. If you have developed a module that you'll share in the Orchard Gallery, you might want to contribute translation files of your own. You'll also likely want to enable the `Localization` module to allow the content that you create to be localized.

Admin Localization

Whether you're looking to localize for Italian, Polish, or some other language, you'll first want to visit <http://orchardproject.net/Localize/> for a listing of available translation files. For each listed locale, you'll be able to download a ZIP file containing .po files with translations for the most common themes, modules, and admin utilities (including the command-line interface).

Browse to the Italian listing at <http://orchardproject.net/Localize/IT-it>. Click "Download the PO files" to get the ZIP file. After downloading, the easiest way to install a new translation is to open the ZIP file, and copy-and-paste the four directories

(*App_Data*, *Core*, *Modules*, and *Themes*) into the *root* of your website (*Orchard.Web*). Windows Explorer will prompt you to merge the directories, which you should do.

After merging, in the dashboard, select Settings→General. Under the “Default Site Culture” section, click “Add or remove supported cultures for this site.” Select the “it-it” culture that you just downloaded and click “Add.” Return to Settings→General and select the new site culture. Click “Save” and you’ll see that the dashboard no longer displays English text—even the confirmation message has changed (Figure 7-1).

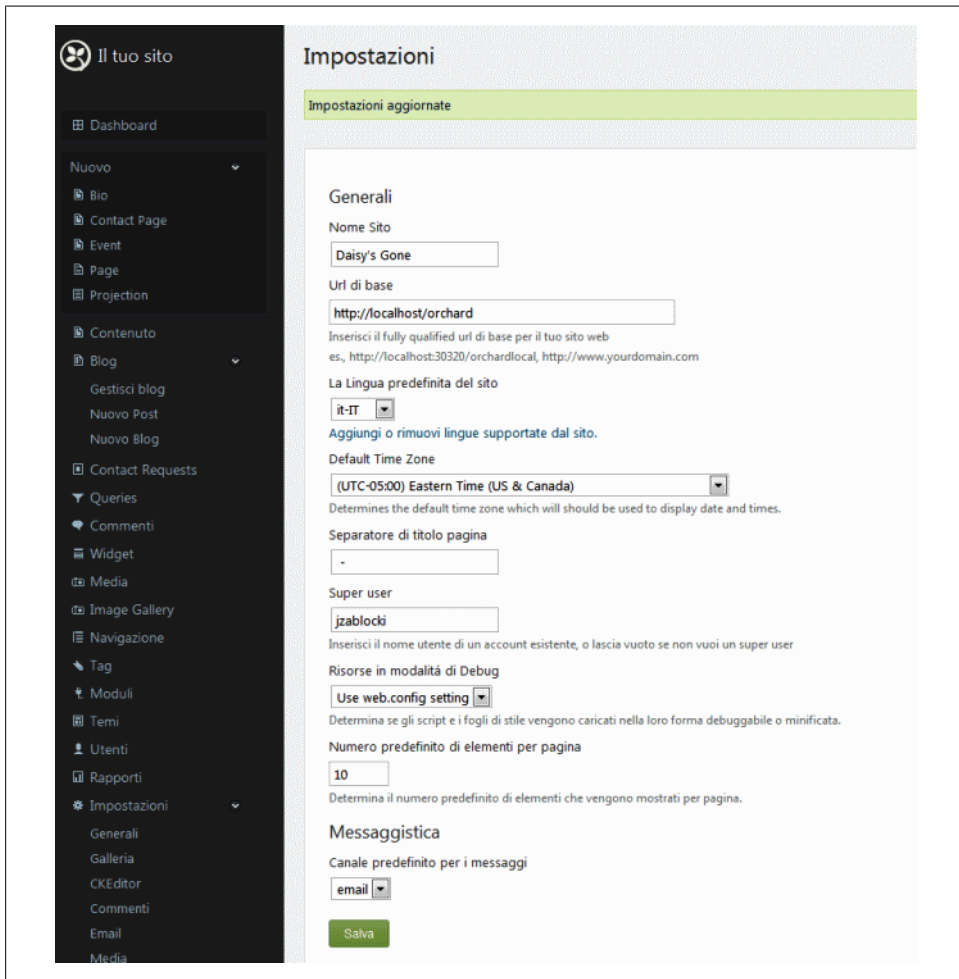


Figure 7-1. The Settings page after setting Italian as the default culture

There’s also a **Translation Manager** module that you could install from the Orchard Gallery. This module allows you to install translation bundles from the Orchard command-line interface.

All you need is the path to the downloaded ZIP file:

```
orchard> install translation C:\Users\John\Downloads\pl-PL.zip
```

PO Files

We won't dig too deeply into creating our own PO files, but we'll take a quick look at how the pieces come together. For more information on PO file options, Bing "PO Translation Files." There's even an open source PO file editor named Poedit available at <https://github.com/vslavik/poedit>.

To see how our Italian language translation works, let's examine the simple case of the "Save" button that appears on the General Settings page. Recall that we just clicked this button when changing our culture to Italian. In Visual Studio, locate the **Orchard.Core** project and navigate to the *Views* directory under the *Settings* directory. In the *Admin* directory, open the file *Index.cshtml*, where you'll find a button defined as follows:

```
<button class="primaryAction" type="submit">@T("Save")</button>
```

You might have seen the *T* method in some templates along the way. This is the primary localization method for Razor templates in Orchard. It's defined in the **WebViewPage** class that we explored in [Chapter 4](#). That method will locate culture specific translations for your view files by using the translation files you install (or create).



T is actually implemented as a property of type **Localizer** in **WebViewPage**. **Localizer** is *delegate*, which allows the property to use method call syntax in the templates.

In the **Orchard.Core** project, navigate to the *App_Data* directory. You might need to show hidden files. Expand the *Localization* directory to find the *IT-it* directory, which contains the translation file *orchard.core.po*. Open this file and search for "~/Core/Settings/Views/Admin/Index.cshtml." You'll find a few results. The one you want has an id-string (second line) of "Save."

In the snippet below, you can see values for the original text, and its translation, as well as the context (template) of this text. The most important values are the last two, which together are sufficient to translate the text of our button. Of course, providing context allows us to avoid collisions in our translations. If you change "*Salva*" to another value, and refresh the "Settings" page while the Italian culture is selected, you'd see your button text appear:

```
#: ~/Core/Settings/Views/Admin/Index.cshtml
#| msgid "Save"
msgctxt "~/Core/Settings/Views/Admin/Index.cshtml"
msgid "Save"
msgstr "Salva"
```

Content Item Localization

Finally, we probably want to translate the content on the pages that we've created so far. Before we continue, be sure to set your site back to English (unless you're feeling adventurous). Select Modules→Features (or *Moduli→Funzionalità*, if you kept Italian) and enable the Localization module. After it's enabled, you can select Content→Content Items, where you'll now see "+ New Translation" links under each page that we've created (the home, gallery, and about pages).

To add translations to our Bio and Event content types, we'll simply need to add the Localization content part to the definition of these types. Click Content→Content Types and click "Edit" in the row for each type. Under "Parts," click "Add parts," check "Localization," and click "Save." Now when you click "Content," you should see the "+ New Translation" link under each of our content items (Figure 7-2).

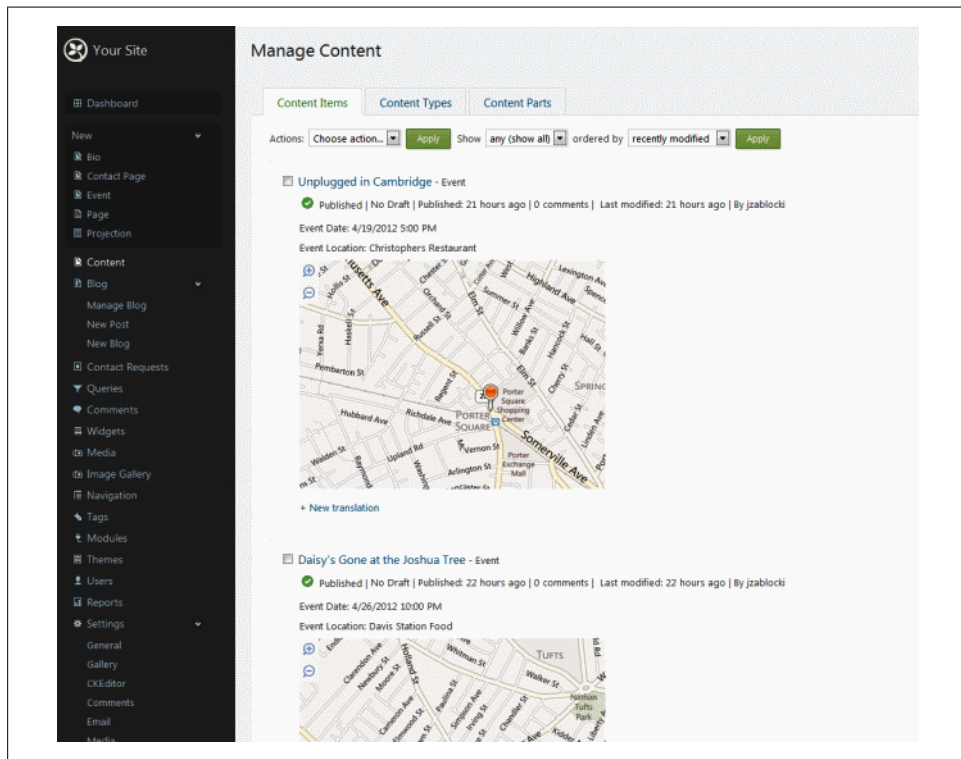


Figure 7-2. Content items awaiting translation

To translate the homepage, return to Content→Content Items and click "+ New Translation" under the home page listing. You'll see that there's now a "Content Localization" section on our page editor. Select "it-IT" from the list. Copy the content from your "Body" section and get a translation from <http://www.bing.com/translate> or your

favorite translator service. Paste the translated text back into the “Body” element and click “Publish Now.”

Browsing to the home page, you’ll see there’s now a “Translations: it-IT” link on the page above the Content zone (Figure 7-3). Click that link to see the Italian language translation of our home page. Note that our translated content is actually a unique content item. Therefore, when we create translations for content items, we are possibly duplicating our content maintenance efforts (for items that aren’t translatable). Another repercussion of creating a new item is that our home page alternate template won’t apply to our translation (Figure 7-4).



Figure 7-3. The translations listing

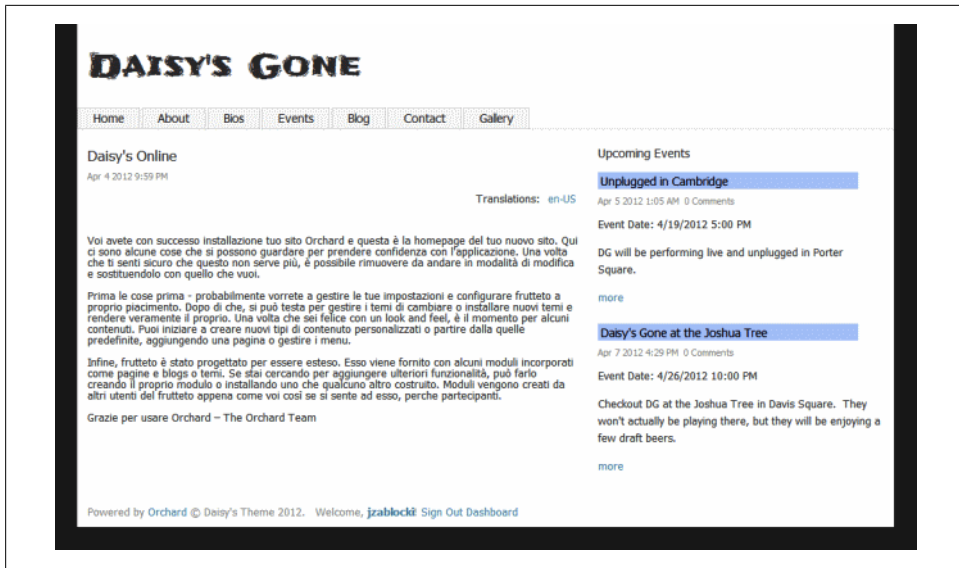


Figure 7-4. The translated homepage without an alternate template

Summary

As we've seen, Orchard has a great deal of flexibility for dealing with localization. However, not all aspects of localization have been fully baked. For example, if we created localized versions of our **Bio** items, our query for the "Bios" projection page would list the translations as well. Keep these limitations in mind as you plan your localization strategy. You might find yourself needing to code for a solution.

Maintaining Orchard Sites

We've seen that installing Orchard themes and modules simply requires copying files into well-known directories. If we were to create a second Orchard site, we could simply copy our modules and theme into the *Modules* and *Themes* directories under the *Orchard.Web* project for the new site. However, we've already seen a better way to share modules—the Orchard Gallery.

Packaging Themes and Modules

Though the Orchard team has supplied the Gallery with many useful modules, most of the nearly 400 packages (as of the writing of this book) have been contributed by Orchard users. Though we created our theme and modules to satisfy the needs of the Daisy's Gone website, there's nothing that would prevent us from sharing our creations with the Orchard community.



Our theme and modules were intentionally kept simple. It's arguably reasonable to share a module or theme with limited functionality. Just be sure not to call it a 1.0 product! In fact, sharing a pre-1.0 version is a good way to help shape your project by getting early feedback.

We'll first package up our theme. "Daisy's Theme" isn't likely to win any UI contests, but you never know who might find our minimalist approach to design appealing. To start the process, we'll return to the command line and enable the `Orchard.Packaging` module:

```
orchard> feature enable Orchard.Packaging
Enabling features Orchard.Packaging
Packaging was enabled
```

Once this module is enabled, we'll be able to use it to create, install, or uninstall packages. Orchard uses the Nuget package format (*.nupkg* files), which like most package

manager formats, are basically just ZIP files with some metadata included. Let's first create our theme's package file:

```
orchard> package create DaisysTheme C:\Dev\OrchardPkg
Package "C:\Dev\OrchardPkg\Orchard.Theme.DaisysTheme.1.0.nupkg" successfully created
```

If you browse to the directory `C:\Dev\OrchardPkg` in Windows Explorer you'll see the packaged theme. If you want to take a look inside to see what's been packaged up, simply change the extension from `.nupkg` to `.zip`. After that, you're able to explore the files. You'll see some metadata files along with the contents of our theme.

Recall that the "DaisysTheme" theme was based on the "TheThemeMachine" theme. The template files for that theme weren't packaged up with ours, so anyone who installs our theme will need to have that base theme installed. Fortunately that's a standard theme, but we can see that dependencies aren't automatically handled in theme packaging. Also notice that our alternate templates were also packaged for us.

Packaging up our two modules will be a similar task. We'll again use the command line to create Nuget packages. Let's start with our `Contrib.PlacesField` module:

```
orchard> package create Contrib.PlacesField C:\Dev\OrchardPkg
Package "C:\Dev\OrchardPkg\Orchard.Module.PlacesField.1.0.nupkg" successfully created
```

I made note of the fact that our modules shouldn't be considered version 1.0, but from the package name Orchard seems to think otherwise. What's actually happening is that the command to create the package is using the *Theme.txt* and *Module.txt* metadata files that we modified when creating our theme and modules.

If you thought to set a proper version while working on those chapters, your package names more accurately reflect the state of our extensions. After making sure we have alpha-appropriate version numbers, when we create the `JWPlayer` module package the filename looks right:

```
orchard> package create JWPlayer C:\Dev\OrchardPkg
Package "C:\Dev\OrchardPkg\Orchard.Module.JWPlayer.0.1.nupkg" successfully created
```

Now that we've packaged our modules, we have two choices for installing them on other sites. If you wish to share with the community, you'll need to create account on the official Orchard Gallery site located at <http://gallery.orchardproject.net/>. Once you have that account, you can upload and manage your theme or module. Once in the gallery, you can download and install your modules on any of your Orchard sites just as you did with modules such as `Bing.Maps`.

If your development efforts were paid for by a client, or you've created a business use module that shouldn't be exposed in a public gallery, you can still make use of the packages that we've created. We'll again return to the command line where we're able to install and uninstall modules from `.nupkg` files. In a new site, simply run the `install` command:

```
orchard> package install Orchard.Module.JWPlayer
C:\Dev\OrchardPkg\ Orchard.Module.JWPlayer.0.1.nupkg
```

You could also install your packaged themes and modules using the admin dashboard. If you click either “Modules” or “Themes” on the admin menu and select the “Installed” tab, you’ll find links to “Install a module from your computer” and “Install a theme from your computer,” respectively. Simply browse to the *.nupkg* file and select it to install it. Once installed, enable it when prompted.

Deploying Orchard Sites

Now that we know how to package up our extensions, we can consider how and to where we’ll deploy our Orchard sites. In general, deploying an Orchard site is no more complex than deploying your own ASP.NET applications. You simply need to copy the content files and binaries from the **Orchard.Web** project to your server.

To create a deployment package from the source, you could use Visual Studio’s publishing feature, found under Build→Publish. You can deploy to your filesystem to get a package that you can copy to a server. Another option is to clone the source, as described earlier in this book, and run the *build.cmd* file that’s found at the *src* root. This build file isn’t part of the zipped source bundle that we’ve been using.

Shared Hosting

Deploying an Orchard site to a shared hosting provider requires a bit more consideration than a typical ASP.NET application. For security reasons, many shared hosts won’t allow your applications to run under **Full** trust. Orchard will run under **Medium** trust, but is optimized for **Full**. If you are able to find a host that supports **Full** trust, you’ll be better off in terms of site performance, as Orchard will be allowed to use all of its optimization techniques.

If you take a look at the worker process (*w3wp.exe*) for your app pool in **Windows Task Manager**, you’ll probably notice memory pressure is a bit higher than you would have expected. My development site is currently running at about 300MB, but I’ve also seen it reach nearly 700MB. Most shared hosts don’t advertise hard memory limits, but when they exist you’ll likely see your app pool recycled more frequently than you’d prefer.

Another cause of app pool recycling is inactivity. If your site isn’t accessed for some period of time, the worker process will be recycled. Again, shared hosts might be more aggressive as to how frequently your app pool is recycled. Since Orchard’s startup time tends to be relatively slow, there’s a **Warmup** module that will periodically generate semi-static versions of pages that should always be served quickly.

Enable the **Warmup** module by visiting Modules→Features and clicking “Enable” on “Warmup.” After enabled, click “Performance” under the “Settings” menu option. In the “Warmup” tab, enter relative URIs (e.g., “/john-zablocki”) for the pages you want to be cached. You can choose to generate these pages on a schedule, as well as when content is published.

Dedicated Hosting

If your Orchard site is a hobby site, such as DaisysGone.com, you're unlikely to be willing to fork over hundreds of dollars each month to have a dedicated server. If you can afford it, though, this option will give you full control over your server and the ability to configure the hardware to meet your site's needs.

A cheaper alternative is to get a Virtual Private Server (VPS). You can generally get your own VPS, complete with **Remote Desktop** access, for under \$30. For that price, you'll find 1-2GB RAM and 25-50GB of storage. However, that usually doesn't include SQL Server access or storage. You'll often have to pay a separate licensing fee for SQL Server with a VPS solution. By contrast, shared hosts will usually include a shared SQL Server in their plans.

Cloud Hosting

Deploying an Orchard site to the cloud is a reasonably straightforward effort. On EC2, you'll just launch your instance and move your Orchard files out via FTP or your preferred file transfer method. Again, you'll have to consider the cost of resources, in particular, SQL Server.

Another script that may be found in the source root when cloning the repository is *ClickToBuildAzurePackage.cmd*. Assuming you've installed the Azure SDK, this batch file will create an Azure package under an *artifacts* directory. In this Azure ZIP file, you'll find a file named *ServiceConfiguration.cscfg* that you can edit with details from your Azure account. There is a detailed tutorial for deploying to Azure in the Orchard documentation.

Multi-Tenancy

Orchard offers a **MultiTenancy** module, which allows a single instance of Orchard to host multiple Orchard sites. This module is best suited for shared hosting, where you might likely pay for each dedicated app pool you are allotted. Your site that's infrequently hit might also benefit from the one that is hit often and keeps both sites alive. Of course, sharing an app pool means one site could bring down another as well.

Modules and Performance

When you install a module from the gallery, you're allowing another developer's code to run inside of your application. All developers occasionally write some bad code. You might think that the **Lolcats** module you just installed is innocent enough, but it actually spawns a background thread that downloads and caches all the cats posted to <http://icanhascheezburger.com/> for the past five years.

If you start to notice performance problems after installing or updating a module, try disabling that module. It's more than possible that Orchard's performance could be impeded by an errant module. Also, limit the number of modules you have installed or deployed as ASP.NET will perform better with fewer assemblies to load.

Workflow

Once your site is deployed, you'll need to consider how you are going to manage content and code. Your process will vary quite a bit based on whether you're creating extensions or just content types. Regardless of which use case best describes your Orchard experience, you're likely going to want to stage your site.

If you're maintaining an Orchard site that's effectively just Orchard out of the box, having regular backups is probably sufficient. There's not much of a need to stage a site that isn't actively being developed. It's certainly useful, however, to have a place to test modules from the gallery before enabling them on a live site.

If you're creating custom content types, alternate templates, or otherwise modifying content, you'll benefit from having a staging version of your site. Changing content types certainly could impact the behavior of a site. It's best to experiment with these modifications in a safe place.

If you're actively developing an Orchard site, whether by creating widgets, modules, or themes, you'll want a standard development experience. Of course, you'll use source control. You'll want a local version of the site and probably a QA and staging version as well. In short, treat Orchard development as you would development with any of your other web projects.

Upgrading

New versions of Orchard are released periodically. Unfortunately, Orchard doesn't yet have in-place upgrades as you might find in other popular CMS platforms. Instead, the basic upgrade path involves creating a new site deployment from the new release. In other words, you'll extract the release package (ZIP file of source or precompiled web files) to a local directory.

Create a new IIS site that points to this new release directory, but don't go through the "Getting Started" wizard. Copy the themes, media, and App_Data from your live site to this new site's directory. If you've installed any modules, copy those as well.

To test the upgrade on the database, you'll probably want a local copy of that as well. If you're using SQL CE, you've already created a copy in the previous step. If you're using SQL Server, you could just restore locally from a production backup. You'll also have to update the connection string information in *Settings.txt* in the directory for your recipe, which is found under *App_Data*.

Now browse to the site and login. Orchard should have run any migrations for modules that might have been upgraded in this new release. Assuming everything went well (i.e., you performed full QA of your site), you're ready to deploy the update to production. Back up the existing site and its database and follow any of the deployment methods detailed in this chapter.

Summary

The Orchard team will no doubt continue to improve on the experience of developing and maintaining Orchard sites. As the community grows, you'll likely see that hosting companies are adopting Orchard as a hosting add-on. Some shared hosting companies already offer Orchard sites via one-click installation from application galleries. At the same time, as the codebase moves from version 1.4 to the next major release, many of the considerations we discussed in this chapter may be moot.

Conclusion

At this point in our exploration of Orchard we've validated the 80-20 rule, which suggests that 80% of effects come from 20% of causes. As it relates to Orchard, 80% of the websites that could be built with Orchard could probably be built with 20% of Orchard's functionality. While we've certainly covered more than 20% of what Orchard is capable of, many readers may have found that their needs were met after learning how to create content types and projection pages.

Along the way, we've certainly left some stones unturned. Queries and projections, for example, could have occupied an entire chapter. Module development could easily fill an entire book. To cover 100% of Orchard's functionality would require a book with at least three to four times the number of pages that are found in this text. Still, we've been exposed to most of the common use cases for building sites with Orchard.

A few months ago when I started writing this book, the Daisy's Gone website was the "killer" Orchard app for me. Now I realize that so many of the ideas I've had for hobby sites over the years could easily be cooked by Orchard with very minimal effort. I hope you've also come to the same conclusion—that Orchard is a great platform for building out your next idea.

About the Author

John Zablocki is a Developer Advocate at Couchbase. He is the organizer of Beantown ALT.NET and a former adjunct at Fairfield University. John holds an M.S. in Computer Science from Rensselaer Hartford. He has worked at startups throughout his career and is interested in the intersection of .NET and open source. Online, John can be found at <http://about.me/johnzablocki>. Offline, he can be found too infrequently around Boston, with his dog, Lady; daughter, MaryKatherine; and his Martin acoustic.

